Міністерство освіти і науки України

Харківський національний університет імені В.Н. Каразіна

Навчально-науковий інститут комп'ютерних наук та штучного інтелекту

Рекомендовано до захисту:

протокол засідання кафедри

№ ___ від ____.____.2025 р.

Завідувач кафедри ІПСіТ

_____ Олешко О. І.

(підпис) (прізвище та ініціали)

Пояснювальна записка

до кваліфікаційної роботи бакалавра

на тему «Розробка проблемно-орієнтованої мови для генерації Mock-сервісів у сервіс-орієнтованих системах»

Захищено на засіданні ЕК № _____

протокол № ____ від ___.____.2025 р.

Оцінка _____ /_____

Голова ЕК

_____ Фесенко Г. В.

(підпис) (прізвище та ініціали)

Виконав:

студент 4 курсу, групи КС-41

Спеціальності:

F3 комп'ютерні науки

(шифр і назва спеціальності підготовки)

Бережний Дмитро Олександрович

(прізвище, ім'я та по батькові, підпис)

Керівник к.т.н., доцент Гамзаєв Р. О.

(ступень, звання, прізвище та ініціали, підпис)

Рецензент к.т.н., ст. викл. Марчук А.В.

(ступень, звання, прізвище та ініціали, підпис)

Харків – 2025

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

Харківський національний університет імені В.Н.Каразіна

Навчально-науковий інститут комп'ютерних наук та штучного інтелекту
Кафедра інтелектуальних програмних систем і технологій
Рівень вищої освіти (освітньо-кваліфікаційний рівень) бакалавр
Спеціальність F3 Комп'ютерні науки

ЗАТВЕРДЖУЮ

Завідувач кафедри ІПСіТ

_____ Олег ОЛЕШКО
підпис ім'я, прізвище

" 11 " січня 2025 року

З А В Д А Н Н Я
НА КВАЛІФІКАЦІЙНУ РОБОТУ

Бережний Дмитро Олександрович
(прізвище, ім'я, по батькові студента)

1. Тема роботи: «Розробка проблемно-орієнтованої мови для генерації Mock-сервісів у сервіс-орієнтованих системах»

керівник роботи Гамзаєв Рустам Олександрович, кандидат технічних наук, доцент кафедри інтелектуальних програмних систем і технологій
(прізвище, ім'я, по батькові, науковий ступінь, вчене звання)

затверджені наказом по університету від " 16 " квітня 2025 року № 4101-5/962

2. Строк подання студентом роботи 2 червня 2025 року

3. Перелік питань, які потрібно розробити:

Ознайомитись з особливостями проблемно-орієнтованих мов для генерації Mock-сервісів у сервіс-орієнтованих системах, провести огляд існуючих методів існуючих методів розробки DSL та трансляції коду, проаналізувати специфіку імплементації алгоритмів побудови абстрактного синтаксичного дерева (AST) та трансляції описів на основі DSL у Java, створити модель проблемно-орієнтованої мови, що включає формальний опис граматики і визначення семантичних правил, провести програмну реалізацію

та експериментальне дослідження використання власної проблемно-орієнтованої мови (DSL), що задається з інтерфейсу користувача, з подальшим синтаксичним аналізом на серверній стороні за допомогою ANTLR для автоматизованого налаштування Mock-ендпоінтів у середовищі WireMock.

4. План роботи

№ з/п	Назви етапів роботи
1	Формулювання мети та повного переліку завдань для виконання кваліфікаційної роботи (КР).
2	Пошук та аналіз інформаційних джерел за темою КР.
3	Визначення особливостей проблемно-орієнтованих мов.
4	Розробка моделей синтаксичного аналізу та трансляції, що включають формальний опис граматики проблемно-орієнтованої мови з використанням ANTLR, побудову абстрактного синтаксичного дерева та відображення його структури у Java-код.
5	Програмна реалізація системи, яка на основі DSL-опису заданого в інтерфейсі користувача, транслює команди за допомогою ANTLR у Java-код, та доменну мову WireMock з генерацією mock-ендпоінтів.
6	Інтегрувати розроблене рішення в Docker-середовище, забезпечити його базову працездатність та провести перевірку взаємодії основних компонентів.
7	Підсумок отриманих результатів, проведення тестування, аналіз отриманих результатів та оформлення пакету документів для процедури захисту КР в екзаменаційній комісії ННІ КН та ШІ.

5. Дата видачі завдання 11 січня 2025

Студент _____ Д. О. Бережний
(підпис) (ініціали, прізвище)

Керівник роботи _____ Р. О. Гамзаєв
(підпис) (ініціали, прізвище)

АНОТАЦІЯ

Пояснювальна записка містить 83 сторінки, 23 рисунки, 5 таблиць, 1 додаток, 25 джерела інформації.

Робота присвячена розробці проблемно-орієнтованої мови для генерації мок-сервісів у сервіс-орієнтованих системах. Метою дослідження є створення зручного інструменту, що дозволяє автоматизувати процес моделювання поведінки зовнішніх компонентів для тестування програмного забезпечення. У межах реалізації розроблено спеціалізовану мову для опису імітованих кінцевих точок. Синтаксичний аналіз реалізовано з використанням ANTLR. Серверну логіку написано мовою Java. Розроблено вебінтерфейс редагування коду на основі Monaco Editor. Також забезпечено повну інтеграцію з інструментом WireMock. Усі компоненти об'єднано у контейнеризоване середовище Docker.

У результаті створено систему, яка дозволяє описувати та імітувати кінцеві точки. Реалізовано підтримку сценаріїв запитів і відповідей, а також умов для їхніх тіл. Реалізація дає змогу моделювати помилки й затримки, обробляти запити у режимі проксі та додавати вебхуки. Новизна дослідження полягає у поєднанні розробки власної проблемно-орієнтованої мови, генерації конфігурацій, гнучкого користувацького інтерфейсу та можливості запуску у відокремленому середовищі. Розроблене рішення може бути використано для прискорення розробки та тестування у сервіс-орієнтованих системах, для освітніх цілей, або як основа для комерційних продуктів у сфері забезпечення якості програмного забезпечення.

Ключові слова: DSL, SOA, МОК-СЕРВІС, ГЕНЕРАЦІЯ, КІНЦЕВА ТОЧКА, ТЕСТУВАННЯ, СЦЕНАРІЙ, ПАРСИНГ, WIREMOCK, ANTLR, JAVA, MONACO EDITOR, АВТОМАТИЗАЦІЯ, СИНТАКСИЧНИЙ АНАЛІЗ.

ABSTRACT

The explanatory note contains 83 pages, 23 figures, 5 tables, 1 appendix, and 25 information sources.

The work is dedicated to the development of a domain-specific language for generating mock services in service-oriented systems. The research aims to create a convenient tool that automates the process of modeling the behavior of external components for software testing. Within the scope of the implementation, a specialized language was developed for describing mock endpoints. Syntactic analysis is implemented using ANTLR. The server logic is written in Java. A web-based code editing interface was developed based on Monaco Editor. Full integration with the WireMock tool has also been ensured. All components are unified within a containerized Docker environment.

The result is a system that enables the description and simulation of endpoints. It supports request and response scenarios, as well as conditions for their bodies. The implementation allows for modeling errors and delays, handling requests in proxy mode, and adding webhooks. The novelty of the research lies in the combination of developing a custom domain-specific language, generating configurations, providing a flexible user interface, and enabling operation in an isolated environment. The developed solution can be used to accelerate development and testing in service-oriented systems, for educational purposes, or as a foundation for commercial products in the field of software quality assurance.

Keywords: DSL, SOA, MOCK SERVICE, GENERATION, ENDPOINT, TESTING, SCENARIO, PARSING, WIREMOCK, ANTLR, JAVA, MONACO EDITOR, AUTOMATION, SYNTACTIC ANALYSIS.

ЗМІСТ

ПЕРЕЛІК СКОРОЧЕНЬ, УМОВНИХ ПОЗНАЧЕНЬ, ТЕРМІНІВ 6
ВСТУП 7
1 АНАЛІЗ ІСНУЮЧИХ РІШЕНЬ ТА ДОСЛІДЖЕНЬ 11
1.1 Проблемно-орієнтовані мови та підходи до їх створення 11
1.2 Порівняльний аналіз мок-сервісів, що існують 14
1.3 Обґрунтування актуальності та доцільності дослідження 20
2 ОГЛЯД ПРОЕКТУВАННЯ ТА ПРОГРАМНОЇ РЕАЛІЗАЦІЇ 23
2.1 Розробка проблемно-орієнтованої мови 23
2.1.1 Визначення цілей, вимог до мови та область застосування 23
2.1.2 Формалізація граматики та правил опису тестових кінцевих точок. 27
2.1.3 Опис основних конструкцій мови 33
2.2 Аналіз DSL-коду та генерація конфігурацій 44
2.2.1 Парсинг вхідного коду та відповідні інструменти 44
2.2.2 Синтаксичний аналіз DSL та обробка результатів парсингу 45
2.2.3 Використання WireMock для генерації мок-сервісів на основі DSL 48
2.3 Інтеграція з сервісом WireMock 50
2.3.1 Функціональні можливості сервісу 50
2.3.2 Налаштування WireMock для створення мок-сервісів 52
3 АНАЛІЗ РЕЗУЛЬТАТІВ ДОСЛІДЖЕННЯ 55
3.1 Опис архітектури рішення 55
3.1.1 Компонентна структура системи 55
3.1.2 Ролі та взаємозв’язки між модулями 57
3.2 Опис програмного рішення 58
3.2.1 Інтерфейс користувача для редагування DSL-коду 59
3.2.2 Використання Monaco Editor для покращення процесу редагування 63
3.2.3 Передача коду до серверної логіки обробки 65

3.3 Оцінка одержаних результатів 66
ВИСНОВКИ 68
ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ 70
ДОДАТОК А. ВИХІДНИЙ КОД 73

ПЕРЕЛІК СКОРОЧЕНЬ, УМОВНИХ ПОЗНАЧЕНЬ, ТЕРМІНІВ

SOA – Service-Oriented Architecture.

DSL – Domain-Specific Language.

HTTP – Hypertext Transfer Protocol.

ANTLR – Another Tool for Language Recognition.

IoT – Internet of Things.

GPL – General Purpose Language.

HTML – HyperText Markup Language.

SQL – Structured Query Language.

API – Application Programming Interface.

JSON – JavaScript Object Notation.

HTTPS – Hypertext Transfer Protocol Secure.

gRPC – Google Remote Procedure Call.

GraphQL – Graph Query Language.

CI/CD – Continuous Integration/Continuous Deployment.

UI – User Interface.

CLI – Command Line Interface.

REST – Representational State Transfer.

YAML – YAML Ain’t Markup Language.

URL – Uniform Resource Locator.

XML – Extensible Markup Language.

AST – Abstract Syntax Tree.

IP – Internet Protocol.

JAR – Java Archive.

JVM – Java Virtual Machine.

ВСТУП

Наразі більшість сучасних програмних платформ створюється за певними архітектурними підходами. Вони дозволяють розробникам будувати масштабовані, гнучкі й зручні в обслуговуванні рішення. Одним із таких є сервісно орієнтована архітектура, або service-oriented architecture (SOA). Вона пропонує спосіб організації програми у вигляді окремих сервісів, які взаємодіють між собою та виконують завдання через чітко визначені інтерфейси. Це вже давно не новинка, але й досі актуальна методика. Всупереч активному впровадженню інших ефективних рішень, SOA залишається хорошою моделлю для побудови систем із високим рівнем інтеграції, повторним використанням компонентів та централізованим управлінням. Саме тому вона продовжує відігравати важливу роль у розробці програмних систем, зокрема у великомасштабних корпоративних технологіях.

Розробка та підтримка таких платформ має свої особливості. Однією з ключових складностей є тестування. Щоб перевірити, як працює певна компонента, часто потрібна робота інших складників, від яких вона залежить. Але ці модулі можуть бути ще не готові, недоступні або нестабільні. Саме для того, щоб не залежати від таких зовнішніх частин, команди часто створюють так звані мок-сервіси. Це тимчасові замінники, які імітують поведінку реальних рішень. Такі сервіси дозволяють симулювати відповіді залежних підсистем. Це особливо корисно при інтеграційному тестуванні або роботі з неготовими середовищами. Також це зручно, бо дозволяє локалізовано перевіряти функціональність без необхідності доступу до справжніх частин системи. Проте з часом ця задача ускладнюється. Особливо, коли потрібно підтримувати десятки або навіть сотні таких штучних рішень для різних сценаріїв.

Сучасні засоби, такі як WireMock, Hoverfly чи MockServer, допомагають створювати мок-сервіси для тестування та розробки. З часом ручне налаштування таких компонентів може стати трудомістким і неефективним.

Аналіз сучасної вітчизняної та закордонної науково-технічної літератури свідчить про зростання інтересу до інструментів моделювання сервісів. Зокрема через використання проблемно-орієнтованих мов, або domain-specific language (DSL), як одного з ефективних підходів. Щоб спростити цей процес і зробити його більш зрозумілим для широкого кола, все частіше застосовують DSL. Це – спеціалізовані мови, розроблені під конкретні завдання, які дозволяють описувати певні задачі лаконічно та просто. Ближче до того, як люди звикли описувати бізнес-логіку, а не до технічної реалізації.

У світі вже існує чимало прикладів створення DSL. Вони дозволяють ефективно моделювати процеси, спрощувати налаштування, зменшувати обсяг рутинної роботи, швидко налаштовувати системи та зменшити кількість помилок, які виникають під час ручного налаштування. Це зручно, економить час і сили команди. Однак у сфері SOA подібні рішення зустрічаються рідше, ніж у мікросервісах чи фронтенд розробці. Там DSL вже давно стали звичним способом спрощення конфігурацій та сценаріїв. А ті, що все ж з’являються, часто виявляються надто складними у використанні, недостатньо гнучкими або погано масштабуються в реальних проєктах.

Об'єктом дослідження є процес конфігурації та управління мок-сервісами в рамках SOA. Зокрема, будуть досліджуватися методи та інструменти для ефективного налаштування замінників справжніх компонентів системи, які імітують їхню поведінку. Вивчатиметься роль цих імітаційних систем у забезпеченні ізоляції тестових середовищ та перевірці взаємодії між складниками. Важливим є огляд здатності до симуляції різноманітних сценаріїв роботи, включно з нестабільними або помилковими ситуаціями. Основна увага приділяється розробці інструментів для автоматизації процесу налаштування платформ для імітації поведінки. Вони зі свого боку зумовлюють спрощення та прискорення тестування в розподілених системах.

Предметом дослідження є створення проблемно-орієнтованої мови, яка значно спрощує опис, налаштування та генерацію мок-сервісів у межах SOA.

Вона дозволятиме визначати кінцеві точки, їхні HTTP-методи, запити, відповіді. Також це рішення має конфігурувати затримки, помилки та умови для вхідних запитів, що необхідні для моделювання поведінки сервісів. Граматика реалізуватиметься за допомогою Another Tool for Language Recognition (ANTLR). Це інструмент, що дозволяє формалізувати синтаксис, генерувати парсер та забезпечити перевірку правильності коду. Обробка повинна відбуватися на сервері Java. Ця розробка інтегруватиметься з WireMock для автоматичного створення мок-сервісів на основі написаного DSL-коду. Запуск виконуватиметься в контейнерах Docker, що забезпечить ізольоване середовище. Важливою частиною також є реалізація вебінтерфейсу з використанням Monaco Editor, який гарантує зручну взаємодію користувача з DSL. Це дослідження спрямовано на ефективне моделювання поведінки сервісів для тестування SOA. Тому воно і є фундаментальним для подальшого удосконалення технологій розробки. Таким чином, робота охоплює всі ключові компоненти: від визначення мови до практичного застосування.

Метою дослідження є створення інноваційного інструменту для автоматизації тестування SOA через використання спеціалізованої мови опису поведінки сервісів. У межах реалізації передбачається сформувати сучасну DSL з чіткою структурою, збудувати серверний компонент для його обробки, реалізувати графічний інтерфейс взаємодії користувача та забезпечити повноцінне з’єднання всіх частин у єдине рішення. Результатом стане гнучка, масштабована система, що дозволяє описувати кінцеві точки, сценарії запитів і відповідей. А також: логіку помилок і затримок, керувати цими сценаріями через інтерфейс, переглядати історію запитів і зберігати створені налаштування. Цей підхід спрямований на швидку трансформацію вимог у зрозумілі тестові сценарії без потреби у складних налаштуваннях. Його розробка зменшить людський фактор, скоротить час на підготовку тестового середовища і мінімізує помилки. Результат відкриє можливості для ефективнішої взаємодії між командами та підвищення якості розробки сучасних застосунків.

Сучасна розробка програмного забезпечення потребує інструментів, що мінімізують витрати часу на налаштування середовища тестування та зменшують складність інтеграції між компонентами. У сфері SOA все ще бракує універсального рішення, яке дозволяло б швидко і зручно описувати поведінку зовнішніх сервісів у формі, зрозумілій усім учасникам розробки. Зростає потреба в стандартизованих, доступних і масштабованих підходах до моделювання мок-сценаріїв. Це пояснює актуальність цього дослідження, що спрямоване на створення практичного засобу автоматизації.

Новизна дослідження полягає у комплексному поєднанні DSL, засобів синтаксичного аналізу, генерації конфігурацій та користувацького інтерфейсу в одному рішенні. Створена мова не лише дозволить легко формалізувати сценарії взаємодії, а й інтегруватися з наявними інструментами розробки, що забезпечить зручність у практичному використанні. Така цілісна структура відкриє можливості для подальшого розвитку в напрямках тестування мікросервісів, IoT-систем тощо.

1 АНАЛІЗ ІСНУЮЧИХ РІШЕНЬ ТА ДОСЛІДЖЕНЬ

1.1 Проблемно-орієнтовані мови та підходи до їх створення

Проблемно-орієнтована мова – це комп’ютерна мова, розроблена для вирішення завдань у конкретній прикладній області, на відміну від мов загального призначення, або general-purpose language (GPL). Вони застосовуються до широкого спектра задач. DSL фокусуються на спрощенні взаємодії з певною проблемою, для того, аби експерти у цій галузі могли описувати задачі на високому рівні абстракції. У дослідженні Фаулера зазначається, що DSL спрощують взаємодію з предметною областю, зменшуючи складність програмування для специфічних задач [1]. Наприклад, HTML є DSL для створення вебсторінок, а SQL – для роботи з базами даних [2].

DSL поділяють на два основних типи: внутрішні та зовнішні. Внутрішні вбудовуються в реальну мову програмування, використовуючи її синтаксис для створення зручних API чи fluent interfaces. Наприклад, у Ruby чи Lisp внутрішні DSL широко застосовуються завдяки гнучкості цих мов [3]. Зовнішні мають власний синтаксис і потребують окремого парсера для обробки, що робить їх більш незалежними, але складнішими у реалізації [4].

DSL широко застосовуються в тестуванні для спрощення створення тестових сценаріїв. Наприклад, Gherkin, який використовується в Cucumber, дозволяє описувати поведінку системи у вигляді зрозумілих потоків виконання. У документації підкреслюється, що цей інструмент забезпечує читабельність і доступність сценаріїв навіть для нефахівців у програмуванні [5]. Приклад його синтаксису наведено на рисунку 1.1.

```
Feature: Guess the word

  # The first example has two steps
  Scenario: Maker starts a game
    When the Maker starts a game
    Then the Maker waits for a Breaker to join

  # The second example has three steps
  Scenario: Breaker joins a game
    Given the Maker has started a game with the word "silky"
    When the Breaker joins the Maker's game
    Then the Breaker must guess a word with 5 characters
```

Рисунок 1.1 – Приклад синтаксису DSL Gherkin для інструменту Cucumber

Іншим прикладом можна навести QueryDSL для Java. Вона надає розробникам можливість створювати безпечні за типом, статично перевірні запити. Завдяки цьому зменшується залежність від технічних деталей і знижуються ризики помилок, пов'язаних з використанням рядкових запитів. Завдяки автогенерації метамоделей, які відображають структуру предметних класів, QueryDSL сприяє легкому формулюванню складних запитів, не вдаючись до написання SQL-коду вручну. Згідно з офіційною документацією QueryDSL, це підвищує надійність коду шляхом перевірки запитів на етапі компіляції [6]. Такі мови зменшують залежність від технічних деталей, дозволяючи спеціалістам зосередитися на бізнес-логіці.

Створення DSL залежить від типу мови та цілей її використання. Для внутрішніх DSL розробники використовують особливості синтаксису хост-мов, наприклад Ruby чи Groovy. Це робиться для того, щоб створювати API, які виглядають як окрема мова. Наприклад, бібліотека JMock у Java використовує внутрішній DSL для визначення очікувань у тестах.

Для зовнішніх DSL необхідні інструменти для створення парсерів, такі як ANTLR, Xtext або MPS [7]. У дослідженні Парра зазначається, що ANTLR є ефективним для генерації парсерів завдяки підтримці різних мов програмування [8].

Процес створення зовнішньої DSL включає певні етапи. У статті Мерніка та співавторів підкреслюється, що чітке структурування кроків розробки DSL підвищує її ефективність та зручність [9].Першим кроком є фаза прийняття рішення. ЇЇ головною метою є визначити потребу у створенні мови, оцінити доцільність і сформулювати основні вимоги. Тут важливо зважити, чи справді нова розробка спростить роботу з визначеною областю застосування, а також хто буде її кінцевим користувачем. Далі йде фаза аналізу, у якій відбувається детальне вивчення предметної області. Тут необхідно виявити ключові характеристики, поняття та типові сценарії використання, які DSL має охопити. Саме цей аналіз допомагає сформувати основу майбутньої мови.

Наступний крок містить у собі ідею розробки лексики, синтаксису та визначення граматики. Вона повинна бути не лише логічно коректною, але й зрозумілою для користувачів, що працюватимуть із нею. На цьому етапі треба врахувати специфіку цільової аудиторії, щоб зробити DSL максимально зручною та природною у використанні.

Далі переходять до технічної реалізації. Це передбачає створення певного парсера, який дозволить обробити написаний користувачем код DSL. Для цього часто використовують спеціалізовані інструменти, наприклад такі як ANTLR, чи інші, що значно полегшують обробку граматичних конструкцій.

І, останнім важливим кроком є генерація вихідного коду або конфігурацій, які слугують основою для подальшої роботи цільової системи. Наприклад, на цьому етапі код DSL може бути перетворений у формат JSON для використання в різноманітних інструментах. У сукупності ці кроки формують повноцінний цикл створення DSL.

Існує й інший підхід – створення графічних проблемно-орієнтованих мов. Такі рішення дозволяють працювати не з текстом, а з візуальними елементами. Ці DSL зазвичай розробляються за допомогою спеціалізованих інструментів, відомих як Language Workbenches. У книзі Фоельтера зазначається, що такі DSL можуть бути інтуїтивними, але їх розробка вимагає значних ресурсів [10]. Проте на практиці вони використовуються рідше,

оскільки їх розробка є доволі складною, вимагає більше ресурсів і часу, а також особливого досвіду.

1.2 Порівняльний аналіз мок-сервісів, що існують

У сучасній розробці програмного забезпечення важливою частиною процесу тестування є створення мок-сервісів, які дозволяють імітувати поведінку реальних систем та зовнішніх API. Для цих цілей існує низка інструментів, кожен з яких має свої особливості, переваги та обмеження. Порівняльний аналіз таких інструментів допоможе вибрати найбільш відповідний варіант для різних сценаріїв і потреб.

У даному аналізі ми розглянемо кілька популярних інструментів для створення мок-сервісів. Зокрема WireMock, MockServer, Hoverfly, Beeceptor та Mocky. Кожен з цих інструментів має свої переваги, що дозволяє використовувати їх для специфічних завдань. Аналіз охоплює простоту використання, функціональність, підтримувані протоколи, інтеграцію, варіанти розгортання, вартість, а також підтримку спільноти. Кожен засіб оцінюється в контексті розробки DSL для генерації мок-сервісів у SOA.

Mocky – це найпростіший інструмент для швидкого створення статичних мок-відповідей. Він дозволяє за лічені секунди сформувати необхідну відповідь у форматі JSON через зрозумілий вебінтерфейс. Завдяки цьому часто використовується для швидкого прототипування та тестування простих API, де не потрібні складні сценарії.

Однією з головних переваг Mocky є його висока простота використання. Зручний вебінтерфейс дозволяє створювати мок-сервіси дуже швидко. Крім того, кожен створений мок отримує унікальне посилання для взаємодії, що робить інтеграцію в тестові процеси зручнішою при роботі з HTTP та HTTPS протоколами. Інструмент надається безплатно, що є плюсом для розробників, які працюють над невеликими проєктами або проводять швидкі демонстрації.

Однак Mocky має і певні недоліки. Його функціональність обмежується тільки статичними відповідями та малим функціоналом, що може стати

бар’єром при тестуванні динамічних сценаріїв чи комплексних систем. Відсутність можливості створення розширених відповідей або інтеграції із більш потужними інструментами тестування обмежує застосування сервісу при масштабній розробці. Таким чином, Mocky є хорошим рішенням для швидкого створення простих мок-відповідей, але може не відповідати потребам складніших проєктів, де потрібна більша гнучкість.

Далі пропоную розглянути WireMock. Він є одним із найпопулярніших інструментів для створення мок-сервісів, особливо в екосистемі Java. Цей інструмент забезпечує широкий набір функцій для імітації поведінки API, що робить його ідеальним для реалізації складних тестових сценаріїв. Його часто використовують як у невеликих проєктах, так і в серйозних корпоративних рішеннях, адже можливості конфігурації практично необмежені. Згідно з даними, WireMock має понад 5 мільйонів завантажень на місяць, що свідчить про його широке використання [11]. Кейс-стаді показує, що команда розробників з Fortune 100 досягла 20% прискорення доставлення завдяки використанню цього інструменту для мокінгу API [12]. Щодо простоти використання, то початківцям може знадобитися певний час на ознайомлення з усіма можливостями. Для зручності існує хмарне рішення WireMock Cloud із вебінтерфейсом. Налаштування через JSON або Java API є зрозумілим.

Однією з основних силових сторін інструменту є гнучкість при зіставленні запитів. WireMock дозволяє проводити складне зіставлення за URL, HTTP-методом, заголовками, файлами кукі, задавати правила для тіла вхідного запиту та іншими параметрами. Завдяки використанню точних збігів, регулярних виразів і JSONPath, можна детально відтворити необхідну поведінку системи. Засіб підтримує проксіювання та запис реальних взаємодій з API, що значно полегшує відтворення складних сценаріїв. Крім того, за допомогою ін’єкції помилок можна імітувати затримки, помилки та збої, що допомагає тестувати стійкість системи [13]. Розширюваність інструменту дозволяє налаштувати майже кожен аспект, що робить його універсальним.

Інтеграція WireMock в модульні тести є досить гнучкою. Його можна використовувати як окремий сервер або запускати через Docker, що спрощує розгортання як локально, так і в хмарних середовищах. Інструмент підтримує різні протоколи: HTTP, HTTPS, gRPC та GraphQL, що робить сумісним із різноманітними технологіями. Щодо вартості, то WireMock доступний безплатно за ліцензією Apache 2.0. Проте також існує платна хмарна версія для тих, хто потребує додаткових можливостей. Активна спільнота користувачів та детальна документація допомагають швидко розв’язувати будь-які питання.

Ще одним інструментом є MockServer, який поєднує можливості мокінгу та проксіювання. Це робить його корисним інструментом для тестування та ізоляції сервісів [14]. Він дозволяє легко симулювати різноманітні поведінкові сценарії, що необхідні в сучасному процесі розробки. Налаштування здійснюється через JSON або клієнтські бібліотеки. Крім того, наявність інтерфейсу допомагає управляти сервісом. Основною функціональною перевагою інструменту є можливість визначення очікувань для запитів і відповідей, що дозволяє точно імітувати потрібну поведінку. Він може пересилати запити до іншого сервера та записувати взаємодії.

MockServer підтримує динамічні відповіді за допомогою JavaScript або Apache Velocity. Інтеграція MockServer досить універсальна. Його можна використовувати як окремий сервер або інтегрувати через бібліотеки для Java, JavaScript та інших мов [15]. Розгортати можна як локально, так і в Docker, що робить використання особливо зручним у різних інфраструктурах.

Вартість інструменту не є перешкодою, оскільки він є безплатним і розповсюджується за ліцензією Apache 2.0. Активна спільнота користувачів і доступна документація на офіційному сайті MockServer сприяють швидкому освоєнню та ефективній підтримці цього рішення.

Beeceptor. Він є зручним вебінструментом, який дозволяє створювати мок-сервіси без необхідності писати код. Завдяки зрозумілому вебінтерфейсу швидко налаштовуються параметри, що значно спрощує процес тестування.

Платформа забезпечує можливість без кодового налаштування, що сприяє створенню штучних API без програмування і значно економить час [16].

Однак існують деякі недоліки, які варто враховувати. Одним з основних недоліків є відсутність локального рішення [14]. Beeceptor працює виключно як хмарний сервіс, що може створити проблеми для компаній або розробників, яким потрібен повний контроль над середовищем тестування або ж в умовах обмеженого інтернет-з'єднання. Відсутність локальної установки може також негативно вплинути на питання безпеки даних та інтеграції в реальні CI/CD-процеси, де важлива автономність системи.

Ще один мінус полягає в обмеженнях безплатного плану. Наприклад, збереження записів запитів доступне лише протягом 10 днів, що може бути недостатньо при масштабних тестуваннях. Деякі розширені можливості платформи знаходяться за платною підпискою, що може обмежити доступ до всіх функціональних можливостей для користувачів, з обмеженим бюджетом.

Також спрощений інтерфейс, який є однією з переваг Beeceptor, може виявитися недостатньо гнучким для високоспеціалізованих сценаріїв тестування. У випадках, коли потрібна детальне налаштування або обробка складних випадків, розробникам може знадобитися потужніший інструмент.

Hoverfly є гнучким рішенням для імітації API, яке дозволяє створювати як прості мок-відповіді, так і складні симуляції реальної поведінки сервісів. Він підходить для тестування різних сценаріїв взаємодії з API. Забезпечує можливості як для статичного мокінгу, так і для динамічних симуляцій. Завдяки відкритості коду та високій продуктивності він знаходить широке застосування як в локальному середовищі, так і у хмарних рішеннях.

Серед основних переваг Hoverfly варто відзначити його універсальність. Інструмент дозволяє налаштовувати реалістичні сценарії взаємодії, симулювати мережеві затримки, відтворювати помилки та навіть управляти станом систем за допомогою комплексних алгоритмів. Також він інтегрується з різними мовами програмування та фреймворками. Висока продуктивність дозволяє обробляти великий обсяг запитів за короткий проміжок часу.

Попри всі свої переваги, Hoverfly має менший функціонал у порівнянні з більш розвиненими рішеннями, такими як WireMock. Зокрема, він підтримує лише протоколи HTTP та HTTPS. Тоді як WireMock працює також із gRPC, GraphQL та має ширші можливості з обробки запитів і відповідей. Це обмежує застосування Hoverfly у складніших архітектурах або сценаріях, де потрібна підтримка кількох типів протоколів та гнучке налаштування логіки мокінгу.

Після проведення аналізу інструментів для імітації сервісів, можемо зробити висновок, що кожен із розглянутих підходів має свої сильні та слабкі сторони. Їх варто враховувати залежно від конкретних вимог проєкту. Зокрема, важливими критеріями при виборі є підтримка протоколів, можливість локального або хмарного розгортання, рівень гнучкості, масштабованість та безпека. Узагальнені результати порівняння інструментів, до уваги яких брався аналіз від MoldStud [17], наведено в таблиці 1.1.

Таблиця 1.1 – Загальне порівняння мок-сервісів

	Mocky	WireMock	MockServer	Beeceptor	Hoverfly
Простота використання	Дуже висока	Середня	Середня	Висока	Середня
Наявність інтерфейсу	Web UI	CLI, Java, Web UI	CLI, Java, UI	Web UI	CLI, Web UI
Поріг входу	Низький	Високий	Помірний	Низький	Помірний
Гнучкість	Низька	Висока	Висока	Середня	Середня
Типи сценаріїв	Дуже прості	Прості та складні	Складні	Прості	Середні

Продовження таблиці 1.1

Ціна	Безплатно	Безплатно/ Платна хмара	Безплатно	Безплатно/ Платно	Безплатно/ Платна хмара
Підтримка	Неактивна	Активна, документація	Активна, документація	Документація	Активна, документація

У таблиці 1.2 представлено порівняння інструментів за низкою ключових технічних характеристик. Зокрема, розглядається підтримка різних протоколів, типи відповідей, які може генерувати інструмент, можливість запису та відтворення реальних запитів, функція проксіювання до реального сервісу, інтеграція з CI/CD-середовищами, підтримувані формати конфігурації, наявність засобів для симуляції мережевих проблем та варіанти розгортання.

Таблиця 1.2 – Порівняння мок-сервісів за технічними характеристиками

	Mocky	WireMock	MockServer	Beeceptor	Hoverfly
Підтримка протоколів	HTTP, HTTPS	HTTP, HTTPS, gRPC, GraphQL	HTTP, HTTPS	HTTP, HTTPS, HTTP/2	HTTP, HTTPS
Типи відповідей	Статичні	Статичні, динамічні, проксі	Статичні, динамічні, проксі	Статичні, динамічні, проксі	Статичні, динамічні
Запис і відтворення	Ні	Так	Так	Ні	Так
Проксіювання	Ні	Так	Так	Так	Так

Продовження таблиці 1.2

Інтеграція з CI/CD	Ні	CLI, REST API	CLI, REST API	REST API	CLI, REST API
Формати конфігурації	Web UI	JSON, Java DSL	JSON, Java DSL	Web UI	YAML, Web UI
Симуляція мережевих проблем	Немає	Затримки, помилки	Затримки, помилки	Затримки, помилки	Затримки, помилки
Розгортання	Хмара	Локально, Docker, хмара, Kubernetes	Локально, Docker	Хмара	Локально, Docker, хмара

Отже, враховуючи всі основні характеристики: гнучкість налаштувань, підтримку різних форматів запитів і відповідей, можливість роботи з умовами, затримками, помилками та інтеграцію з інструментами автоматизації можемо зробити висновки. WireMock є потужним і зручним інструментом для моделювання поведінки HTTP-сервісів. Наявні можливості роблять його одним із найкращих виборів для розробників і тестувальників, які працюють у середовищі розподілених систем або складної взаємодії між сервісами.

1.3 Обґрунтування актуальності та доцільності дослідження

У сучасному програмному забезпеченні SOA та мікросервісні підходи набули широкого поширення завдяки їх гнучкості та масштабованості. Однак ці архітектури ускладнюють тестування через численні залежності між компонентами, які можуть бути розроблені різними командами або навіть організаціями. Мок-сервіси відіграють ключову роль у перевірці якості таких систем. Вони надають змогу імітувати поведінку зовнішніх сервісів, що забезпечує ізольоване тестування, симуляцію різних сценаріїв і паралельну розробку. Попри важливість таких рішень, їх налаштування за допомогою

наявних інструментів, таких як WireMock чи MockServer, може бути складним та вимагати значних зусиль. Це дослідження пропонує розробку DSL для генерації мок-сервісів. Результат має спростити процес, та зробити його доступним для ширшої авдиторії.

Інструменти для створення мок-сервісів мають потужний функціонал і дозволяють симулювати поведінку сервісів та тестувати різні сценарії. Проте налаштування часто вимагає багато часу. Конфігурація через JSON або API може бути складною. Це стає викликом для користувачів [18]. Симулювання динамічних даних потребує складної логіки. Реальні сервіси швидко змінюються, та їх копіювання часто виявляється складним завданням. Інтеграція з CI/CD процесами також вимагає додаткових зусиль. Автоматизація тестів може бути не зовсім ефективною через ці обмеження. Це викликає потребу в простіших рішеннях [19]. Всі ці обмеження підкреслюють необхідність створення засобів, що спрощують налаштування та управління мок-сервісами.

DSL створені для конкретних предметних областей. Вони простіші у використанні, ніж універсальні мови програмування. DSL дозволяють описувати бізнес-логіку та поведінку користувачів. У контексті тестування вони дозволяють створювати тести, які відображають бізнес-логіку та поведінку користувачів, а не технічні деталі [20].

DSL мають чотири головні переваги. По-перше, вони виразні. Мова використовує терміни, близькі до конкретної сфери. По-друге, вони прості у використанні. Користувачам легше брати участь у написанні логіки. По-третє, абстракція дозволяє скоротити час написання та підтримки коду. По-четверте, DSL покращують комунікацію між розробниками та експертами.

У цьому дослідженні буде розроблятися DSL для опису поведінки мок-сервісів у SOA. Користувачі зможуть створювати мок-сценарії через вебінтерфейс. Використовуватиметься Monaco Editor. Він допоможе завдяки підсвічуванню синтаксису та автодоповненню. Код оброблятиметься на Java сервері з допомогою ANTLR. Після парсингу генеруватимуться інструкції для

WireMock, який працює в Docker. Цей підхід автоматизує створення мок-сервісів та робить процес більш ефективним.

Швидкий розвиток програмного забезпечення, особливо в SOA, вимагає ефективних методів тестування. Затримки в тестуванні через залежності від зовнішніх сервісів можуть уповільнити випуск продукту, що є критичним у конкурентному середовищі. Це дослідження відповідає цим потребам. Воно яке пропонує автоматизоване та доступне рішення для створення мок-сервісів. Також сприяє прискоренню розробки та підвищенню якості програмного забезпечення [21]. Це дозволить розробникам зосередитися на вирішенні більш важливих завдань.

DSL розширює аудиторію, яка може брати участь у тестуванні. Бізнес-аналітики та інші нетехнічні користувачі також можуть долучитися до процесу. Це зумовлює ширшу участь команди в розробці.

Використання DSL підвищує ефективність тестування. Інтеграція з CI/CD за допомогою Docker дозволяє швидко розгортати середовища. Автоматизація тестів робить процес більш послідовним. Зрозумілий синтаксис та автоматизований парсинг зменшують кількість помилок. Це буде значити стабільну роботу сервісів та допоможе уникнути людських похибок.

Хоча інструменти, такі як Mockoon, пропонують прості інтерфейси для створення мок-сервісів, вони все ще орієнтовані на розробників і не надають високорівневої абстракції, яку пропонує DSL. Наприклад, Apiary використовує мову Blueprint для опису API, яка може застосовуватися для мокінгу. Проте вона не є настільки гнучкою для складних сценаріїв. Запропоноване рішення заповнить цю прогалину, і надасть спосіб опису поведінки мок-сервісів, що інтегрується з потужним інструментом WireMock.

Дослідження є актуальним. Воно розв’язує проблему складного створення мок-сервісів. Пропонується інноваційний підхід з використанням DSL. Цей підхід відповідає сучасним вимогам швидкого розвитку ПЗ. Таким чином, розробка DSL має великий потенціал для покращення процесів тестування.

2 ОГЛЯД ПРОЕКТУВАННЯ ТА ПРОГРАМНОЇ РЕАЛІЗАЦІЇ

Другий розділ пояснювальної записки є основною частиною дослідження. У цьому розділі буде виконано детальний аналіз поставленої задачі. Окреслимо основні та загальні особливості проблеми. Оглянемо цілі та вимоги до створення DSL. Далі задача буде декомпозована на простіші складники для полегшення подальшої роботи.

В процесі визначатиметься послідовність дій дослідження. Опишемо, як застосовуватимуться алгоритми та інші технічні підходи для розв'язання проблеми. На основі цього буде сформовано структуроване рішення.

2.1 Розробка проблемно-орієнтованої мови

У цьому підрозділі буде розглянуто розробку проблемно-орієнтованої мови. Спочатку визначимо цілі, вимоги до мови та її область застосування. Це для того, аби чітко окреслити завдання та очікувані результати. Далі буде проведена формалізація граматики, зокрема за нотацією Бекуса-Наура, та встановлено правила опису тестових кінцевих точок що забезпечить структурованість та однозначність синтаксису. Потім, буде описано основні конструкції мови, які становитимуть основу для опису поведінки сервісів.

2.1.1 Визначення цілей, вимог до мови та область застосування

Цей пункт визначає цілі розробки DSL, вимоги до мови та її область застосування, враховуючи структуру системи.

Розробка DSL має на меті розв'язання проблеми складного створення мок-сервісів. А саме мова створюється для того, щоб спростити конфігурацію та опис їх поведінки. Вона спрямована на підвищення ефективності. Автоматизація генерації мок-сценаріїв має зменшувати час на налаштування тестових середовищ. Розробка матиме на меті стандартизувати спосіб визначення мок-сервісів, що створюватиме єдиний підхід у командах. Вона

також має підтримувати складні сценарії. DSL слід визначати характеристику вхідного запиту, відповіді, затримки, вебхуки та симулювати помилки.

Для досягнення поставлених цілей DSL повинна відповідати низці функціональних і нефункціональних вимог. Вони мають враховувати потреби користувачів, базуватися на типових сценаріях використання мок-сервісів та можливостей WireMock.

DSL має бути інтуїтивно зрозумілою для користувача. Її синтаксис повинен бути простим, наближеним до природної мови або звичних програмних конструкцій, щоб забезпечити легкість у сприйнятті для користувачів.

Повинна бути можливість визначати кінцеві точки з чітким зазначенням HTTP-методів і маршрутів для точного опису поведінки сервісу у відповідь на конкретні запити. Очікується підтримка умов відповідності. Користувач повинен мати змогу задавати правила перевірки URL, заголовків, параметрів, файлів кукі та вмісту тіла запиту з використанням різних типів перевірок. Вони можуть бути такі, як повна відповідність, регулярні вирази чи логічні перевірки структури. Необхідна й підтримка умовної логіки для створення складних сценаріїв, залежно від вхідних параметрів. Можливість комбінувати умови за допомогою логічних операцій підвищить гнучкість опису поведінки.

Система має забезпечувати опис відповідей на запити, включаючи як стандартні відповіді, так і варіанти для симуляції помилок чи проксіювання до реального сервісу. Щоб імітувати справжні умови роботи сервісу, потрібно передбачити підтримку затримок. Це дасть змогу моделювати повільні відповіді, затримки мережі або варіативність у часі обробки.

Також має бути можливість додавати вебхуки. Це дії, що виконуються після запиту, наприклад, для сповіщення чи взаємодії з іншими системами. Кінцевий інструмент з використанням DSL повинен мати вебінтерфейс, забезпечувати зручну валідацію, автодоповнення, підсвічування синтаксису та інші механізми підтримки розробника в інтерактивному середовищі.

Далі буде розглянуто нетехнічні аспекти у вимогах до розроблюваної DSL. Всі характеристики важливі для інструменту. Вони описані з урахуванням не лише функціональності, а й зручності впровадження в командну роботу чи освітній проєкт. Їх буде наведено у таблиці 2.1.

Таблиця 2.1 – Перелік нефункціональних вимог до кінцевої DSL

Вимога	Опис
Простота освоєння	DSL має бути доступною для користувачів із базовими знаннями програмування через зрозумілий синтаксис і документацію.
Розширюваність	Мова повинна дозволяти додавання нових функцій, таких як підтримка інших протоколів або інструментів, без значних змін у граматиці.
Продуктивність	Парсинг і генерація конфігурацій мають бути швидкими, щоб не затримувати процес тестування.
Документація	Необхідна документація з прикладами, що пояснюють синтаксис і типові сценарії використання.
Сумісність	DSL має бути сумісною з сучасними інструментами розробки, через інтеграцію з Docker.

DSL призначена для використання в SOA системах, де мок-сервіси відіграють критичну роль у розробці та тестуванні. Вона дозволяє моделювати різні сценарії тестування. Це модульне тестування для ізоляції компонентів, інтеграційне для перевірки взаємодії між компонентами. Дослідження "To Mock or Not to Mock? An Empirical Study on Mocking Practices" показує, що мок-об’єкти широко використовуються в проєктах з відкритим кодом [22]. У

ньому було проаналізовано понад 2000 тестових залежностей у трьох таких розробках та одній промисловій системі. Опитування понад 100 професіоналів виявило, що розробники часто використовують мокінг для залежностей, які ускладнюють тестування, та навіть імітують класи, що інкапсулюють бізнес-логіку.

Globall App Testing повідомляє, що ринок тестування невпинно зростає, та має сукупний середньорічний темп росту на рівні 5% з 2023 по 2027 роки. Теж заявлено, що 30% розробників вважають ручне тестування найбільш часовитратним аспектом [23]. Це підкреслює важливість автоматизованих методів, таких як мокінг, для підвищення ефективності. Наприклад, мокінг дозволяє швидше проводити unit testing, зменшуючи залежність від зовнішніх систем. Наприклад, аналіз показує, що зростання ринку тестування до $15,93 млрд у 2023 році свідчить про те, що інвестиції в тестування, включаючи мокінг, є пріоритетом для компаній [23].

За допомогою DSL можлива паралельна розробка, коли команди можуть працювати незалежно і не чекати готовності зовнішніх сервісів. Також ця мова полегшує прототипування. Доказом є можливість швидко створювати мок-сервіси для демонстрації функціональності або перевірки концепцій.

Формальний синтаксис є важливим для створення мови, яка описує мок-сервіси, включно з їхніми інтерфейсами та поведінкою. Відповідно до досліджень Гамзаєва Р.О. та Ткачука М.В. [24], створення проблемно-орієнтованої мови моделювання вимагає чіткого визначення такого синтаксису та граматики, що є ключовим для точного зображення структури та поведінки мок-сервісів у системах SOA.

DSL знаходить застосування не лише в технічних процесах, а й у навчанні та освіті. Вона допомагає вивчати принципи роботи API та стратегії тестування. Цільова аудиторія включає розробників, які створюють тестові середовища. Також тестувальників, бізнес-аналітиків та технічних лідерів, що стандартизують процеси тестування в командах.

Визначені цілі відповідають сучасним вимогам розвитку програмного забезпечення. Відповідність ним зробить тестування швидшим, простішим і надійнішим.

2.1.2 Формалізація граматики та правил опису тестових кінцевих точок

У цьому пункті пояснюється спосіб формалізації граматики DSL. Також буде наданий опис структури граматики та її ключові компоненти.

Відомим генератором парсерів для формальних мов є ANTLR. Він приймає граматику мови у спеціальному форматі та автоматично створює код для лексичного і синтаксичного аналізу. Інструмент використовується для побудови компіляторів, інтерпретаторів, трансляторів та аналізаторів вхідного тексту. Завдяки йому розробник отримує готову структуру для подальшої обробки синтаксичних конструкцій.

Засіб підтримує декілька мов програмування, включно з Java, JavaScript, TypeScript та іншими. Він генерує дерева розбору та надає API для їх обробки. Це полегшує реалізацію складних мовних конструкцій та їх трансформацію. ANTLR часто застосовується в академічних, дослідницьких і промислових проєктах, де потрібно працювати з формальними мовами або складною структурою даних.

У цьому дослідженні ми будемо використовувати цей інструмент для побудови парсера на основі визначеної нами граматики. Його застосування дає можливість створити точне й однозначне формальне представлення тестових кінцевих точок.

При розробці DSL важливо мати чітко визначені лексичні правила, які розбивають текстовий потік на окремі токени. Ці правила є основою для подальшого синтаксичного аналізу та побудови логічної структури коду. Завдяки їм кожен елемент: рядок, число чи коментар матиме однозначне визначення, що є ключовим для стабільності та точності роботи системи.

Спочатку буде описано основні лексичні складники DSL. Це такі, як токени для рядків, чисел, булевих значень та інші, які формують базові елементи синтаксису.

Токен "STRING" призначений для розпізнавання звичайних літералів, які обгорнені подвійними лапками. Усередині такого рядка допустимі як прості символи, так і спеціальні, екрановані за допомогою символу оберненої скісної риски. Цей механізм дозволяє безпечно включати будь-який вміст у рядок і не порушувати структуру коду.

Токен "TRIPLE_STRING" розширює можливості запису тексту, дозволяючи використовувати потрійні лапки для визначення рядків. Такий формат корисний для багаторядкових текстових блоків або даних з великою кількістю лапок. Це усуває необхідність постійного екранування символів. Завдяки цьому можна легко писати великий зміст і зберігати його цілісність.

Токен "NUMBER" служить для розпізнавання числових значень. Він враховує можливу наявність знака мінус, а також дозволяє вводити як цілі числа, так і числа з десятковою частиною. Така гнучкість допомагає точно обробляти числові параметри у DSL.

Для логічних операцій застосовується токен "BOOLEAN", який чітко визначає допустимі значення true та false. Вони можливі як у нижньому, так і у верхньому регістрі.

Коментарі в коді розділяються двома категоріями. "LINE_COMMENT" відповідає за однорядкові коментарі, які починаються з двох символів скісної риски та продовжуються до кінця рядка. Це зроблено для того, аби швидко відключати певні фрагменти коду чи робити короткі пояснення. "BLOCK_COMMENT" дозволяє розміщувати багаторядкові коментарі, оскільки він охоплює весь текст між символами /* та */. Обидва типи коментарів позначені для пропуску. Тобто вони ігноруються під час аналізу коду, що дозволяє зосередитися на важливих елементах.

Токен "WS" обробляє пробільні символи, включно з пробілами, табуляцією та символами нового рядка. Вони важливі для форматування коду,

але не впливають на логіку, тому також пропускаються при побудові синтаксичного дерева.

Таким чином, всі ці лексичні правила для токенів разом створюють основу для точного та безпомилкового розпізнавання елементів DSL. Кожне правило виконує свою функцію і допомагає розбити текст на зрозумілі блоки. Вони потім перетворюються в логічну структуру за допомогою синтаксичного аналізу. У таблиці 2.2 представлені ключові лексичні компоненти для розроблюваної DSL.

Таблиця 2.2 – Ключові лексичні компоненти DSL

Назва лексичного токена	Значення виразу для лексичного токена	Опис токена
STRING	'"' ('\\' ["\\] | ~["])* '"'	Рядок у лапках з підтримкою екранування послідовностей
TRIPLE_STRING	'"""' ('\\"""' | .)*? '"""'	Рядок у потрійних лапках, що може містити будь-який вміст
NUMBER	'-'? [0-9]+ ('.' [0-9]+)?	Ціле або десяткове число
BOOLEAN	'true' | 'false' | 'TRUE' | 'FALSE'	Логічні значення (булеві константи)
LINE_COMMENT	'//' ~[\r\n]* -> skip	Коментар у рядку, що ігнорується
BLOCK_COMMENT	'/*' .*? '*/' -> skip	Блоковий коментар, що ігнорується
WS	[\t\r\n]+ -> skip	Пробіли, табуляції та перенесення рядків, що ігноруються

Далі подано опис правил для опису тестових кінцевих точок. Він демонструє, як має виглядати граматика DSL для генерації мок-сервісів.

У розробці цієї спеціалізованої мови кожна тестова кінцева точка описується як єдиний блок, що починається з ключової конструкції "DEFINE ENDPOINT". Після цього йде блок у фігурних дужках, який містить опис кінцевої точки. Загальна схема структури визначення кінцевої точки зображена на рисунку 2.1.

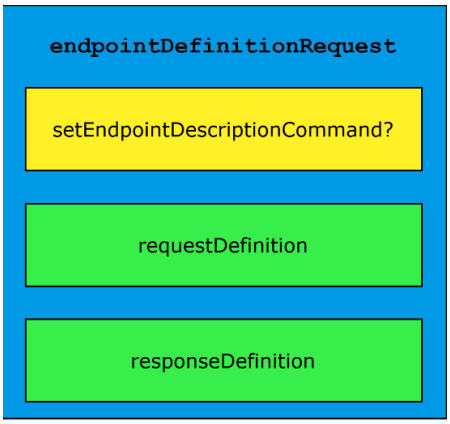

Рисунок 2.1 – Схема структури визначення кінцевої точки

Опціонально можна визначити команду для визначення опису через setEndpointDescriptionCommand. Далі обов’язково слідує блок опису запиту і блок опису відповіді.

Запит описується всередині блоку, який починається з "DEFINE REQUEST". Він містить ряд обов’язкових команд, такі як setHttpMethodCommand, setUrlMatchTypeCommand та setPathCommand. Також є можливість задавати додаткові налаштування. А саме: визначення пріоритету, параметрів запиту, заголовків, файлів кукі, форм-параметрів і різноманітних правил для тіла вхідного запиту. Кожна із цих конструкцій доповнює загальну картину і надає точніше налаштування запиту з більшим функціоналом.

Відповідь формується всередині блоку "DEFINE RESPONSE". У цьому блоці можна вказати один із трьох варіантів відповіді: пряму, зі спеціальним

типом помилки або проксі. Для прямої відповіді використовується блок "ADD DIRECT RESPONSE" з командою встановлення статус-коду, типу тіла відповіді та відповідного вмісту. Якщо потрібно змоделювати помилкову ситуацію, то використовується "ADD FAULT RESPONSE". Разом з цим встановлюється тип помилки через setFaultTypeCommand. В разі проксі-відповіді опис формується через "ADD PROXY RESPONSE". Всередині є команда для задання URL-адреси віддаленого сервісу і додатковими параметрами для переписування хосту та використання шаблонів.

Окремі блоки у граматиці розширюють можливості опису поведінки сервісу для відповіді. Секція "DELAY" дозволяє задавати різні типи затримок. Це фіксована, випадкова та інші. Аналогічно, блок "WEB_HOOKS" створює можливості для налаштування вебхуків. "HEADERS" всередині запиту або відповіді, відображають додавання набору заголовків.

Крім того, граматика передбачає строго формалізоване описання умов для перевірки запиту. Команди для них записуються за допомогою спеціальних конструкцій. Є прості та складені умови. Усі вони мають однаковий формат, де вказується тип умови, її значення. За необхідності є можливість заперечення через ключове слово "NOT".

Дана граматика чітко структурує опис тестових кінцевих точок. Всі правила є формалізованими та послідовними. Вони дозволяють розбити опис на структурні блоки, які відповідають за певну частину визначення тестової кінцевої точки. На рисунку 2.2 представлена граматика в нотації Бекуса-Наура для формального опису розробленої DSL.

```
<endpointDefinitionRequest> ::= "DEFINE" "ENDPOINT" "{"
   <optionalEndpointDescription>
   <requestDefinition>
   <responseDefinition>
"}"

<optionalEndpointDescription> ::= <setEndpointDescriptionCommand> | ""

<requestDefinition> ::= "DEFINE" "REQUEST" "{"
   <setHttpMethodCommand>
   <setUrlMatchTypeCommand>
   <setPathCommand>
   <optionalPriorityCommand>
   <optionalRequestQueryParams>
   <optionalRequestHeaders>
   <optionalRequestCookies>
   <optionalRequestFormParams>
   <optionalRequestBodyRules>
"}"

<optionalPriorityCommand> ::= <setPriorityCommand> | ""
<optionalRequestQueryParams> ::= <requestQueryParams> | ""
<optionalRequestHeaders> ::= <requestHeaders> | ""
<optionalRequestCookies> ::= <requestCookies> | ""
<optionalRequestFormParams> ::= <requestFormParams> | ""
<optionalRequestBodyRules> ::= <requestBodyRules> | ""

<responseDefinition> ::= "DEFINE" "RESPONSE" "{"
   (<directResponseDefinition> | <faultResponseDefinition> | <proxyResponseDefinition>)
   <optionalResponseDelay>
   <optionalResponseWebHooks>
"}"

<optionalResponseDelay> ::= <responseDelay> | ""
<optionalResponseWebHooks> ::= <responseWebHooks> | ""

<directResponseDefinition> ::= "ADD" "DIRECT" "RESPONSE" "{"
   <setResponseStatusCodeCommand>
   <optionalDynamicResponseTemplating>
   <optionalResponseHeaders>
   <setResponseBodyTypeCommand>
   <setValueCommand>
"}"

<optionalDynamicResponseTemplating> ::= <setDynamicResponseTemplatingCommand> | ""
<optionalResponseHeaders> ::= <responseHeaders> | ""

<faultResponseDefinition> ::= "ADD" "FAULT" "RESPONSE" "{"
   <setFaultTypeCommand>
"}"

<faultTypes> ::=
    "NO_FAULT"
  | "CLOSE_SOCKET_WITH_NO_RESPONSE"
  | "SEND_BAD_HTTP_DATA_THEN_CLOSE_SOCKET"
  | "SEND_200_RESPONSE_THEN_BAD_HTTP_DATA_THEN_CLOSE_SOCKET"
  | "PEER_CONNECTION_RESET"

<proxyResponseDefinition> ::= "ADD" "PROXY" "RESPONSE" "{"
   <setUrlCommand>
   <optionalProxyResponseHeaders>
   <optionalTemplatingCommand>
   <optionalHostnameRewritingCommand>
"}"

<optionalProxyResponseHeaders> ::= <responseHeaders> | ""
<optionalTemplatingCommand> ::= <setTemplatingCommand> | ""
<optionalHostnameRewritingCommand> ::= <setHostnameRewritingCommand> | ""

<responseDelay> ::= "DELAY" "{" (<noDelay> | ("SET" "TIME_UNIT" "TO" <timeUnit> ... )) "}"
```

Рисунок 2.2 – Формалізований опис DSL в нотації Бекуса-Наура

2.1.3 Опис основних конструкцій мови

У цьому пункті ми спочатку ознайомимося з базовими поняттями та ключовими термінами нашої DSL. Далі, крок за кроком, заглибимося в кожну з основних конструкцій. Від оголошення кінцевої точки до опису запиту, відповіді та їх складників. Продемонструємо деталі синтаксису й приклади використання. Такий поступовий перехід від узагальнень до конкретики допоможе спершу побудувати міцну основу розуміння, а потім розкрити всі нюанси окремих блоків та команд.

Визначення кінцевої точки – це ключовий елемент, який описується за допомогою конструкції "DEFINE ENDPOINT". Саме вона об’єднує всередині себе як запит, так і відповідь. Тобто вона є оболонкою, що чітко формалізує, як саме має клієнт взаємодіяти з нашою розробкою. Крім основних компонентів, визначення кінцевої точки може доповнюватися опціональним описом. Для цього використовується команда "SET DESCRIPTION TO". Вона дозволяє додати пояснення, яке уточнює призначення або особливості цієї кінцевої точки. Загальна структура була визначена раніше і вказана на рисунку 2.1. Фактичний вигляд конструкції у вигляді DSL наведено на рисунку 2.3.

```
DEFINE ENDPOINT {
    SET DESCRIPTION TO "Отримання списку користувачів"
    DEFINE REQUEST {
        SET METHOD TO GET
        SET URL_MATCH_TYPE TO PATH
        SET REQUEST_PATH TO "/api/users"
    }
    DEFINE RESPONSE {
        ADD DIRECT RESPONSE {
            SET STATUS_CODE TO 200
            SET BODY_TYPE TO JSON
            SET VALUE TO "{\"users\": []}"
        }
    }
}
```

Рисунок 2.3 – Приклад не складного визначення кінцевої точки розроблюваною DSL

Цей фрагмент оголошує кінцеву точку для отримання списку користувачів через GET-запит за шляхом "/api/users". У відповідь сервер повертає статус 200 і JSON-об'єкт із порожнім списком.

Конструкція "DEFINE REQUEST" дозволяє задати, який саме HTTP-запит повинен викликати відповідь кінцевої точки. Спочатку визначається метод запиту. Наприклад, GET, POST, PUT, DELETE, PATCH, OPTIONS, TRACE, ANY. Потім задається шлях відповідно до вказаного типу. Він може бути простим, включати параметри або навіть використовувати регулярні вирази. Задається такими типами: PATH, PATH_AND_QUERY, ANY_URL, PATH_AND_QUERY_REGEX, PATH_TEMPLATE, PATH_REGEX.

Додатково можна встановити пріоритет, щоб у випадку кількох збігів знати, який запит обробляти першим. Також є можливість задати точні умови щодо заголовків, параметрів запиту чи форм, файлів кукі, а також змісту тіла запиту. Наприклад, перевірити, чи відповідає він певному JSON. Усе це забезпечує гнучке і точне зіставлення вхідних запитів. Приклад визначення правил для вхідного запиту у вигляді DSL наведено на рисунку 2.4.

```
DEFINE REQUEST {
    SET METHOD TO GET
    SET URL_MATCH_TYPE TO PATH_AND_QUERY
    SET REQUEST_PATH TO "/api/users"
    SET PRIORITY TO 1
    QUERY_PARAMS {
        ADD QUERY_PARAM_RULE {
            SET NAME TO "page"
            ADD CONDITION {
                SET REQUEST_CONDITION_TYPE TO EQUALS
                SET VALUE TO "1"
            }
        }
    }
    HEADERS {
        ADD HEADER_RULE {
            SET NAME TO "Authorization"
            ADD CONDITION {
                SET REQUEST_CONDITION_TYPE TO MATCHES_REGEX
                SET VALUE TO "Bearer .*"
            }
        }
    }
}
```

Рисунок 2.4 – Приклад визначення вхідного запиту із заданими параметрами у формі DSL

Складовими частинами визначення запиту є конструкції "QUERY_PARAMS", "FORM_PARAMS", "HEADERS", "COOKIES". Усі вони є опціональними. Кожна з них відповідає за окремий тип вхідних даних. Всередині вони містять спеціальні блоки, які описують умови для конкретних полів. Ці блоки складаються з одного або кількох правил. Кожне з них вказує назву поля і визначає, яким критеріям воно має відповідати. Це дуже гнучко описує, за яких умов запит має активувати відповідь кінцевої точки. Приклад використання цих конструкцій наведено на рисунку 2.5.

```
DEFINE ENDPOINT {
  SET DESCRIPTION TO "Login API endpoint"
  DEFINE REQUEST {
    SET METHOD TO POST
    SET URL_MATCH_TYPE TO PATH
    SET REQUEST_PATH TO "/api/login"
    QUERY_PARAMS {
      ADD QUERY_PARAM_RULE {
        SET NAME TO "version"
        ADD CONDITION {
          SET REQUEST_CONDITION_TYPE TO EQUALS
          SET VALUE TO "1.0"
        }
      }
    }
    HEADERS {
      ADD HEADER_RULE {
        SET NAME TO "Content-Type"
        ADD CONDITION {
          SET REQUEST_CONDITION_TYPE TO EQUALS
          SET VALUE TO "application/json"
        }
      }
    }
    COOKIES {
      ADD COOKIE_RULE {
        SET NAME TO "session_id"
        ADD CONDITION {
          SET REQUEST_CONDITION_TYPE TO MATCHES_REGEX
          SET VALUE TO "[0-9a-f]{32}"
        }
      }
    }
    FORM_PARAMS {
      ADD FORM_PARAM_RULE {
        SET NAME TO "username"
        ADD CONDITION {
          SET REQUEST_CONDITION_TYPE TO IS_PRESENT
        }
      }
    }
  }
  DEFINE RESPONSE {
    ADD DIRECT RESPONSE {
      SET STATUS_CODE TO 200
      SET BODY_TYPE TO JSON
      SET VALUE TO "{\"message\": \"Login successful\"}"
    }
  }
}
```

Рисунок 2.5 – Приклад визначення параметрів та умов для них у вхідному запиті через DSL

Важливою частиною в контексті опису запиту є умови. Вони використовуються для того, аби вказати, яким саме критеріям мають відповідати вхідні дані. Можуть бути простими або складеними. Проста умова є перевіркою одного значення. Найпростіша з них – EQUALS, яка перевіряє, чи точно збігається значення. Далі йде CONTAINS, що дізнається, чи містить значення підрядок. MATCHES_REGEX застосовується для регулярних виразів, а такі різновиди типів, як MATCHES_JSON_PATH або MATCHES_XPATH для перевірки за специфічним шляхом у структурі JSON або XML відповідно.

Є також умови для перевірки складніших форматів. Наприклад, MATCHES_JSON_SCHEMA перевіряє, чи відповідає вхідний JSON заданій схемі. EQUALS_JSON або EQUALS_XML порівнюють відповідні тіла запитів на ідентичність. Для дат можна використати EQUALS_DATE_TIME, BEFORE або AFTER, щоб перевірити часові межі чи точні збіги.

Складені умови об’єднують декілька простих. Тут можна використовувати логічні оператори, які задаються таким типом. Найпоширенішими є AND та OR. Також є VALUES_INCLUDE, який перевіряє, чи значення з множини умов включені у вхідні дані, і VALUES_ARE_EXACTLY, що вимагає точного збігу множини значень.

Крім того, будь-яку просту або складену умову можна заперечити та вказати, що вона не має виконуватися. Для цього використовуються конструкції NOT, які обгортають умову в круглі дужки. У такий спосіб можна побудувати цілісну систему перевірок від найпростішої до комплексної логіки, яка враховує кілька факторів одночасно. Приклад фрагментів простої та складеної умов наведено на рисунку 2.6.

```
QUERY_PARAMS {
   ADD QUERY_PARAM_RULE {
      SET NAME TO "filter"
      ADD CONDITION {
         SET REQUEST_CONDITION_TYPE TO MATCHES_REGEX
         SET VALUE TO "CAT-[0-9]+"
      }
   }
}


QUERY_PARAMS {
   ADD QUERY_PARAM_RULE {
      SET NAME TO "filter"
      ADD CONDITION {
         SET REQUEST_CONDITION_TYPE TO OR
         ADD CONDITION {
            SET REQUEST_CONDITION_TYPE TO EQUALS
            SET VALUE TO "active"
         }
         ADD CONDITION {
            NOT (
               SET REQUEST_CONDITION_TYPE TO EQUALS
               SET VALUE TO "inactive"
            )
         }
      }
   }
}
```

Рисунок 2.6 – Приклад фрагментів простої та складеної умов написаних DSL

Конструкція "REQUEST_BODY_RULES" дозволяє задавати умови для перевірки тіла HTTP-запиту у визначенні імітованої кінцевої точки. Цей розділ включається до requestDefinition. Він використовується, коли потрібно перевірити вміст тіла запиту перед активацією відповіді кінцевої точки. Ця конструкція містить одну або кілька умов для оцінки тіла запиту. Кожна створюється через "ADD REQUEST_BODY_RULE" і включає одну перевірку. Вона може бути прямою або запереченою. Пряма пропонує різноманітні способи валідації, тоді як заперечна форма дозволяє інвертувати логіку, через конструкцію NOT. Ці умови відрізняються від попередньо описаних.

Прямі перевірки тіла запиту поділяються на кілька категорій. Кожна з них пропонує свій спосіб валідації. Базова перевірка значення дає змогу порівняти тіло запиту з очікуваним текстом за чотирма критеріями. Вона дозволяє перевірити точну відповідність заданому значенню, відповідність регулярному виразу, наявність підрядка або відповідність JSON-схемі.

Спочатку визначається тип перевірки, а потім вказується значення для порівняння.

Найпростіший спосіб валідації перевіряє лише наявність тіла запиту. Він не звертає уваги на вміст. Це зручно, коли потрібно переконатися, що запит не порожній, і конкретні дані не очікуються.

Наступний спосіб валідації зосереджений на структурі тіла запиту, як XML. Він порівнює вміст з очікуваним документом цього формату. Порівняння відбувається посимвольно, та можна активувати XMLUnit. Це дозволяє ігнорувати дрібні відмінності, наприклад, у форматуванні.

Перевірка структури JSON звіряє тіло запиту з очікуваним об’єктом цього формату. Є можливість налаштувати перевірку, щоб не зважати на порядок у масивах. Також можна ігнорувати надлишкові елементи, яких немає в очікуваному значенні.

Перевірка за JSONPath отримує частину тіла запиту через відповідний вираз для подальшої оцінки. Спочатку задається вираз, а потім додаткова умова, яка визначає критерій для значення. Вона може перевіряти точну відповідність, регулярний вираз, підрядок чи порівняння дат. Наприклад, чи значення передує заданій даті. Також може перевіряти лише наявність витягнутого значення. Цю підумову можна заперечити, щоб вказати, що значення не має відповідати критерію.

Перевірка за XPath працює подібно, але для XML. Вона знаходить частину тіла через відповідний вираз. Отримане значення оцінюється за тими ж додатковими умовами, що й у JSONPath. Це точна відповідність, регулярний вираз, підрядок, порівняння дат або наявність. Підумову також можна заперечити, якщо потрібно інвертувати логіку перевірки.

Приклад використання конструкції наведено на рисунку 2.7.

```
REQUEST_BODY_RULES {

 ADD REQUEST_BODY_RULE {
  SET REQUEST_BODY_CONDITION_TYPE TO EQUALS
  SET VALUE TO "submit"
 }

 ADD REQUEST_BODY_RULE {
  SET REQUEST_BODY_CONDITION_TYPE TO EQUALS_JSON
  SET VALUE TO "{\"type\": \"submit\", \"id\": 123}"
  SET IGNORE_ARRAY_ORDER TO TRUE
  SET IGNORE_EXTRA_ELEMENTS TO TRUE
 }

 ADD REQUEST_BODY_RULE {
  SET REQUEST_BODY_CONDITION_TYPE TO IS_PRESENT
 }

 ADD REQUEST_BODY_RULE {
  SET REQUEST_BODY_CONDITION_TYPE TO MATCHES_JSON_PATH
  SET NAME TO "$.type"
  SET CONDITION_TYPE TO EQUALS
  SET VALUE TO "submit"
 }

 ADD REQUEST_BODY_RULE {
  SET REQUEST_BODY_CONDITION_TYPE TO MATCHES_X_PATH
  SET NAME TO "//action"
  NOT (
   SET CONDITION_TYPE TO EQUALS
   SET VALUE TO "delete"
  )
 }

 ADD REQUEST_BODY_RULE {
  NOT (
   SET REQUEST_BODY_CONDITION_TYPE TO CONTAINS
   SET VALUE TO "error"
  )
 }
```

Рисунок 2.7 – Приклад фрагментів умов для тіла вхідного запиту виражених через DSL

Попередній приклад містить шість умов валідації тіла запиту. Вони перевіряють його вміст різними способами. Перша вимагає точної відповідності рядка "submit". Друга звіряє тіло з об’єктом JSON, де тип має бути "submit" і ідентифікатор дорівнювати 123, ігноруючи порядок масивів та зайві поля. Третя просто перевіряє наявність тіла без уваги до вмісту. Четверта за виразом JSONPath отримує значення типу і вимагає, щоб воно дорівнювало "submit". П’ята за XPath перевіряє, щоб значення дії не було "delete". А шоста забороняє тілу містити підрядок "error".

Відповідь на запит визначається через блок responseDefinition, який може бути прямим, з помилкою або проксі. Кожен тип відповіді має свої

особливості та застосування, а також додаткові опції. Є можливість додавати затримки та вебхуки, для створення реалістичних сценаріїв тестування.

Пряма відповідь визначається у блоці "ADD DIRECT RESPONSE" і дозволяє задати власну відповідь. В ній можна вказати статус-код, тип тіла та вміст. Також дозволено додавати заголовки. Можна увімкнути динамічне шаблонування для гнучкості. Статус-код задається через зрозумілу команду "SET STATUS_CODE TO". Тип тіла визначається через "SET BODY_TYPE TO". Підтримуються формати: JSON, XML, HTML, TEXT та BASE64. Вміст відповіді задається командою "SET VALUE TO". Заголовки додаються опціонально через "HEADERS". Якщо потрібно, можна активувати шаблонування через setDynamicResponseTemplatingCommand. Це дозволяє створювати динамічний вміст у відповіді. Приклад цієї конструкції наведено на рисунку 2.8.

```
DEFINE RESPONSE {
   ADD DIRECT RESPONSE {

      SET STATUS_CODE TO 200
      SET DYNAMIC_RESPONSE_TEMPLATING TO TRUE
         HEADERS {
            ADD HEADER {
               SET NAME TO "headerName"
               SET VALUE TO "headerValue"
            }
            ADD HEADER {
               SET NAME TO "headerName"
               SET VALUE TO "headerValue"
            }
         }

         SET BODY_TYPE TO TEXT
         SET VALUE TO "response body query"
   }
}
```

Рисунок 2.8 – Приклад визначення прямої відповіді на запит до кінцевої точки через DSL

Відповідь з помилкою визначається через "ADD FAULT RESPONSE". Використовується для імітації різних типів помилок. Можна змоделювати ситуації, коли з’єднання закривається без відповіді, або коли надсилаються

некоректні дані. Тип помилки задається через setFaultTypeCommand. Список можливих значень включає кілька варіантів. Вони та пояснення до них містяться у таблиці 2.3. Приклад повної конструкції наведено на рисунку 2.9.

Таблиця 2.3 – Значення можливих типів помилок для відповіді

Тип помилки	Опис
NO_FAULT	Відсутність помилки.
CLOSE_SOCKET_WITH_NO_RESPONSE	Закриває з’єднання без відповіді.
SEND_BAD_HTTP_DATA_THEN_CLOSE_SOCKET	Надсилає неправильні дані, а потім закриває сокет.
PEER_CONNECTION_RESET	Імітує ситуацію, коли з’єднання скидається на рівні мережі.
SEND_200_RESPONSE_THEN_BAD_HTTP_DATA_THEN_CLOSE_SOCKET	Спочатку надсилає валідну відповідь, а потім некоректні дані.

```
DEFINE RESPONSE {
  ADD FAULT RESPONSE {
    SET FAULT_TYPE TO SEND_BAD_HTTP_DATA_THEN_CLOSE_SOCKET
  }
  DELAY {
    SET TIME_UNIT TO MS
    SET DELAY_TYPE TO FIXED
    SET VALUE TO 1000
  }
}
```

Рисунок 2.9 – Приклад визначення відповіді з помилкою на запит до кінцевої точки через DSL

Проксі-відповідь визначається у блоці "ADD PROXY RESPONSE" і дозволяє перенаправити запит на інший URL. У цьому типі відповіді можна задати цільову адресу за допомогою команди setUrlCommand. Також є

можливість додати заголовки через responseHeaders. Якщо потрібно змінювати вміст під час перенаправлення, можна увімкнути шаблонування за допомогою setTemplatingCommand. Крім того, доступна опція переписування імені хоста через setHostnameRewritingCommand. Приклад цієї конструкції наведено на рисунку 2.10.

```
DEFINE RESPONSE {
    ADD PROXY RESPONSE {
        SET URL TO "https://wc-echo.wiremockapi.cloud/webhook/echo"

        HEADERS {
            ADD HEADER {
                SET NAME TO "X-My-Header"
                SET VALUE TO "headerValue"
            }
            ADD HEADER {
                SET NAME TO "X-My-Another-Header"
                SET VALUE TO "headerValue"
            }
        }
        SET TEMPLATING TO TRUE
        SET HOSTNAME_REWRITING TO FALSE
    }
}
```

Рисунок 2.10 – Приклад визначення проксі відповіді на запит до кінцевої точки через DSL

Затримки для відповіді визначаються в блоці "DELAY" і дозволяють симулювати час обробки чи мережеві затримки. Якщо затримка не потрібна, використовується тип, що відповідає значенню NO_DELAY. Для фіксованої затримки використовується FIXED, де вказується конкретне значення й одиниці часу. Якщо потрібно змоделювати випадкову затримку, можна застосувати RANDOM_UNIFORM. Вона задається через нижню та верхню межі. RANDOM_LOG_NORMAL використовується разом з медіаною та стандартним відхиленням. Для симуляції поступового надсилання відповіді використовується спосіб CHUNKED_DRIBBLE. В ньому зазначається кількість частин та загальна затримка. Всі типи підтримують налаштування одиниць часу через timeUnits, які можуть бути мілісекундами, секундами, хвилинами або годинами.

Вебхуки розміщені у блоці "WEB_HOOKS" і дозволяють при зверненні до кінцевої точки автоматично надсилати додаткові HTTP-запити. Це зручно, коли потрібно повідомити інші сервіси про певну подію. Кожен задається через webHookDefinition. У ньому визначається метод запиту за допомогою setHttpMethodCommand, вказується адреса URL через setUrlCommand, а також можна додати заголовки. Тип тіла визначається через setResponseBodyTypeCommand. Сам вміст – через setValueCommand. Крім цього, можна встановити затримку перед надсиланням вебхука, використовуючи responseDelay. Така структура дає велику гнучкість і дозволяє не лише формувати стандартні, помилкові чи проксі-відповіді, а й розширити функціональність за допомогою інтеграцій з іншими сервісами. Приклад відповіді з затримкою та вебхуком наведена на рисунку 2.11.

```
DEFINE RESPONSE {
  ADD DIRECT RESPONSE {

    SET STATUS_CODE TO 200
    SET DYNAMIC_RESPONSE_TEMPLATING TO TRUE
      HEADERS {
        ADD HEADER {
          SET NAME TO "headerName"
          SET VALUE TO "headerValue"
        }
       }
      SET BODY_TYPE TO TEXT
      SET VALUE TO "response body query"
  }

  DELAY {
    SET TIME_UNIT TO MS
    SET DELAY_TYPE TO RANDOM_UNIFORM
    SET LOWER_BOUND TO 300
    SET UPPER_BOUND TO 400
  }

  WEB_HOOKS {
    ADD WEB_HOOK {
      SET METHOD TO POST
      SET URL TO "https://wc-echo.wiremockapi.cloud/webhook/echo"
        HEADERS {
          ADD HEADER {
            SET NAME TO "headerName"
            SET VALUE TO "headerValue"
          }
        }
        SET BODY_TYPE TO TEXT
        SET VALUE TO "BODY"
      }
    }
  }
}
```

Рисунок 2.11 – Приклад визначення проксі відповіді на запит до кінцевої точки через DSL

2.2 Аналіз DSL-коду та генерація конфігурацій

У цьому підрозділі розглядається процес парсингу коду написаного за допомогою DSL. Буде описано інструменти для лексичного і синтаксичного аналізу та обробку результатів парсингу. Останні використовуються для генерації конфігурацій для мок-сервісів з використанням WireMock.

2.2.1 Парсинг вхідного коду та відповідні інструменти

ANTLR – інструмент для створення лексичних і синтаксичних аналізаторів. Його розробив Терренс Парр, професор Університету Сан-Франциско. Він написаний на Java і підтримує генерацію парсерів для мов, таких як C#, C++, Dart, Java, JavaScript, TypeScript, Go, PHP, Python та Swift. ANTLR інтегрується в проєкти через бібліотеки та плагіни, зокрема Maven для Java. У нашому випадку цей засіб є ключовим для розробки, що генерує мок-сервіси. Він перетворює DSL-код на структуровані дані. ANTLR розбиває код на токени й будує абстрактне синтаксичне дерево, або abstract syntax tree (AST) за граматикою. Цим він забезпечує точну обробку команд.

ANTLR компілює граматику з написаного нами файлу DmytroMockDSL.g4 у код мовою програмування Java разом із лексером і парсером. Створений файл граматики містить лексичні та синтаксичні правила. Після компіляції генеруються класи DmytroMockDSLLexer і DmytroMockDSLParser для токенізації й аналізу. Лексер розпізнає токени, а парсер перевіряє їх синтаксис. Класи ANTLR інтегруються в Java-застосунок на сервері. Той ініціалізує лексер для обробки DSL-коду, створюючи потік токенів.

Лексер DmytroMockDSLLexer, створений ANTLR, розпізнає токени для DSL. Він охоплює ключові слова, символи, типи даних і конструкції для налаштування WireMock. Разом з цим обробляє код, генеруючи токени для синтаксичного аналізу. Підтримує коментарі та пробіли для точної токенізації.

Парсер створює AST для подальшої обробки. Це дерево представляє код структуровано і забезпечує зв’язок між вебінтерфейсом та сервером. У парсері

усі класи є внутрішніми формують ключові компоненти DSL. Вони представляють контексти для правил граматики. EndpointDefinitionRequestContext описує повне визначення кінцевої точки. RequestDefinitionContext визначає всю основну структуру вхідного запиту. ResponseDefinitionContext містить інформацію про необхідну відповідь. ResponseDelayContext конфігурує бажані затримки. ResponseWebHooksContext налаштовує вебхуки.

Для обробки DSL-коду на Java-сервері використовується Spring Boot. Він забезпечує надійну інфраструктуру для створення REST API. Також цей фреймворк спрощує налаштування сервера, обробку HTTP-запитів із фронтенд сторони, що передають код до парсера. Бібліотека antlr4-runtime інтегрується в проєкт для виконання лексичного та синтаксичного аналізу. Вона запускає згенеровані класи DmytroMockDSLLexer і DmytroMockDSLParser, які обробляють DSL-код.

ANTLR забезпечує надійний парсер для DSL, автоматизуючи обробку коду. Він підтримує складні граматики й масштабується для розширення мови. Автоматизація знижує помилки, а популярність цього інструменту гарантує стабільність. З граматикою DmytroMockDSL він створює основу для аналізу DSL-коду, готуючи його до подальшої обробки з метою генерації мок-сервісів із WireMock.

2.2.2 Синтаксичний аналіз DSL та обробка результатів парсингу

В цьому пункті буде надано огляд обробки результатів парсингу DSL. Реалізовано всю логіку було мовою програмування Java.

Діаграма послідовності показує процес обробки DSL-скрипта для налаштування мок-сервера з WireMock. Її можна побачити на рисунку 2.12. Вона відображає, як клієнт надсилає запит, який обробляється сервісами. Процес включає парсинг DSL, обробку запиту та відповіді, а також налаштування заглушки. Учасники – клієнт і компоненти системи для обробки даних. Деталі про складники будуть далі.

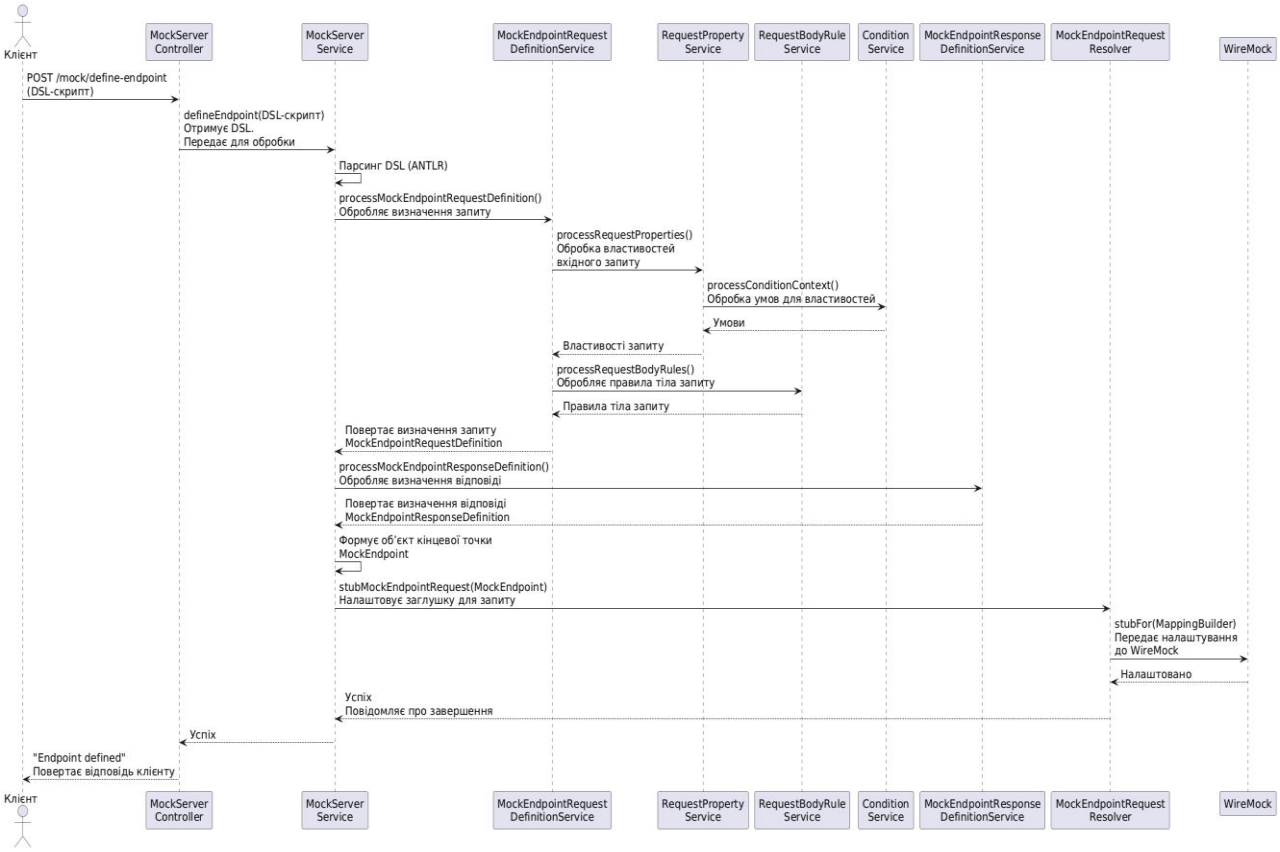

Рисунок 2.12 – Діаграма послідовності процесу обробки DSL для налаштування кінцевої точки

Лексер розбиває код на токени. Парсер аналізує їх за граматикою. Далі будується AST. Воно представляє структуру коду у вигляді класів. Наступне – ми переходимо до необхідного опрацювання результатів. Для цього було написано низку сервісних класів. Вони використовуються для аналізу результатів, валідації та уніфікації результатів. Класи з пакета org.dmytro.demodsl.parsingservice є частиною системи для обробки та виконання конфігурацій. Їх задачею є взаємодія із парсером DmytroMockDSLParser, обробляти структури запитів і відповідей, а також інтегруватися з бібліотекою WireMock для створення мок-серверів. Нижче наведено аналіз функцій кожного класу, що описує їхню роль у системі.

MockEndpointRequestDefinitionProcessingService клас відповідає за обробку контексту запиту RequestDefinitionContext з парсера. Його задача

створити об’єкт MockEndpointRequestDefinition, який описує імітований HTTP-запит для кінцевої точки. Він витягує тип методу, відповідності URL, шлях, пріоритет, а також викликає сервіси для обробки параметрів запиту. До них належать заголовки, файли кукі, параметри форми, тіло. Клас перевіряє коректність конфігурації. Наприклад, забороняє параметри запиту для певних типів URL, і забезпечує валідацію, щоб уникнути конфліктів між параметрами форми та правилами тіла. Логування допомагає відстежувати процес обробки.

Клас RequestPropertyProcessorService обробляє властивості, а саме параметри запиту, заголовки, файли кукі, параметри форми із контексту парсера. Він додає їх до об’єкта MockEndpointRequestDefinition, що є представленням сутності запиту. Він використовує додатково написані адаптери, такі як RequestQueryParamsDefinition, RequestHeadersDefinition, RequestCookiesDefinition, RequestFormParamsDefinition для перетворення контекстів парсера на списки об’єктів. Вони представлені у класах RequestQueryParam, RequestHeader, RequestCookie, RequestFormParam. Клас викликає ConditionProcessingService для обробки умов кожної властивості.

ConditionProcessingService обробляє умови із контекстів парсера, і перетворює їх на об’єкти типу Condition для використання в конфігурації запитів. Він аналізує прості, складені, а також заперечні умови, що надано. Обробляє в собі набір логічних типів SingleValueLogicalConditionType, PresenceLogicalConditionType, CompositeConditionLogicalType. Клас витягує типи умов і значення з контекстів, створюючи об’єкти SimpleCondition, PresenceCondition або CompositeCondition, і забезпечує валідацію, кидаючи винятки при відсутності необхідних даних. Цей сервіс є ключовим для обробки логіки умов у визначенні перевірок для параметрів запиту.

RequestBodyRuleProcessorService обробляє правила тіла запиту з контексту RequestBodyRulesContext парсера. Головною задачею є створення об’єктів RequestBodyRule для перевірки відповідних частин. Він підтримує різні типи правил, такі як порівняння значень, відповідність файлам XML, JSON, наявність даних, а також JSONPath/XPath. Клас витягує типи умов,

значення, імена та заперечення. Забезпечує валідацію і кидає винятки при некоректних даних.

MockEndpointResponseDefinitionService обробляє контекст відповіді ResponseDefinitionContext з парсера, та створює об’єкти для відповідей MockEndpointResponseDefinition. До наслідувачів цього класу є пряма відповідь DirectResponseDefinition, помилкова FaultResponseDefinition та проксі ProxyResponseDefinition. Він витягує код стану, тип тіла, заголовки, значення відповіді, а також налаштовує затримки DelayDefinition і вебхуки WebHook. Компонент підтримує різні типи затримок і забезпечує валідацію.

MockEndpointRequestResolver дуже важливий в контексті цієї роботи. Він інтегрує конфігурацію кінцевої точки класу MockEndpoint з бібліотекою WireMock і створює мок-запити та відповіді. Він перетворює об’єкти MockEndpointRequestDefinition і MockEndpointResponseDefinition на мапінги. Виконує для цього всі необхідні дії для налаштування. А саме через отримані дані класів виконує уніфікацію та приведення їх до необхідного формату та передає їх до API інструменту WireMock.

Усі ці класи разом формують систему для парсингу, обробки та виконання конфігурацій для імітування кінцевих точок.

2.2.3 Використання WireMock для генерації мок-сервісів на основі DSL

У цьому пункті розглядається, як працює клас MockEndpointRequestResolver для створення мок-сервісів. Він взаємодіє разом з WireMock, який запущено в Docker. Пояснюється, як DSL код перетворюється на налаштування WireMock і як працює з його API. Лістинг 1, що знаходиться у додатку А містить код цього класу.

MockEndpointRequestResolver клас є сервісом Spring, призначеним для обробки MockEndpoint об’єктів. Ті містять визначення запитів і відповідей кінцевих точок, яких ми хочемо створити. Клас починає з методу stubMockEndpointRequest, який приймає MockEndpoint і створює

MappingBuilder для WireMock. Наприклад, requestDefinition перетворюється на типи відповідності, такі як urlPathEqualTo для шляху URL і withMethod для HTTP-методу, використовуючи withHttpMethodForUrlPattern. Це відображає DSL на WireMock, робить своєрідну проєкцію на цей інструмент. Такий спосіб надає точне зіставлення запитів визначених більш абстрактно до технічних запитів нашого наявного мок-сервісу. Об’єкт responseDefinition транслюється в ResponseDefinitionBuilder, де задаються статус, заголовки через withHeader, тіло і його тип. Робиться це за допомогою конструкції withBody. Такий підхід гарантує, що структура DSL точно відображається і перетворюється на заглушки WireMock.

Для взаємодії з WireMock, MockEndpointRequestResolver використовує Java-бібліотеку wiremock-standalone. Вона розроблена, аби була можливість програмного доступу до API. Метод stubMockEndpointRequest викликає WireMock.stubFor(mappingBuilder) і реєструє мапінг у WireMock. Це альтернатива прямому HTTP POST до ресурсу "/__admin/mappings". Проте бібліотека швидша й інтегрована в Java-сервер. Наприклад, withHttpMethodForUrlPattern використовує статичні методи WireMock, такі як get чи post, для створення MappingBuilder, що спрощує налаштування. Такий підхід зменшує залежність від зовнішніх HTTP-запитів і підвищує надійність.

Клас обробляє спеціальні конструкції DSL через допоміжні методи. Для затримок, визначених у responseDelay, метод addDelayToResponse транслює їх у WireMock. Наприклад, фіксована затримка співвідноситься з withFixedDelay, випадкова з withUniformRandomDelay, а логнормальна з withLogNormalRandomDelay. Вебхуки, визначені в responseWebHooks, реалізуються через processWebHookDefinition, і дозволяє відправляти HTTP-запити після відповіді. Проксі-відповіді, як у proxyResponseDefinition, обробляються через processProxyResponseDefinition. Всі ці можливості реалізовані для того, аби гнучко тестувати складні сценарії, такі як імітація мережевих затримок чи побічні ефекти.

Мок-сервіси запускаються в контейнері інструменту Docker, де WireMock розгортається ізольовано. DevApplicationConfiguration налаштовує зв’язок, вказуючи хост і порт контейнера. MockEndpointRequestResolver відправляє конфігурації через WireMock API, використовуючи мережу Docker для комунікації. Це забезпечує стабільність і масштабованість, з можливістю запускати кілька контейнерів для різних тестів. Наприклад, образ wiremock/wiremock забезпечує готове середовище, а зв’язок через мережу полегшує інтеграцію з сервером на Java.

Загалом клас MockEndpointRequestResolver автоматизує створення мок-сервісів, транслюючи оброблений DSL у WireMock конфігурації передаючи їх через бібліотеку Java. Він обробляє спеціальні конструкції, налаштовує правила та створює конфігурації для управління мок-сервісом у Docker. Цей підхід спрощує тестування SOA систем та зменшує залежність від ручного налаштування.

2.3 Інтеграція з сервісом WireMock

WireMock – це популярний інструмент із відкритим кодом для створення макетів HTTP API. Він дозволяє розробникам і тестувальникам імітувати поведінку реальних API. Це корисно, коли такі системи недоступні, не стабільні чи неготові. Цей сервіс допомагає створювати стабільні середовища для тестування та розробки.

2.3.1 Функціональні можливості сервісу

Основною функцією WireMock є створення заглушок кінцевих точок, що повертають заздалегідь визначені HTTP-відповіді для запитів. Вони мають відповідати певним критеріям [25]. Заглушки можна налаштувати через JSON-файли в теці mappings, REST API або Java API. Наприклад, для GET-запиту на /some/thing можна створити імітацію, яка повертає статус 200, тіло "Hello!!!" і заголовок Content-Type: text/plain. Підтримуються всі стандартні HTTP-

методи: GET, POST, PUT, DELETE, HEAD, TRACE, OPTIONS, GET_OR_HEAD, ANY.

Для складніших сценаріїв є підтримка пріоритетів імітованих кінцевих точок, що дозволяє визначити порядок обробки. Наприклад, заглушка із пріоритетом 1 для "/api/specific-resource" матиме перевагу над тією, що із пріоритетом 5 для "/api/.*". Тіло відповіді можна вказати як рядок, JSON через jsonBody, або завантажити з файлу в теку __files за допомогою withBodyFile. Підтримується також бінарний вміст через byte[] або base64 у JSON API, з обов'язковим кодуванням UTF-8.

Запити можна перевіряти за допомогою різних критеріїв: точний збіг, регулярні вирази, JSONPath, XPath. Доступні типи відповідності для URL, параметрів запиту, заголовків, файлів кукі, тіла запиту. Це дає можливість точно налаштувати, коли заглушка має спрацьовувати. Наприклад, можна перевірити, чи містить заголовок певне значення, або чи відповідає тіло запиту певній схемі JSON. З версії 3.13.0 додано підтримку порівняння IP-адреси клієнта.

Шаблонізація відповідей дозволяє динамічно генерувати відповіді за допомогою Handlebars. Це включає доступ до атрибутів запиту, таких як request.url, request.headers, request.body. З версії 3.8.0 також request.bodyAsBase64 і request.multipart. Наприклад, можна створити відповідь, яка включає сегмент шляху запиту, як {{request.pathSegments.[1]}}. Шаблонізацію можна увімкнути глобально через options().globalTemplating(true) або для окремої заглушки, додавши трансформер response-template. Є підтримка кешування шаблонів із можливістю задати обмеження розміру кешу. Наприклад, options().withMaxTemplateCacheEntries(10000).

Перевірка запитів дозволяє переконатися, що певні запити були надіслані до сервера. Це включає перевірку кількості запитів, використання операторів, таких як lessThan, moreThan, або фільтрацію за датою, обмеження розміру результатів. Можна отримати всі запити через GET на ресурс

"__admin/requests" або знайти невідповідні запити через WireMock.findUnmatchedRequests(). З версії 3.13.0 додано можливість знаходити найближчі невідповідності для аналізу.

WireMock можна розширити через плагіни, додаючи власну логіку, наприклад, для фільтрації запитів, трансформації відповідей чи прослуховування подій. Це робить інструмент гнучким для складних сценаріїв.

WireMock – дуже потужний і зручний інструмент. З його використанням можна створювати гнучкі мок-відповіді для різних HTTP-запитів. Він підтримує багато методів, умов, шаблонів і навіть дозволяє перевіряти, які запити реально були відправлені. Його можна налаштовувати через JSON, REST API або Java, а ще додавати плагіни. Це робить його ідеальним варіантом для тестування без справжнього бекенду.

2.3.2 Налаштування WireMock для створення мок-сервісів

Налаштування WireMock гнучке, але особливу увагу приділимо розгортанню через Docker, що ідеально підходить для сучасних CI/CD-пайплайнів. Документація на офіційному сайті детально описує процес, і робить акцент на простоті та адаптивності. Версія 3.13.0 є актуальною. Інструмент підтримує розгортання через JAR-файл, REST API, Java API та Docker, але контейнери виділяються своєю зручністю.

Для налаштування, як окремого сервера треба виконати ряд дій. По-перше, необхідно здійснити завантаження файлу JAR з офіційного сайту. Далі виконати команду "java -jar wiremock-standalone-3.13.0.jar", і сервер запуститься на порту 8080. Є можливість змінити порт за допомогою передачі параметра --port в команду. При запуску створюються дві теки: mappings для JSON-файлів із визначенням зашлушок і __files для файлів, які можна обслуговувати як відповіді. Для визначення макетів створюють файли JSON в теці mappings.

Docker забезпечує ізольоване середовище для WireMock, що усуває проблеми з конфігурацією. Офіційний образ wiremock/wiremock:3.13.0 легко розгортається однією командою. Наприклад, команда "docker run -it --rm -p 8080:8080 wiremock/wiremock:3.13.0" запускає сервер на порту 8080. Це дозволяє швидко розгорнути сервіс без встановлення Java чи інших залежностей. Додатково можна увімкнути томи для передачі файлів mappings та __files, що містять заглушки та ресурси.

Налаштування через Docker підтримує додаткові параметри. Наприклад, опція --port змінює порт сервера, а --https-port вмикає HTTPS. Захист адмін-API можливий через --admin-api-basic-auth, що додає базову автентифікацію. Ці параметри передаються через аргументи командного рядка в Docker. Такий спосіб спрощує керування імітованими кінцевими точками без редагування контейнера.

Java API дозволяє програмно налаштовувати макети, інтегрувати їх у тести та керувати поведінкою сервера. Воно є основним для тестів у JVM-екосистемі, таких як JUnit чи TestNG, і підтримує складні сценарії. Центральним класом є WireMockServer, який керує запуском і зупинкою сервера. Його створюють із конфігурацією через WireMockConfiguration, де можна вказати порт, HTTPS чи шлях до файлів. Метод start запускає сервер, а stop припиняє його роботу. Наприклад, у тестах JUnit сервер ініціалізується в методі BeforeEach і зупиняється в AfterEach. WireMockServer також надає методи для створення імітованих кінцевих точок і перевірки запитів, що робить його основою для програмного керування.

Для створення заглушок використовують клас StubMapping, який представляє один макет. Його створюють через статичний метод stubFor, що приймає об’єкт MappingBuilder. Той зі свого боку визначає критерії запиту, такі як URL чи метод HTTP, через методи get, post, put тощо. Метод willReturn з об’єктом ResponseDefinitionBuilder задає відповідь, включаючи статус, заголовки чи тіло. StubMapping додається до сервера через WireMockServer або WireMockRule, яка спрощує інтеграцію в тести.

Загалом цей інструмент вражає гнучкістю і можливостями. Java API дозволяє легко інтегрувати його в тести JUnit чи TestNG, а класи WireMockServer і StubMapping забезпечують повний контроль над сервером і імітованими кінцевими точками. Docker спрощує розгортання, усуваючи проблеми з конфігурацією. Він зручний і в запуску через JAR разом із динамічним керуванням через REST API. WireMock ідеально вписується в CI/CD, економить час і робить тестування надійним.

3 АНАЛІЗ РЕЗУЛЬТАТІВ ДОСЛІДЖЕННЯ

У цьому розділі ми опишемо загальну архітектуру створеного рішення. Розглянемо, з яких компонентів складається система. Пояснимо, які модулі вона має, та як вони взаємодіють між собою. Далі зосередимося на прикладному рішенні. Розглянемо інтерфейс, через який користувач може редагувати код через DSL. Опишемо, як використання Monaco Editor полегшує процес редагування. Також пояснимо, як створений код передається на сервер.

Завершимо розділ оцінкою отриманих результатів дослідження. Проаналізуємо, наскільки ефективним виявився підхід і чи відповідає реалізація поставленим на початку цілям.

3.1 Опис архітектури рішення

У цьому підрозділі буде надано узагальнений огляд архітектури системи. Ми опишемо основні компоненти, які її складають, та визначимо їхні функції. Особливу увагу приділимо тому, як вони взаємодіють між собою. Метою цього опису є показати, як побудова рішення дозволяє створювати та використовувати мок-сервіси.

3.1.1 Компонентна структура системи

Система розроблена для створення та управління мок-сервісами в SOA. Вона використовує DSL для спрощення процесу. Структура модульна, тому можна легко додавати нові функції чи адаптувати до змін у вимогах. Складається з чотирьох основних компонентів: вебінтерфейсу, проміжного програмного забезпечення у вигляді сервера на Java, бази даних MongoDB та інструменту WireMock. Кожен з них має чітко визначену роль. Їхня взаємодія побудована на стандартних технологіях, що забезпечує надійність і простоту.

Вебінтерфейс є основною точкою взаємодії користувача з системою. Він надає зручне середовище для написання та редагування коду DSL. Для цього, як редактор використовується Monaco Editor, відомий своєю потужною

функціональністю. Завдяки йому є підсвітка синтаксису, автодоповнення та валідація коду в реальному часі.

Інтерфейс побудовано на TypeScript із використанням фреймворку React та збирається за допомогою Vite. Такий підхід спричиняє високу швидкість розробки, миттєве оновлення вигляду та зручність користування. Код, який вводить користувач, надсилається до серверної частини через HTTP/REST API для подальшої обробки.

Проміжне програмне забезпечення, реалізоване на Java. Воно є центральним компонентом системи. Відповідає за обробку DSL-коду, отриманого від вебінтерфейсу. Для парсингу коду використовується ANTLR. Він точно аналізує синтаксис і створює AST. На основі цього дерева цей інструмент генерує конфігурації для мок-сервісів у форматі, зрозумілому для WireMock. Для реалізації серверної логіки використовується фреймворк Spring Boot, який спрощує створення REST-сервісів. Сервіс також взаємодіє з базою даних MongoDB і зберігає надіслані скрипти проблемно-орієнтованої мови, статус їх обробки та проміжні напрацювання. Це зручно, коли користувачам треба повертатися до попередньої роботи через функцію QuickSave.

MongoDB виступає як головне сховище даних для збереження DSL скриптів, результатів парсингу та тимчасових збережень. У системі передбачено дві основні колекції – definition_attempts і quick_saves. Перша містить усі спроби інтерпретації коду, з інформацією про успішність і можливі помилки. Це дає змогу зберігати історію використання системи та діагностувати помилки. Друга дозволяє зберігати останній варіант коду без створення повної конфігурації, що зручно для проміжного збереження під час редагування. Для роботи з цими колекціями створено відповідні сервіси. Вся логіка зберігання побудована з використанням Spring Data для MongoDB.

WireMock використовується як головний осередок для налаштування мок-сервісів. Він розгорнутий у контейнері Docker, що забезпечує ізольоване середовище та спрощує його розгортання. WireMock функціонує як окремий

сервіс, який приймає HTTP-запити від проміжного програмного забезпечення на Java. У цих запитах передаються конфігурації для імітованих систем, на основі яких створюються імітації кінцевих точок.

3.1.2 Ролі та взаємозв'язки між модулями

Кожен модуль системи виконує власну функцію. Вебінтерфейс – точка взаємодії з користувачем. Java-сервер обробляє логіку. MongoDB зберігає дані. WireMock генерує мок-відповіді. Взаємодія між компонентами побудована на стандартних технологіях.

Вебінтерфейс – це основний спосіб взаємодії користувача з системою. Він побудований на React, TypeScript і Vite. Використовує Monaco Editor для редагування змісту. Monaco забезпечує підсвітку синтаксису, автодоповнення та валідацію в реальному часі. Користувач вводить код DSL у редакторі, де той перевіряється на клієнтській стороні. Після цього є можливість відправити його на сервер через REST API. Інтерфейс швидкий і зручний. Цей інструмент забезпечує миттєве оновлення змін.

Java-сервер містить основну логіку перетворення та створення конфігурацій. Він побудований на Spring Boot. Отримує код DSL від вебінтерфейсу. Використовує ANTLR для парсингу коду. Цей парсер створює абстрактне синтаксичне дерево AST. На його основі обробник генерує конфігурації для WireMock. Паралельно взаємодіє з MongoDB, зберігаючи DSL-скрипти, статуси обробки та проміжні збереження. Сервер є центральним хабом для обробки даних.

MongoDB виступає в ролі основного сховища. У ньому зберігаються скрипти DSL, історія спроб, конфігурації та проміжні збереження. Це дозволяє повертатися до попередніх версій, переглядати помилки або відновлювати робочі сценарії.

WireMock є окремим контейнером, який виконує інструкції, отримані від сервера Java. Він приймає HTTP-конфігурації та створює імітовані кінцеві точки. Клієнтські або тестові системи можуть надсилати запити до цього

компоненту та отримувати відповіді, які імітують реальні сервіси. Це дозволяє тестувати поведінку без потреби звертатися до зовнішніх API.

Уся система побудована на принципі взаємодії через стандартні вебтехнології. Вебінтерфейс надсилає код до центрального сервера Java. Він, зі своєї сторони обробляє його і передає готові дані до WireMock. Паралельно сервер читає або записує дані до бази MongoDB. WireMock надає відповіді, які імітують зовнішні сервіси, замінюючи реальні API у тестовому середовищі.

Діаграма компонентів для цієї взаємодії наведена на рисунку 3.1.

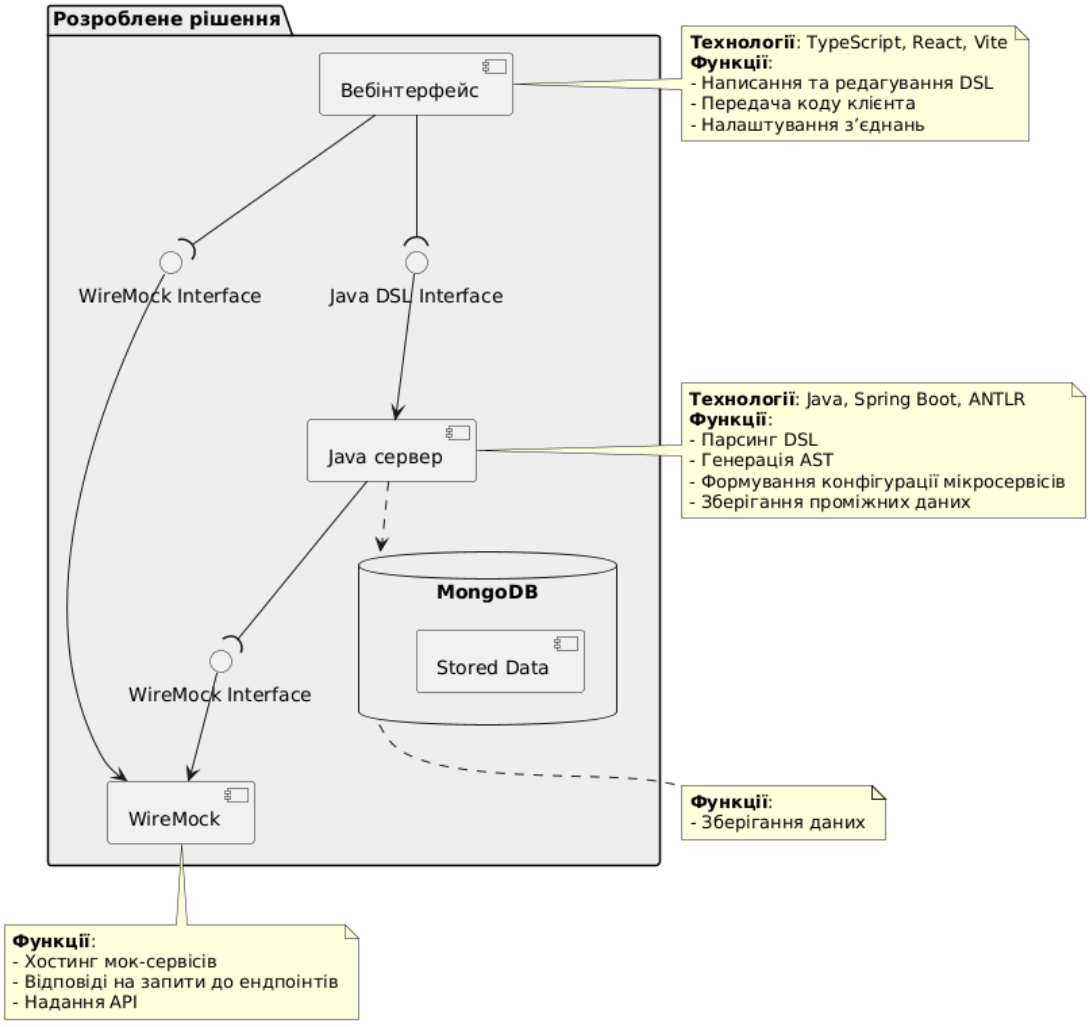

Рисунок 3.1 – Діаграма компонентів для створеної системи

3.2 Опис програмного рішення

У цьому розділі описано структуру клієнтської частини системи, яка забезпечує зручне редагування та обробку DSL-коду. Особливу увагу

приділено інтерфейсу користувача, використанню Monaco Editor та механізмам передачі коду до серверної частини для подальшої обробки.

3.2.1 Інтерфейс користувача для редагування DSL-коду

У цьому пункті буде показано, як реалізовано компонент для взаємодії користувача з системою. Розглядається структура вебінтерфейсу, можливості, які він надає, зокрема для введення та редагування DSL-коду. Також описуються дії з редактором та іншими елементами інтерфейсу.

Рисунок 3.2 – Зображення головної сторінки інтерфейсу користувача

На рисунку 3.2 показано головну сторінку "Create Endpoint" вебінтерфейсу для створення DSL-коду. Основним елементом є текстовий редактор із підсвіткою синтаксису. Код структуровано, розбито на рядки з номерами та має вкладені блоки зі власною логічною структурою. Кожна команда має кольорове виділення.

Ліворуч розташована бічна панель із меню. Білі написи допомагають користувачу швидко переходити між функціями системи. У ній є кілька опцій: "Create Endpoint", "Mappings", "Request History", "Endpoint Creation Attempts" і "Settings". Кожна з них позначена іконкою.

Кнопки під редактором відображають основні функції роботи з DSL. Кнопка "Send Content" відправляє поточний фрагмент на обробку до Java-сервера. Це основна дія для визначення кінцевої точки. "Format" автоматично вирівнює текст, додаючи правильні відступи та структуру. "Check Syntax" виконує перевірку синтаксису коду. Вона дозволяє виявити помилки ще до відправлення або збереження. "Quick Save" зберігає поточну версію написаного в системі. "Quick Load" завантажує збережений раніше сегмент, що пришвидшує повторне використання коду. "Clear" очищає весь вміст редактора. "Fill by Draft" автоматично заповнює зміст кодом із заготовленого шаблону. Це зручно для швидкого старту або тестування. Нижче цього ряду розташована окрема кнопка "Open Docs", що відкриває документацію. Вона дає змогу швидко знайти опис команд DSL або приклади їх використання.

Нижче панелі кнопок видно область з текстовим представленням написаного DSL у вигляді AST. Це – результат парсингу введеного коду у внутрішню модель інструменту ANTLR.

На рисунку 3.3 наведено секцію для перегляду списку створених імітованих кінцевих точок. У центрі екрана розташований список відображень. Заголовок "Mappings" написаний зверху великими чорними літерами. Кожен мапінг представлений у вигляді блоку. Він має унікальний ідентифікатор і категорії, позначені стрілками: "Request", "Response" і "Full Mapping". Вона згорнута, але її можна розгорнути для перегляду деталей про визначений в ній зміст та правила. Блоки мають кнопку "Delete" праворуч для можливості видалення вказаної кінцевої точки. Зверху праворуч є кнопка "Delete All" для усунення всіх мапінгів одразу.

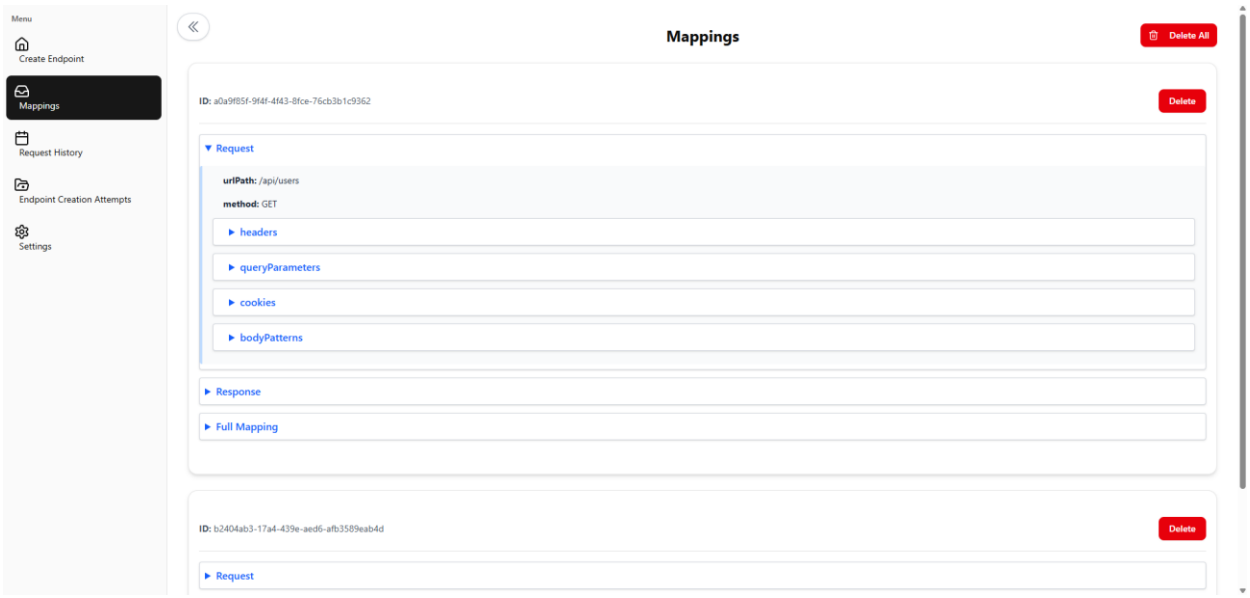

Рисунок 3.3 – Зображення сторінки інтерфейсу для перегляду створених кінцевих точок

На рисунку 3.4 наведено секцію для перегляду історії запитів на імітовані кінцеві точки.

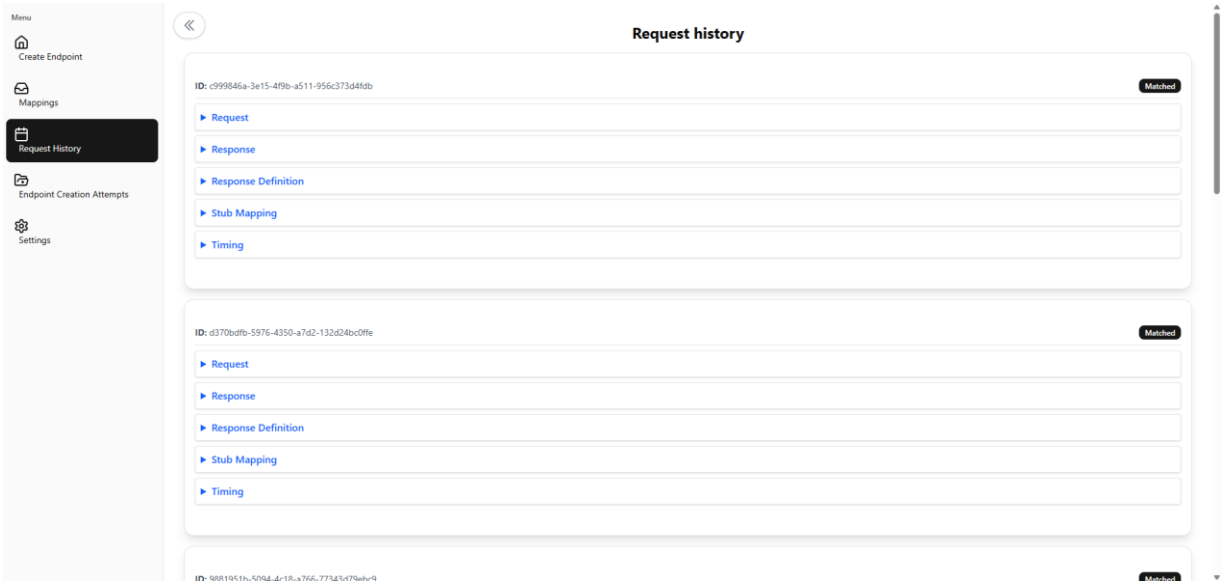

Рисунок 3.4 – Зображення сторінки інтерфейсу для перегляду історії запитів на кінцеві точки

У центрі екрана розташований список записів для перегляду історії запитів. Заголовок "Request history" написаний зверху великими чорними

літерами. Кожен запис представлений у вигляді блоку, що має ідентифікатор та категорії: "Request", "Response", "Response Definition", "Stub Mapping" і "Timing". Кожна з них згорнута. Це дозволяє швидко переглядати список без зайвих деталей. Праворуч від кожного представлення є візуальне позначення "Matched", якщо запит збігся з імітованою кінцевою точкою, та "Not matched" в протилежному разі.

На рисунку 3.5 наведено секцію для перегляду списку спроб для створення імітованих кінцевих точок.

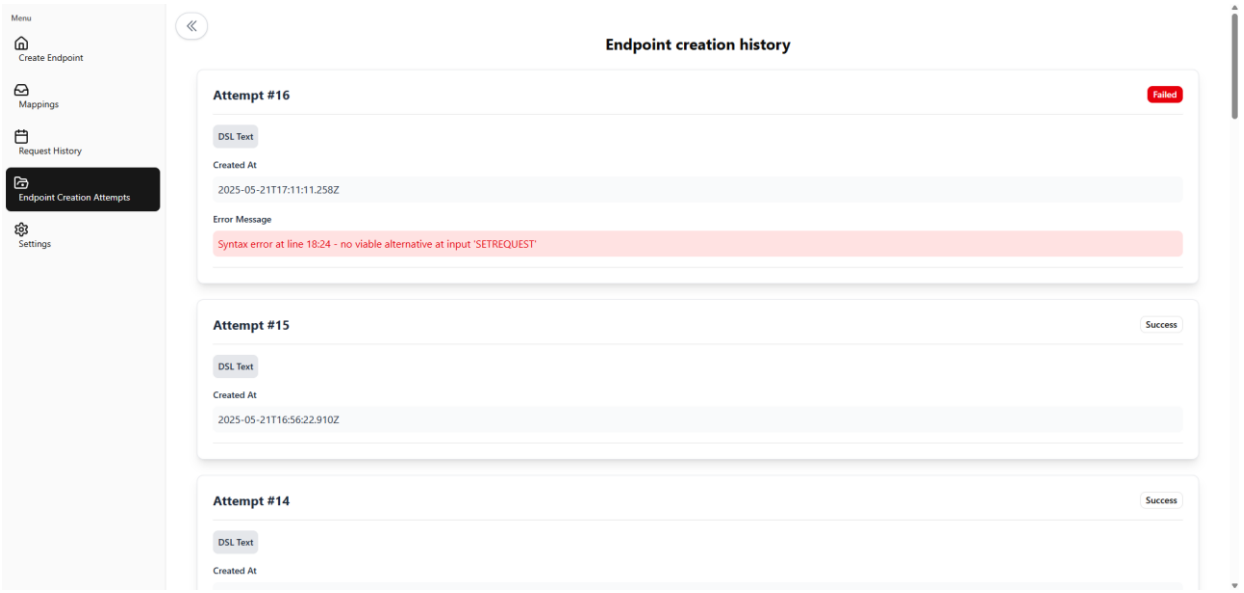

Рисунок 3.5 – Зображення сторінки інтерфейсу для перегляду історії спроб створення імітованих кінцевих точок

Інтерфейс для перегляду історії створення кінцевих точок містить список спроб створення. Заголовок "Endpoint creation history" написаний зверху великими чорними літерами. Спроба має поле "DSL Text", яке згорнуте та містить у собі відповідних фрагментів, з яким виконувалася спроба. Нижче наведено дату її створення. В правому кутку блоку міститься маркер для позначення статусу успішності створення. Додатково виводиться повідомлення про помилку, за наявності.

Секцію з налаштуваннями зображено на рисунку 3.6.

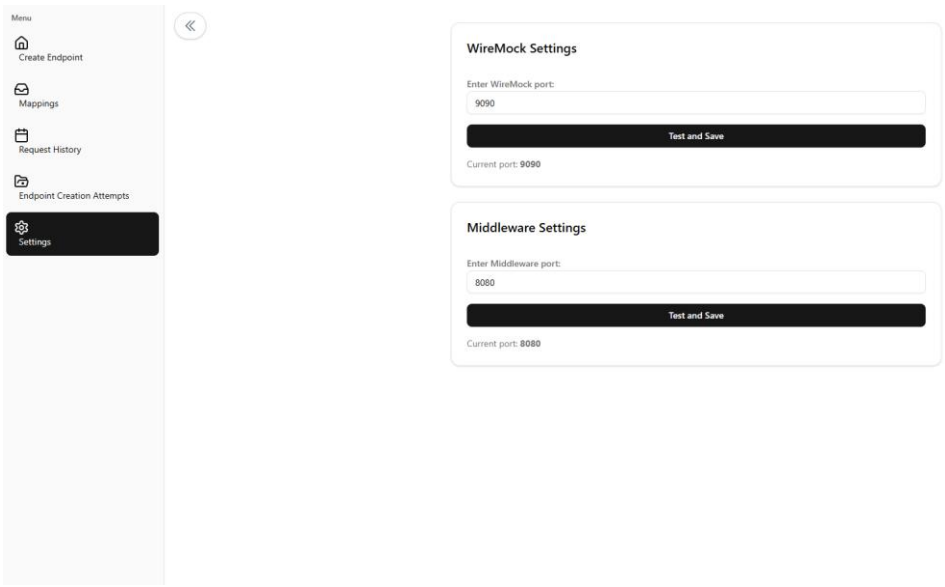

Рисунок 3.6 – Зображення сторінки інтерфейсу для налаштування застосунку

Інтерфейс для налаштувань має два основні блоки, кожен з яких відповідає за підключення до окремого сервісу.

Першим блоком є "WireMock Settings". Він використовується для підключення до WireMock. У полі "Enter WireMock port" користувач вводить порт, на якому працює цей сервіс. Кнопка "Test and Save" перевіряє з’єднання з наведеним і зберігає його, якщо все працює. Нижче відображається напис "Current port", який показує поточний активний порт. Це допомагає переконатися, що налаштування застосовані.

Другий блок – це "Middleware Settings". Він служить для підключення до сервера на Java. Тут також є поле з написом "Enter Middleware port", куди вводиться номер порту. Кнопка "Test and Save" працює аналогічно з попереднім блоком. Напис "Current port" вказує, який саме використовується зараз.

3.2.2 Використання Monaco Editor для покращення процесу редагування

Monaco Editor, відомий як основа редактора Visual Studio Code. Це потужний інструментом для редагування, зокрема для роботи з DSL. Далі наведено детальний аналіз його ключових можливостей.

Одна з найважливіших можливостей – це підсвітка синтаксису. Monaco Editor дозволяє налаштувати виділення різних елементів DSL, таких як ключові слова, значення та конструкції різними кольорами. Це візуально розмежовує частини коду та робить його більш читабельним. Приклад можна побачити на раніше наведеному рисунку 3.2.

Наступна ключова функція – автодоповнення. Інструмент пропонує розумні підказки під час набору тексту, базуючись на контексті. Це скорочує час, необхідний для написання коду, і зменшує ймовірність синтаксичних помилок. Цю можливість можна побачити на рисунку 3.7. Monaco Editor також може бути налаштований для миттєвої перевірки DSL на наявність помилок під час набору.

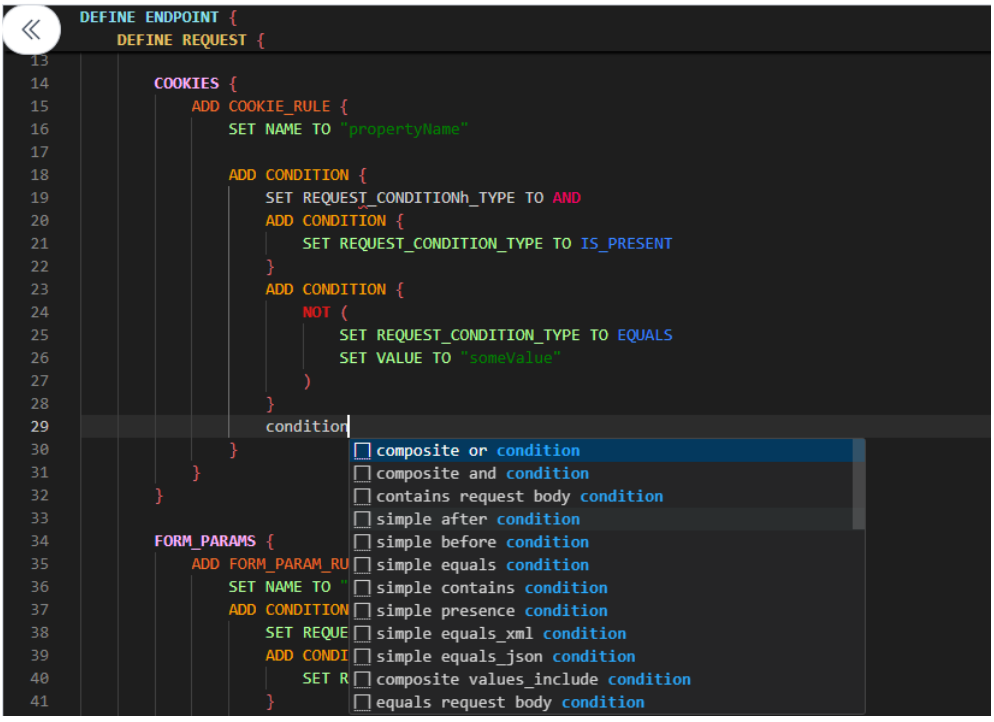

Рисунок 3.7 – Зображення можливості підказок в застосунку

Ще однією корисною функцією є згортання коду. Вона дозволяє користувачам згортати та розгортати розділи. Це корисно для роботи з довгими файлами, де користувач може зосередитися лише на потрібній частині. Приклад на рисунку 3.8. Окрім цього, Monaco Editor підтримує

документацію та підказки, які можуть бути налаштовані для DSL. Користувачі можуть навести курсор на команду і побачити пояснення чи приклад використання.

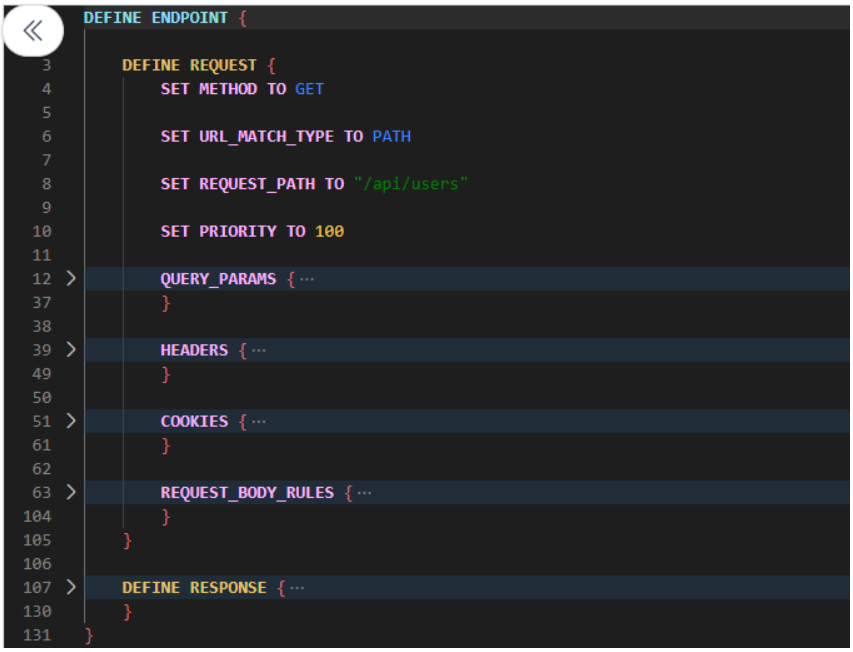

Рисунок 3.8 – Приклад згортання блоків коду DSL

Ці можливості створюють швидкий, зручний і менш схильний до помилок процес редагування DSL. Вони зменшують навантаження на користувачів, підвищують продуктивність і роблять роботу більш інтуїтивною.

3.2.3 Передача коду до серверної логіки обробки

Передача відбувається з головної сторінки інтерфейсу. Кнопка "Send Content" ініціює процес надсилання DSL-коду на сервер для обробки. При натисканні викликається функція sendEditorContent, яка отримує значення з редактора, формує HTTP POST-запит на адресу "http://localhost:{port}/mock/define-endpoint", де {port} завантажується з локального сховища браузера. Запит надсилається у вигляді звичайного тексту, що відповідає специфіці DSL. Після відправлення вмикається індикатор завантаження, і залежно від відповіді сервера користувач отримує

повідомлення про успіх або помилку. У разі успіху відображається повідомлення з текстом відповіді, у разі помилки – інформація про збій. Приклад такого надсилання наведено на рисунку 3.9.

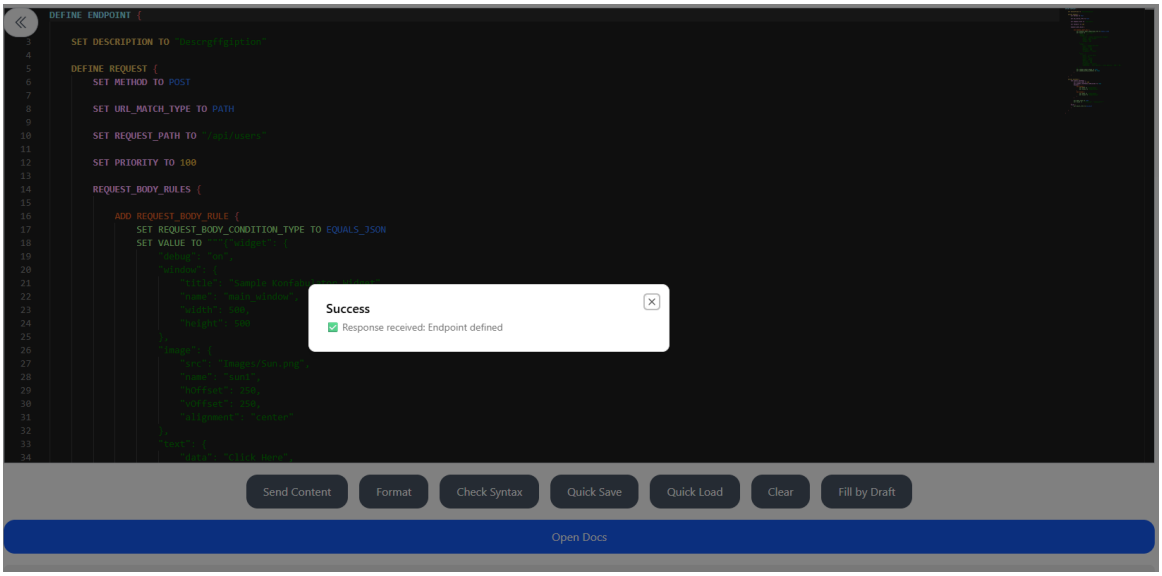

Рисунок 3.9 – Приклад надсилання коду DSL на сервер для обробки

3.3 Оцінка одержаних результатів

Дослідження створило ефективний інструмент для конфігурації мок-сервісів у SOA за допомогою DSL. Це спрощує створення та управління мок-сервісами, робить процес інтуїтивним і швидким. Система базується на сучасних технологіях: ANTLR для парсингу, Java та Spring Boot для проміжного програмного забезпечення, TypeScript, React, Monaco Editor для вебінтерфейсу та WireMock у Docker для мок-сервісів.

Усі заплановані завдання були успішно завершені. Це включало аналіз методів і інструментів, що існують, дизайн чіткого синтаксису DSL, реалізацію надійного проміжного програмного забезпечення з Java та ANTLR, створення зручного вебінтерфейсу з Monaco Editor і перевірку стабільності та зручності системи.

Система працює, як очікувалося. DSL ефективно генерує коректні конфігурації для WireMock. Вебінтерфейс забезпечує полегшене написання коду. Інтеграція з WireMock у Docker забезпечує ізольоване та кероване

середовище. Загалом, система скорочує час і зусилля, потрібні для налаштування мок-сервісів, порівняно з традиційними методами. Приклад використання тестової кінцевої точки наведено на рисунку 3.10.

Рисунок 3.10 – Приклад використання тестової кінцевої точки

Робота відповідає глобальним тенденціям у програмній інженерії, які підтримують використання DSL для вирішення спеціалізованих завдань. Вона вирішує конкретну потребу тестування в SOA та мікросервісах, що є важливим у сучасних архітектурах систем.

Система придатна для команд тестування програмного забезпечення, розробників, які працюють над SOA, та організацій, що потребують ефективних можливостей для мок-тестів. Вона цінна в умовах швидких циклів розробки та тестування.

Економічно, система, що була розроблена в рамках дослідження, може знизити витрати. Це можливо через скорочення часу тестування та зменшення помилок. Наслідком є прискорення виведення продуктів на ринок і підвищення їхньої якості. Науково, вона демонструє практичне застосування DSL у тестуванні та сприяє розвитку програмної інженерії. Соціально, роблячи перевірки доступнішим. Вона дозволяє ширшому колу користувачів ефективно брати участь у розробці та випробуванні програмного забезпечення.

ВИСНОВКИ

Розробка рішення для створення та управління мок-сервісами в архітектурі SOA за допомогою DSL є значним досягненням, яке відповідає сучасним потребам програмної інженерії. Дослідження, проведене в рамках цього проєкту, успішно досягло мети реалізації інструменту. Воно спрощує і прискорює процес конфігурації мок-сервісів. Платформа базується на модульній побудові. Вона гнучка, масштабована і портативна.

Особливо цінними моментами проєктування та реалізації є її здатність автоматизувати процес створення мок-сервісів. Інструмент зменшує рутинну роботу та час для ручної конфігурації, порівняно з засобами, такими як WireMock і MockServer. DSL абстрагує технічні деталі, тим самим створює можливість описувати поведінку певних компонентів на високому рівні. З цього випливає зростання продуктивності та зниження ймовірності помилок. Платформа також демонструє гнучкість, підтримуючи складні сценарії, такі як умовна логіка, динамічні відповіді, симуляція мережевих затримок. Це позиціює її, як універсальну для тестування SOA-систем.

Для продовження та вдосконалення рішення є кілька перспективних напрямків. По-перше, оптимізація продуктивності платформи, особливо для обробки великих проєктів із численними мок-сервісами, могла б зменшити затримки в парсингу та генерації конфігурацій. По-друге, розширення підтримки протоколів за межі HTTP/HTTPS. Наприклад, WebSocket або gRpc, дозволило б застосовувати її в архітектурах, таких як IoT чи системах реального часу. По-третє, посилення безпеки, зокрема шифрування збережених DSL-скриптів у MongoDB. Можливість покращення автентифікації для WireMock і проміжного сервера, для забезпечення захисту даних у виробничих середовищах, де можуть виникати ризики витоку інформації.

Крім того, доцільно розглянути гібридну модель, яка поєднує текст і графіку. Це дозволить задовольнити потреби як технічних, так і менш

досвідчених користувачів. Хоча візуальні інтерфейси складніші у реалізації, вони можуть зробити систему привабливішою для ширшої аудиторії. Додатково, реальні випробування в різних середовищах дозволили б краще виявити слабкі місця пов’язані з навантаженням. Це може включати тестування в CI/CD або проєктах з великою кількістю сервісів.

У цілому, запропоноване рішення є вагомим внеском у сферу тестування SOA. Воно забезпечує автоматизацію, знижує ризик помилок і покращує взаємодію в командах. Також може надихнути нові дослідження в інших галузях, наприклад, у кібербезпеці. У перспективі ця робота може стати важливою частиною сучасної інфраструктури створення програм, особливо в умовах складних розподілених систем.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1. Domain-Specific Languages / M. Fowler et al. Boston : Addison-Wesley Professional, 2010. 640 p.

2. What developers need to know about domain-specific languages. URL: https://opensource.com/article/20/2/domain-specific-languages. Дата звернення: 25.03.2025.

3. Domain-Specific Languages Guide. URL: https://martinfowler.com/dsl.html. Дата звернення: 29.03.2025.

4. The complete guide to (external) Domain Specific Languages. URL: https://tomassetti.me/domain-specific-languages/. Дата звернення: 29.03.2025.

5. Cucumber Documentation: Gherkin Syntax. URL: https://cucumber.io/docs/gherkin/. Дата звернення: 29.03.2025.

6. Querydsl Reference Guide. URL: http://querydsl.com/static/querydsl/5.0.0/reference/html_single/. Дата звернення: 02.04.2025.

7. Domain-Specific Languages. URL: https://www.jetbrains.com/mps/concepts/domain-specific-languages/. Дата звернення: 05.04.2025.

8. Parr T. The Definitive ANTLR 4. Reference. Raleigh : Pragmatic Bookshelf, 2013. 328 p.

9. Mernik M., Heering J., Sloane A. M. When and How to Develop Domain-Specific Languages // ACM Computing Surveys. New York : Association for Computing Machinery, 2005. Vol. 37, No. 4. P. 316–344.

10. DSL Engineering: Designing, Implementing and Using Domain-Specific Languages. URL: https://voelter.de/dslbook/markusvoelter-dslengineering-1.0.pdf. Дата звернення: 10.04.2025.

11. WireMock GitHub Repository. URL: https://github.com/wiremock/wiremock. Дата звернення: 12.04.2025.

12. WireMock Case Studies. URL: https://www.wiremock.io/case-studies. Дата звернення: 12.04.2025.

13. WireMock Documentation. URL: https://wiremock.org/docs/. Дата звернення: 12.04.2025.

14. 10 Best API Mocking Tools. URL: https://apidog.com/blog/best-api-mock-tools/. Дата звернення: 13.04.2025.

15. MockServer Official Website. URL: https://www.mock-server.com/. Дата звернення: 13.04.2025.

16. Beeceptor Official Website. URL: https://beeceptor.com/. Дата звернення: 16.04.2025.

17. MoldStud. Comparative Review - The Best API Mocking Tools of 2025. URL: https://moldstud.com/articles/p-comparative-review-the-best-api-mocking-tools-of-2025. Дата звернення: 20.04.2025.

18. What are the benefits and challenges of using mock services in UAT? URL: https://www.linkedin.com/advice/0/what-benefits-challenges-using-mock-services. Дата звернення: 20.04.2025.

19. Challenges and Benefits of Mocking External Services With Examples. URL: https://www.thegreenreport.blog/articles/challenges-and-benefits-of-mocking-external-services-with-examples/challenges-and-benefits-of-mocking-external-services-with-examples.html. Дата звернення: 03.05.2025.

20. Understanding Domain-Specific Language (DSL) in Test Automation: Focusing on Business Logic and User Behaviors. URL: https://medium.com/%40aydinserbest34/understanding-domain-specific-language-dsl-in-test-automation-focusing-on-business-logic-and-62fe9a79b1e6. Дата звернення: 10.05.2025.

21. Mock Testing: Understanding the Benefits and Best Practices. URL: https://www.qodo.ai/blog/mock-testing/. Дата звернення: 12.05.2025.

22. Spadini D., Aniche M., Bruntink M., Bacchelli A. To Mock or Not To Mock? An Empirical Study on Mocking Practices // Proceedings of the 14th IEEE/ACM

International Conference on Mining Software Repositories (MSR). New York : IEEE/ACM, 2017. P. 402–412.

23. 32 Software Testing Statistics for Your Presentation in 2025. URL: https://www.globalapptesting.com/blog/software-testing-statistics. Дата звернення: 15.05.2025.

24. Гамзаєв Р.О., Ткачук М.В. Розробка проблемно-орієнтованої мови моделювання для підтримки варіабельності програмного забезпечення в системах "Розумний будинок" // Сучасний стан наукових досліджень та технологій в промисловості. 2023. № 1 (23). С. 45–56. URL: https://doi.org/10.30837/ITSSI.2023.23.045, 17.05.2025.

25. WireMock Stubbing. URL: https://wiremock.org/docs/stubbing. Дата звернення: 17.05.2025.

ДОДАТОК А. ВИХІДНИЙ КОД

Лістинг 1 – клас MockEndpointRequestResolver

```
@Service
@Slf4j
@RequiredArgsConstructor
public class MockEndpointRequestResolver {

    private static final String UNSUPPORTED_CONDITION =
"Unsupported condition: %s";

    public void stubMockEndpointRequest(MockEndpoint
mockEndpoint) {
        MockEndpointRequestDefinition request =
mockEndpoint.getRequestDefinition();
        MockEndpointResponseDefinition response =
mockEndpoint.getResponseDefinition();

        MappingBuilder mappingBuilder =
withHttpMethodForUrlPattern(request.getMethod(),
resolveUrlPattern(request));
        log.info("Created MappingBuilder with HTTP method, URL
pattern");
        mappingBuilder =
mappingBuilder.atPriority(request.getPriority());
        log.info("Set priority for mapping");
        if (request.getRequestQueryParams() != null &&
!request.getRequestQueryParams().isEmpty()) {
            mappingBuilder = withQueryParams(mappingBuilder,
request.getRequestQueryParams());
            log.info("Added query parameters to mapping");
        }
        if (request.getRequestHeaders() != null &&
!request.getRequestHeaders().isEmpty()) {
            mappingBuilder = withHeaders(mappingBuilder,
request.getRequestHeaders());
            log.info("Added headers to mapping");
        }
        if (request.getRequestCookies() != null &&
!request.getRequestCookies().isEmpty()) {
            mappingBuilder = withCookies(mappingBuilder,
request.getRequestCookies());
            log.info("Added cookies to mapping");
        }
        if (request.getRequestFormParams() != null &&
!request.getRequestFormParams().isEmpty()) {
            mappingBuilder =
withRequestFormParams(mappingBuilder,
request.getRequestFormParams());
            log.info("Added form parameters to mapping");
        }
```

```
        mappingBuilder = withRequestBody(mappingBuilder,
request.getRequestBodyRules());
        mappingBuilder.willReturn(
                this.withResponse(response)
        );
        log.info("Configured response: status 200, body and
Content-Type header");
        WireMock.stubFor(mappingBuilder);
    }

    private ResponseDefinitionBuilder
withResponse(MockEndpointResponseDefinition response) {

        ResponseDefinitionBuilder responseDefinitionBuilder;
        boolean enableTemplating = false;
        List<String> transformers = new ArrayList<>();
        switch (response) {
            case DirectResponseDefinition
directResponseDefinition -> {
                enableTemplating =
directResponseDefinition.isEnableDynamicResponseTemplating();
                responseDefinitionBuilder = aResponse()

.withStatus(directResponseDefinition.getResponseHttpStatus().val
ue())

.withHeaders(mapMultiValueMapToHttpHeaders(directResponseDefinit
ion.getResponseHeaders()))
                        .withHeader("Content-Type",
directResponseDefinition.getResponseContentType().toString())

.withBody(directResponseDefinition.getResponse());
            }
            case FaultResponseDefinition faultResponseDefinition
-> {
                Fault fault =
mapFaultType(faultResponseDefinition.getFaultType());
                if (fault == null) {
                    responseDefinitionBuilder = ok();
                } else {
                    responseDefinitionBuilder = aResponse()
                            .withFault(fault);
                }
            }
            case ProxyResponseDefinition proxyResponseDefinition
-> {
                enableTemplating =
proxyResponseDefinition.isEnableTemplating();
                boolean hostnameRewriting =
proxyResponseDefinition.isEnableHostNameRewriting();
                responseDefinitionBuilder = aResponse()

.proxiedFrom(proxyResponseDefinition.getProxyUrl())
```

```
.withHeaders(mapMultiValueMapToHttpHeaders(proxyResponseDefiniti
on.getResponseHeaders()));
                if (hostnameRewriting) {
                    transformers.add("rewrite-host-header");
                }
            }
            default -> throw new
IllegalArgumentException("Unsupported response definition: " +
response);
        }

        if (enableTemplating) {
            transformers.add("response-template");
        }
        responseDefinitionBuilder =
responseDefinitionBuilder.withTransformers(transformers.toArray(
String[]::new));

        responseDefinitionBuilder = addDelayToResponse(response,
responseDefinitionBuilder);
        return responseDefinitionBuilder;
    }

    private ResponseDefinitionBuilder addDelayToResponse(
            @NotNull MockEndpointResponseDefinition response,
@NotNull ResponseDefinitionBuilder responseDefinitionBuilder
    ) {
        DelayDefinition delayDefinition =
response.getDelayDefinition();
        switch (delayDefinition) {
            case FixedDelayDefinition fixedDelayDefinition -> {
                ChronoUnit unit =
fixedDelayDefinition.getUnit();
                int delay = fixedDelayDefinition.getDelay();
                long millis = unit.getDuration().toMillis() *
delay;
                responseDefinitionBuilder =
responseDefinitionBuilder.withFixedDelay((int) millis);
            }
            case RandomUniformDelayDefinition
randomUniformDelayDefinition -> {
                int lower =
randomUniformDelayDefinition.getLowerBound();
                int upper =
randomUniformDelayDefinition.getUpperBound();
                responseDefinitionBuilder =
responseDefinitionBuilder.withUniformRandomDelay(lower, upper);
            }
            case RandomLogNormalDelayDefinition
randomLogNormalDelayDefinition -> {
                double median =
randomLogNormalDelayDefinition.getMedianDelay();
```

```
                double sigma =
randomLogNormalDelayDefinition.getStandardDeviation();
                responseDefinitionBuilder =
responseDefinitionBuilder.withLogNormalRandomDelay(median,
sigma);
            }
            case ChunkedDribbleDelayDefinition
chunkedDribbleDelayDefinition -> {
                int chunkNumber =
chunkedDribbleDelayDefinition.getChunkNumber();
                int delay =
chunkedDribbleDelayDefinition.getTotalDelay();
                responseDefinitionBuilder =
responseDefinitionBuilder.withChunkedDribbleDelay(chunkNumber,
delay);
            }
            case null, default -> {/*Does not require to do
operation*/}
        }
        return responseDefinitionBuilder;
    }

    private Fault mapFaultType(FaultType faultType) {
        return switch (faultType) {
            case NO_FAULT -> null;
            case CLOSE_SOCKET_WITH_NO_RESPONSE ->
Fault.EMPTY_RESPONSE;
            case SEND_BAD_HTTP_DATA_THEN_CLOSE_SOCKET ->
Fault.MALFORMED_RESPONSE_CHUNK;
            case
SEND_200_RESPONSE_THEN_BAD_HTTP_DATA_THEN_CLOSE_SOCKET ->
Fault.RANDOM_DATA_THEN_CLOSE;
            case PEER_CONNECTION_RESET ->
Fault.CONNECTION_RESET_BY_PEER;
        };
    }

    private HttpHeaders
mapMultiValueMapToHttpHeaders(MultiValueMap<String, String>
multiValueMap) {
        List<HttpHeader> list =
multiValueMap.asSingleValueMap().entrySet().stream()
                .map(entry -> new HttpHeader(entry.getKey(),
entry.getValue()))
                .toList();
        System.out.println("list = " + list);
        return new HttpHeaders(list);
    }


    private MappingBuilder withRequestBody(@NotNull
MappingBuilder mappingBuilder, @NotNull List<RequestBodyRule>
requestBodyRules) {
```

```
        requestBodyRules.forEach(requestBodyRule -> {
            StringValuePattern stringValuePattern =
processValidationCondition(requestBodyRule);
            mappingBuilder.withRequestBody(stringValuePattern);
        });
        return mappingBuilder;
    }

    private StringValuePattern
processValidationCondition(@NotNull RequestBodyRule
requestBodyRule) {
        switch (requestBodyRule) {
            case ValueOnlyRequestBodyRule rule -> {
                return processValueOnlyRequestBodyRule(rule);
            }
            case EqualsXmlRequestBodyRule rule -> {
                EqualToXmlPattern equalToXmlPattern =
equalToXml(rule.getValue(),
rule.getIsEnabledXmlUnitPlaceholders());
                return rule.getIsNegated() ?
not(equalToXmlPattern) : equalToXmlPattern;
            }
            case EqualsJsonRequestBodyRule rule -> {
                StringValuePattern equalToJsonPattern =
equalToJson(rule.getValue(), rule.getIsIgnoredArrayOrder(),
rule.getIsIgnoredExtraElements());
                return rule.getIsNegated() ?
not(equalToJsonPattern) : equalToJsonPattern;
            }
            case PresenceRequestBodyRule rule -> {
                return rule.getIsNegated() ? absent() :
not(absent());
            }
            case MatchesJsonPathRequestBodyRule rule -> {
                return processMatchesPathRequestBodyRule(rule,
WireMock::matchingJsonPath);
            }
            case MatchesXPathRequestBodyRule rule -> {
                return processMatchesPathRequestBodyRule(rule,
WireMock::matchingXPath);
            }
            default -> throw new
IllegalArgumentException("Unsupported validation rule: " +
requestBodyRule);
        }
    }

    private StringValuePattern
processMatchesPathRequestBodyRule(
            MatchesPathRequestBodyRule rule,
            BiFunction<String, StringValuePattern,
StringValuePattern> ruleMapper
    ) {
```

```
        String name = rule.getName();
        BodyMatchesJsonPathOrXPathLogicalConditionType
innerConditionType = rule.getInnerConditionType();
        boolean isInnerConditionNegated =
rule.getIsInnerConditionNegated();
        String value = rule.getValue();

        StringValuePattern innerStringValuePattern = switch
(innerConditionType) {
            case EQUALS -> equalTo(value);
            case CONTAINS -> containing(value);
            case MATCHES_REGEX -> matching(value);
            case IS_PRESENT -> not(absent());
            case BEFORE -> before(value);
            case AFTER -> after(value);
            case EQUALS_DATE_TIME -> equalToDateTime(value);
            default -> throw new
IllegalArgumentException("Unsupported inner condition type: " +
innerConditionType);
        };

        innerStringValuePattern = isInnerConditionNegated ?
not(innerStringValuePattern) : innerStringValuePattern;
        StringValuePattern result = ruleMapper.apply(name,
innerStringValuePattern);
        return rule.getIsNegated() ? not(result) : result;
    }

    private StringValuePattern
processValueOnlyRequestBodyRule(ValueOnlyRequestBodyRule rule) {
        StringValuePattern stringValuePattern = switch
(rule.getConditionType()) {
            case EQUALS -> equalTo(rule.getValue());
            case CONTAINS -> containing(rule.getValue());
            case MATCHES_REGEX -> matching(rule.getValue());
            case MATCHES_JSON_SCHEMA ->
matchingJsonSchema(rule.getValue());
            default -> throw new
IllegalArgumentException("Unsupported condition type: " +
rule.getConditionType());
        };
        return rule.getIsNegated() ? not(stringValuePattern) :
stringValuePattern;
    }

    private MappingBuilder withQueryParams(@NotNull
MappingBuilder mappingBuilder, @NotNull List<RequestQueryParam>
requestQueryParams) {
        return withRequestProperties(mappingBuilder,
requestQueryParams, mappingBuilder::withQueryParam);
    }

    private MappingBuilder withHeaders(@NotNull MappingBuilder
```

```
mappingBuilder, @NotNull List<RequestHeader> requestHeaders) {
        return withRequestProperties(mappingBuilder,
requestHeaders, mappingBuilder::withHeader);
    }

    private MappingBuilder withCookies(@NotNull MappingBuilder
mappingBuilder, @NotNull List<RequestCookie> requestCookies) {
        requestCookies.forEach(requestProperty -> {
            String name = requestProperty.getName();
            Condition condition =
requestProperty.getCondition();
            StringValuePattern stringValuePattern =
mapConditionWithoutMultiValuePattern(condition);
            mappingBuilder.withCookie(name, stringValuePattern);
        });
        return mappingBuilder;
    }

    private MappingBuilder withRequestFormParams(@NotNull
MappingBuilder mappingBuilder, @NotNull List<RequestFormParam>
requestFormParams) {
        return withRequestProperties(mappingBuilder,
requestFormParams, mappingBuilder::withFormParam);
    }

    private <T extends RequestProperty> MappingBuilder
withRequestProperties(
            @NotNull MappingBuilder mappingBuilder,
            @NotNull List<T> requestProperties,
            BiFunction<String, MultiValuePattern,
MappingBuilder> mappingFunction
    ) {
        requestProperties.forEach(requestProperty -> {
            String name = requestProperty.getName();
            Condition condition =
requestProperty.getCondition();
            MultiValuePattern multiValuePattern =
mapCondition(condition);
            mappingFunction.apply(name, multiValuePattern);
        });
        return mappingBuilder;

    }

    private MultiValuePattern mapCondition(@NotNull Condition
condition) {
        MultiValuePattern result;
        switch (condition) {
            case SimpleCondition simpleCondition -> result =
MultiValuePattern.of(mapSimpleCondition(simpleCondition));
            case CompositeCondition compositeCondition -> result
= mapCompositeCondition(compositeCondition);
            case PresenceCondition presenceCondition ->
```

```
                    result =
MultiValuePattern.of(mapPresenceCondition(presenceCondition));
            default -> throw new
IllegalArgumentException(UNSUPPORTED_CONDITION.formatted(conditi
on));
        }
        return result;
    }

    private StringValuePattern mapSimpleCondition(@NotNull
SimpleCondition simpleCondition) {
        String value = simpleCondition.getValue();
        StringValuePattern result = switch
(simpleCondition.getRequestConditionType()) {
            case EQUALS -> equalTo(value);
            case MATCHES_REGEX -> matching(value);
            case CONTAINS -> containing(value);
            case MATCHES_JSON_SCHEMA ->
matchingJsonSchema(value);
            case EQUALS_XML -> equalToXml(value);
            case MATCHES_XPATH -> matchingXPath(value);
            case EQUALS_JSON -> equalToJson(value);
            case MATCHES_JSON_PATH -> matchingJsonPath(value);
            case BEFORE -> before(value);
            case AFTER -> after(value);
            case EQUALS_DATE_TIME -> equalToDateTime(value);
            default -> throw
ExceptionUtils.illegalArgument(UNSUPPORTED_CONDITION.formatted(s
impleCondition));
        };
        return simpleCondition.isNegated() ? not(result) :
result;
    }

    private MultiValuePattern mapCompositeCondition(@NotNull
CompositeCondition compositeCondition) {
        boolean isNegated = compositeCondition.isNegated();

        if (compositeCondition.getSubConditions() == null) {
            throw ExceptionUtils.illegalArgument("Composite
condition without sub conditions: " + compositeCondition);
        }

        StringValuePattern[] stringValuePatterns =
compositeCondition.getSubConditions().stream()
                .map(this::mapConditionWithoutMultiValuePattern)
                .toArray(StringValuePattern[]::new);

        MultiValuePattern result = switch
(compositeCondition.getRequestConditionType()) {
            case AND -> {
                StringValuePattern and =
and(stringValuePatterns);
```

```
                and = isNegated ? not(and) : and;
                yield MultiValuePattern.of(and);
            }
            case OR -> {
                StringValuePattern or = or(stringValuePatterns);
                or = isNegated ? not(or) : or;
                yield MultiValuePattern.of(or);
            }
            case VALUES_INCLUDE -> {
                if (isNegated) {
                    for (int i = 0; i <
stringValuePatterns.length; i++) {
                        stringValuePatterns[i] =
not(stringValuePatterns[i]);
                    }
                }
                yield including(stringValuePatterns);
            }
            case VALUES_ARE_EXACTLY -> {
                if (isNegated) {
                    for (int i = 0; i <
stringValuePatterns.length; i++) {
                        stringValuePatterns[i] =
not(stringValuePatterns[i]);
                    }
                }
                yield havingExactly(stringValuePatterns);
            }
            default -> throw new
IllegalArgumentException(UNSUPPORTED_CONDITION.formatted(composi
teCondition));
        };
        log.info("Mapped composite condition: {}", result);
        return result;
    }

    private StringValuePattern
mapCompositeConditionWithoutMultiValuePattern(@NotNull
CompositeCondition condition) {
        List<Condition> subConditions =
condition.getSubConditions();
        if (subConditions == null) {
            throw ExceptionUtils.illegalArgument("Composite
condition without sub conditions: " + condition);
        }

        subConditions.stream()
                .filter(CompositeCondition.class::isInstance)
                .map(CompositeCondition.class::cast)
                .filter(subCondition ->
                        subCondition.getRequestConditionType()
== CompositeConditionLogicalType.VALUES_INCLUDE ||
```

```
subCondition.getRequestConditionType() ==
CompositeConditionLogicalType.VALUES_ARE_EXACTLY
                )
                .findAny()
                .ifPresent(
                        subCondition -> {
                            throw
ExceptionUtils.illegalArgument("Composite condition type " +
subCondition.getRequestConditionType() + " is not supported for
nested values.");
                        }
                );

        StringValuePattern[] stringValuePatterns =
subConditions.stream()
                .map(this::mapConditionWithoutMultiValuePattern)
                .toArray(StringValuePattern[]::new);

        StringValuePattern result = switch
(condition.getRequestConditionType()) {
            case AND -> and(stringValuePatterns);
            case OR -> or(stringValuePatterns);
            default -> throw new
IllegalArgumentException(UNSUPPORTED_CONDITION.formatted(conditi
on));
        };
        return condition.isNegated() ? not(result) : result;
    }

    private StringValuePattern
mapConditionWithoutMultiValuePattern(@NotNull Condition
condition) {
        return switch (condition) {
            case SimpleCondition simpleCondition ->
mapSimpleCondition(simpleCondition);
            case CompositeCondition compositeCondition ->

mapCompositeConditionWithoutMultiValuePattern(compositeCondition
);
            case PresenceCondition presenceCondition ->
mapPresenceCondition(presenceCondition);
            case null, default -> throw
ExceptionUtils.illegalArgument(UNSUPPORTED_CONDITION.formatted(c
ondition));
        };
    }

    private StringValuePattern
mapPresenceCondition(PresenceCondition presenceCondition) {
        StringValuePattern result = switch
(presenceCondition.getRequestConditionType()) {
            case IS_PRESENT -> not(absent());
            default -> throw new
```

```
IllegalArgumentException(UNSUPPORTED_CONDITION.formatted(presenc
eCondition));
        };
        return presenceCondition.isNegated() ? not(result) :
result;
    }

    public UrlPattern
resolveUrlPattern(MockEndpointRequestDefinition request) {
        UrlMatchingType urlMatchingType =
request.getUrlMatchingType();
        UrlPattern urlPattern = switch (urlMatchingType) {
            case PATH -> urlPathEqualTo(request.getPath());
            case PATH_AND_QUERY ->
urlEqualTo(request.getPath());
            case PATH_AND_QUERY_REGEX ->
urlMatching(request.getPath());
            case PATH_REGEX ->
urlPathMatching(request.getPath());
            case PATH_TEMPLATE ->
urlPathTemplate(request.getPath());
            case ANY_URL -> anyUrl();
            default -> throw new
IllegalArgumentException("Unsupported URL matching type: " +
urlMatchingType);
        };
        log.info("Resolved URL pattern: {}", urlPattern);
        log.info("Path {}", request.getPath());
        return urlPattern;
    }

    public MappingBuilder
withHttpMethodForUrlPattern(HttpMethodEnum httpMethod,
UrlPattern urlPattern) {
        return switch (httpMethod) {
            case GET -> WireMock.get(urlPattern);
            case POST -> WireMock.post(urlPattern);
            case PUT -> WireMock.put(urlPattern);
            case PATCH -> WireMock.patch(urlPattern);
            case DELETE -> WireMock.delete(urlPattern);
            case OPTIONS -> WireMock.options(urlPattern);
            case TRACE -> WireMock.trace(urlPattern);
            case ANY -> WireMock.any(urlPattern);
            default -> throw new
IllegalArgumentException("Unsupported HTTP method: " +
httpMethod);
        };
    }
}
```