ДЕРЖАВНИЙ УНІВЕРСИТЕТ ТЕЛЕКОМУНІКАЦІЙ

НАВЧАЛЬНО-НАУКОВИЙ ІНСТИТУТ ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ

Кафедра інженерії програмного забезпечення автоматизованих систем

Пояснювальна записка

до магістерської роботи
на ступінь вищої освіти магістр

на тему: «**РОЗРОБКА МОДЕЛІ БІБЛІОТЕКИ ENDPOINT RESPONSE MOCKING ТА РЕАЛІЗАЦІЯ МОВОЮ ПРОГРАМУВАННЯ JAVA»**

Виконав: студент 6 курсу, групи ІСДМ-61
спеціальності 126 Інформаційні системи та технології
освітня програма «Інформаційні системи та технології»
(шифр і назва спеціальності)
Алтинніков Д.Є.
(прізвище та ініціали)
Керівник Тушич А.М.
(прізвище та ініціали)
Рецензент_____
(прізвище та ініціали)
Нормоконтроль_____
(прізвище та ініціали)

Київ – 2021

НАВЧАЛЬНО-НАУКОВИЙ ІНСТИТУТ ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ

Кафедра Інженерії програмного забезпечення автоматизованих систем
Ступінь вищої освіти - «Магістр»
Спеціальність підготовки 126 Інформаційні системи та технології
Освітня програма «Інформаційні системи та технології»

ЗАТВЕРДЖУЮ
Завідувач кафедри ІСТ
_____ К.П.Сторчак
"____" _____2021 року

З А В Д А Н Н Я
НА МАГІСТЕРСЬКУ РОБОТУ СТУДЕНТУ

Алтинніков Дмитро Євгенович
(прізвище, ім'я, по батькові)

1.Тема роботи: «Розробка моделі бібліотеки endpoint response mocking та реалізація мовою програмування java»
Керівник роботи: Тушич Аліна Миколаївна, доктор філософії.
(прізвище, ім'я, по батькові, науковий ступінь, вчене звання)
затверджені наказом вищого навчального закладу від _____року №
2. Строк подання студентом роботи _____
3. Вхідні дані до роботи :
1. Науково-технічна література
2. Існуючі інструменти розроблені на Java
3. Готові інструменти для роботи з запита зі Spring Framework

4. Зміст розрахунково-пояснювальної записки (перелік питань, які потрібно розробити):
1. Аналіз протоколів передачі даних для mocking бібліотеки
2. Розробка responce mocking бібліотеки

5. Перелік графічного матеріалу
1. Титульний слайд
2. Постановка завдання
3. Проект та його пакети
6. Дата видачі завдання _____

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів бакалаврської роботи	Строк виконання етапів роботи	Приміт ка
1	Підбір науково-технічної літератури		
2	Вивчення матеріалів для подальшої взаємодії з ними		
3	Огляд протоколів передачі даних		
4	Визначення технічного завдання		
5	Розробка сценарію взаємодії		
6	Розробка бібліотеки		
7	Вступ, висновки, реферат		
8	Розробка демонстраційних матеріалів		
9	Попередній захист роботи		

Студент_____ Алтинніков Д.Є.
(підпис) (прізвище та ініціали)

Керівник роботи _____ Тушич А.М..
(підпис) (прізвище та ініціали)

РЕФЕРАТ

Текстова частина бакалаврської роботи 68 с., 67 рис., 20 джерела.

Об'єкт дослідження: процес створення бібліотеки мовою програмування Java.

Предмет дослідження: підхід підміни REST, SOAP запиту до певного сервісу.

Мета роботи: дослідити процес створення бібліотеки мовою програмування Java та створити прототип бібліотеки, ціллю якої є підміна відповіді на REST, SOAP запит до певного сервісу.

Методи дослідження: методи теорії інформації, методи наукового моделювання, методи дослідження інформаційних систем.

У даній магістерській роботі розроблено бібліотеку підміни запитів мовою програмування Java. Проаналізовано принципи роботи протоколів передачі даних.

Проведено огляд частини сучасного фреймворку для розробки веб-додатків.

На базі отриманих даних було розроблено бібліотеку, яка надає можливість контролювати дані зі сторонніх сервісів під час розробки певної функціональності.

Система розроблена відповідно до архітектурних вимог і відповідає сучасним найкращим практикам.

Галузь використання: Java додатки, розроблені з використанням Spring Framework, Java, Spring Framework, REST, SOAP

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ

API	прикладний програмний інтерфейс
RESTful	передача репрезентативного стану
REST	Representational State Transfer
HTTP	HyperText Transfer Protocol
SOAP	Simple Object Access Protocol
FTP	File Transfer Protocol
XML	Extensible Markup Language
HTML	HyperText Markup Language
CSS	Cascading Style Sheets
URL	Uniform Resource Locator

ЗМІСТ

Вступ 9

1 АНАЛІЗ ПРОТОКОЛІВ ПЕРЕДАЧІ ДАННИХ ДЛЯ MOCKING БІБЛІОТЕКИ 11

1.1 Огляд REST концепції через HTTP протокол 11

1.2 Огляд SOAP протоколу 24

2 РОЗРОБКА RESPONCE MOCKING БІБЛІОТЕКИ 35

2.1 Визначення технічного завдання 35

2.2 Опис сценарію взаємодії користувача з бібліотекою 42

2.3 Огляд концепції бінів у Spring Framework 43

2.4 Визначення структури проекту 52

2.5 Розробка службового сервісу 54

2.6 Розробка моделей 55

2.7 Розробка сервісів 58

2.8 Розробка інтерсепторів 66

2.9 Огляд принципу роботи Spring Controller 71

2.10 Розробка контролера 73

ВИСНОВКИ 78

ПЕРЕЛІК ПОСИЛАНЬ 80

ДЕМОНСТРАЦІЙНІ МАТЕРІАЛИ (Презентація) 81

Вступ

В сучасних умовах все більше сторонніх сервісів використовуються при розробці програмних продуктів. І як правило впливати на них не має можливості, що стає справжнім викликом, коли один з таких сервісів перестаю працювати через можливі власні проблеми зі стабільність, із сервером або деякі інші технічні проблеми. Виникнення таких збоїв зупиняє розробку продуктів, що спираються на такі сервіси. І в результаті ціли команди розробників заблоковані несправністю на яку вони не можуть вплинути.

Окрім технічних негараздів існує проблема контролю даних у сторонньому сервісі. При використанні таких сервісів вони віддають ті дані, які на даний момент часу вони мають, але не завжди при розробці певного продукти необхідні актуальні дані. Наприклад, є десять сценаріїв роботи стороннього сервісу і на ці всі сценарії розроблюваний програмний продукт має мати відповідну поведінку. І виходить, що на певний момент часу розробник може покрити один сценарій, а інші дев’ять залишать не перевіреними та не покритими необхідною логікою. І виходить, що розробник знову заблокований на невизначений проміжок часу через проблему на яку знову немає певного рішення окрім очікування.

Також при розробці певного програмного продукту майже на кожному проекті існують автоматизовані тести, які перевірюють всі сценарії роботи продукту в автоматичному режимі. Але знову з’являється проблема відсутності необхідних даних для покриття всі сценаріїв роботи продукту в певний проміжок часу, що змушує автоматизовані тести провалювати під час свого виконання, або вимагає від розробників вибіркового їх запуску, що з точки зору призначення автоматичних тестів зовсім безглуздо.

Об’єкт дослідження є процес створення бібліотеки мовою програмування Java.

Ця бібліотека в межах мови Java модуль є комплексним програмним рішення головною задачею якого – є надання інструменту контролю відповідей від сторонніх сервісів.

Предмет дослідження є підхід підміни REST, SOAP запиту до певного сервісу.

В результаті виконаної роботи мають бути проаналізовані основні підходи надання доступу до сторонніх сервісів, таких як REST, SOAP, та створення бібліотеки мовою програмування Java, вхідними даними для якої буде початкова конфігурація, а результат роботи перехоплена та змінена відповідь стороннього сервісу.

Роботи виконувалась по аналогії розробки Java модуля з використанням Spring Framework. Тобто спочатку були проаналізовані підходи з якими доведеться працювати та проаналізовані Spring інтерфейси для роботи з цими підходами

Наукова новизна цього дослідження полягає у: розробці готового модулю підміни відповіді на REST, SOAP запит на сторонній сервіс.

Результат дослідження можна використовувати, як приклад готового програмного модулю, який надає розробникам програмний інтерфейс для підміни відповіді на запит.

1 АНАЛІЗ ПРОТОКОЛІВ ПЕРЕДАЧІ ДАННИХ ДЛЯ MOCKING БІБЛІОТЕКИ

1.1 Огляд REST концепції через HTTP протокол

Перед тип як братися безпосередньо за написання, необхідно проаналізувати технології з якими доведеться зіштовхнутися. Часто терміни REST і HTTP використовуються як взаємозамінні. Я би хотів розглянути ці два терміни і визначити, що насправді означає кожен термін і чому це дві різні речі.

Hypertext Transfer Protocol забезпечує стандарт мережевого протоколу, який браузери та сервери використовують для спілкування. Під час відвідування веб-сайту ви бачите протокол HTTP, оскільки протокол відображається в URL -адресі (наприклад, http://www.google.com).

Цей протокол схожий на інші, протоколи передачі файлів, оскільки він використовується клієнтською програмою для запиту файлів з віддаленого сервера. У випадку HTTP веб-браузер запитує HTML-файли з веб-сервера, які потім відображаються у браузері з текстом, зображеннями, гіперпосиланнями та пов’язаними ресурсами.

Оскільки браузери спілкуються за допомогою HTTP, зазвичай ви можете видалити протокол із URL-адреси, коли введете його в адресний рядок браузера.

Історичний аспект HTTP пов’язаний з такою персоною як Тім Бернерс-Лі, який створив початковий стандарт HTTP на початку 1990-х років у рамках своєї роботи над визначенням оригінальної Всесвітньої павутини.

Три основні версії були розгорнуті протягом 1990-х років:

- HTTP 0.9: Підтримка основних гіпертекстових документів.
- HTTP 1.0: Розширення для підтримки розширених веб-сайтів.

- HTTP 1.1: Розроблено для усунення обмежень продуктивності HTTP 1.0, зазначених у Internet RFC 2068.

Остання версія HTTP 2.0 стала затвердженим стандартом у 2015 році. Вона підтримує зворотну сумісність з HTTP 1.1, але пропонує додаткові покращення продуктивності.

Хоча стандартний HTTP не шифрує трафік, що надсилається через мережу, стандарт HTTPS додає шифрування до HTTP за допомогою рівня Secure Sockets Layer або, пізніше, Transport Layer Security.

Як працює HTTP? HTTP — це протокол прикладного рівня, побудований на основі TCP, який використовує модель зв'язку клієнт-сервер. Клієнти і сервери HTTP спілкуються за допомогою повідомлень запитів і відповідей. Існують три основні типи повідомлень HTTP: GET, POST і HEAD.

HTTP GET повідомлення, надіслані на сервер, містять лише URL-адресу. У кінці URL-адреси може бути додано нуль або більше необов’язкових параметрів. Сервер обробляє необов’язкову частину URL-адреси, якщо вона є, і повертає результат ,веб-сторінку або елемент веб-сторінки, браузеру або певний ресурс у вигляді певних даних.

HTTP POST повідомлення розміщують будь-які додаткові параметри даних у тілі повідомлення запиту, а не додають їх у кінець URL-адреси.

HTTP HEAD: Запити працюють так само, як і запити GET. Замість відповіді з повним вмістом URL-адреси сервер надсилає назад лише інформацію заголовка, яка міститься всередині розділу HTML.

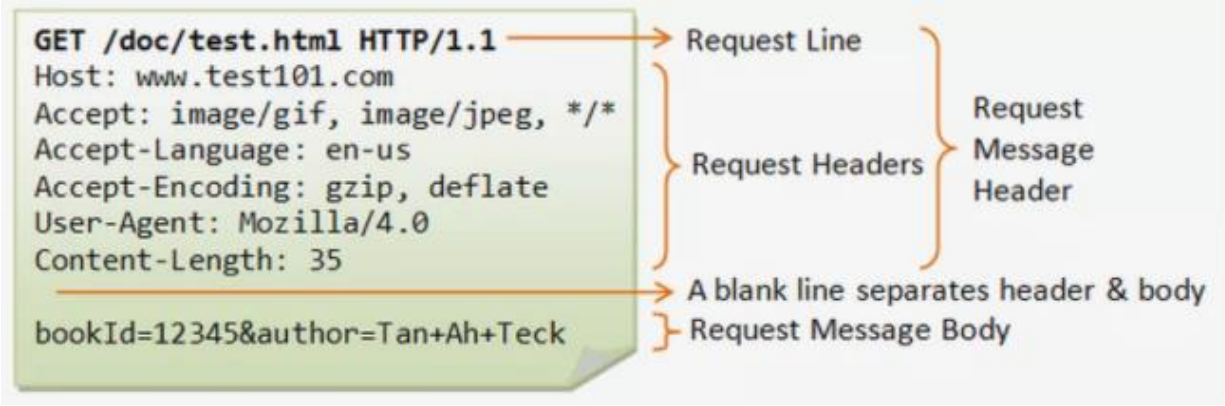


Рисунок 1.1 – структура HTTP запиту

Браузер ініціює зв'язок із HTTP сервером, ініціюючи TCP-з'єднання з сервером. Сеанси веб-перегляду за замовчуванням використовують порт сервера 80, хоча замість них іноді використовуються інші порти, наприклад 8080. Після встановлення сеансу ви запускаєте надсилання та отримання повідомлень HTTP, відвідуючи веб - сторінку. HTTP - це те, що називається системою без стану. Це означає, що, на відміну від інших протоколів передачі файлів, таких як FTP, з’єднання HTTP розривається після завершення запиту. Отже, після того, як веб-браузер надсилає запит і сервер відповість, з'єднання розривається. Кожен окремий запит надсилається на сервер, який обробляє його та надає відповідь, яка називається відповіддю. Між клієнтом і сервером є численні сутності, які разом називаються проксі, які виконують різні операції і діють, наприклад, як шлюзи або кеші. Насправді між браузером та сервером, що обробляє запит, є більше комп’ютерів: є маршрутизатори, модеми тощо. Завдяки багатошаровому дизайну Інтернету вони приховані в мережевих та транспортних рівнях. HTTP знаходиться зверху, на рівні додатка.

Також досить часто згадувалось таке поняття як клієнт або іншими словами користувач-агент. Користувача-агент— це будь-який інструмент, який діє від імені

користувача. Цю роль насамперед виконує веб-браузер, але її також можуть виконувати програми, які використовуються інженерами та веб-розробниками для налагодження їхніх програм. Браузер завжди є суб’єктом, який ініціює запит. Але іноді це сервер або деякий бекенд додаток. Щоб відобразити веб-сторінку, браузер надсилає оригінальний запит на отримання документа HTML, що представляє сторінку. Потім він аналізує цей файл, роблячи додаткові запити, що відповідають сценаріям виконання, інформації про макет (CSS) для відображення). Потім веб-браузер об’єднує ці ресурси, щоб представити повний документ, веб-сторінку. Сценарії, які виконуються браузером, можуть отримати додаткові ресурси на пізніх етапах, і браузер відповідно оновить веб-сторінку. Веб-сторінка - це гіпертекстовий документ. Це означає, що деякі частини відображеного вмісту є посиланнями, які можна активувати, зазвичай клацанням миші, щоб отримати нову веб-сторінку, дозволяючи користувачеві спрямовувати свого агента користувача та переміщатися по Інтернету. Браузер перетворює ці вказівки на HTTP запити та додатково інтерпретує відповіді HTTP, щоб представити користувачеві чітку відповідь.

На протилежному боці каналу зв'язку знаходиться сервер, який обслуговує документ за запитом клієнта. Сервер виглядає практично лише як одна машина; але насправді це може бути сукупність серверів, що розподіляють навантаження, балансують навантаження, або складне програмне забезпечення, яке опитує інші комп’ютери, наприклад, кеш, сервер БД або сервери електронної комерції, повністю або частково створюючи документ на вимогу. Сервер - це не обов’язково одна машина, але на одній машині може розміщуватися кілька екземплярів програмного забезпечення сервера. За допомогою протоколу HTTP/1.1 та заголовка Host вони навіть можуть мати спільну IP -адресу.

Між веб-браузером і сервером численні комп'ютери та машини передають HTTP-повідомлення. Завдяки багатошаровій структурі веб-стека більшість із них працює на транспортному, мережевому або фізичному рівнях, стаючи прозорими на рівні HTTP

і потенційно впливаючи на продуктивність. Ті, що працюють на прикладних рівнях, зазвичай називаються проксі. Вони можуть бути прозорими, пересилати отримані запити, не змінюючи їх у будь-який спосіб, або непрозорими, в цьому випадку вони певним чином змінять запит, перш ніж передавати його на сервер. Проксі можуть виконувати безліч функцій:

- кешування (кеш може бути загальнодоступним або приватним, як кеш браузера)
- фільтрація (наприклад, антивірусне сканування або батьківський контроль)
- балансування навантаження (щоб дозволити кільком серверам обслуговувати різні запити)
- аутентифікація (для контролю доступу до різних ресурсів)
- ведення журналу (дозволяє зберігати історичну інформацію)

Основні аспекти HTTP, які роблять його затребуваним:

- HTTP простий. Як правило, протокол HTTP є простим і читаним людиною, навіть з додатковою складністю, введеною в протоколі HTTP/2 шляхом інкапсуляції повідомлень HTTP у фрейми. Повідомлення HTTP можуть бути прочитані та зрозумілі людьми, що полегшує тестування для розробників та зменшує складність для новачків.
- HTTP є розширюваним. Введені в протоколі HTTP/1.0 заголовки HTTP полегшують розширення та експериментування з цим протоколом. Нову функціональність можна ввести навіть шляхом простої угоди між клієнтом і сервером про семантику нового заголовка.
- HTTP не має статусу, але не є безсесійним. HTTP не має стану: немає зв’язку між двома запитами, які послідовно виконуються в одному з’єднанні. Це відразу ж може стати проблематичним для користувачів, які намагаються взаємодіяти з

певними сторінками послідовно, наприклад, використовуючи кошики для покупок електронної комерції. Але в той час як ядро самого HTTP не має статусу, файли cookie HTTP дозволяють використовувати сеанси зі збереженням стану. За допомогою розширення заголовка файли cookie HTTP додаються до робочого процесу, дозволяючи створювати сеанси для кожного запиту HTTP для спільного використання одного контексту або того самого стану.

- HTTP і з'єднання. З'єднання контролюється на транспортному рівні, а тому принципово поза сферою використання HTTP. HTTP не вимагає, щоб базовий транспортний протокол був заснований на з'єднанні; він вимагає лише того, щоб він був надійним або не втрачав повідомлень, як мінімум, у таких випадках видавав би помилку. Серед двох найпоширеніших транспортних протоколів в Інтернеті TCP є надійним, а UDP - ні. Тому HTTP покладається на стандарт TCP, який базується на підключенні. Перш ніж клієнт і сервер можуть обмінятися парою запит/відповідь HTTP, вони повинні встановити TCP-з'єднання, процес, який вимагає кількох зворотних звернень. Поведінка HTTP/1.0 за замовчуванням полягає у відкритті окремого TCP-з'єднання для кожної пари запит/відповідь HTTP. Це менш ефективно, ніж спільний доступ до одного з'єднання TCP, коли кілька запитів надсилаються послідовно. Щоб пом'якшити цю ваду, HTTP/1.1 запровадив конвеєризацію. HTTP/2 пішов ще далі, мультиплексувавши повідомлення через одне з’єднання, допомагаючи підтримувати з’єднання теплим і ефективнішим. Проводяться експерименти з розробки кращого транспортного протоколу, більш пристосованого до HTTP. Наприклад, Google експериментує з QUIC, який базується на UDP, щоб забезпечити більш надійний та ефективний транспортний протокол.

Ця розширювана природа HTTP з часом дозволила збільшити контроль і функціональність Інтернету. Кеш і методи автентифікації були функціями, які

оброблялися на початку історії HTTP. Здатність послабити обмеження походження, навпаки, була додана лише в 2010-х роках. Ось список поширених функцій, якими можна керувати за допомогою HTTP:

- Кешування Кешування документів можна контролювати за допомогою HTTP. Сервер може проінформувати проксі та клієнтів про те, що кешувати і як довго. Клієнт може доручити проксі-серверам проміжного кешу ігнорувати збережений документ.

- Розслаблення обмеження походження Щоб запобігти відслідковуванню та іншим вторгненням у конфіденційність, веб-браузери забезпечують суворе розділення між веб-сайтами. Тільки сторінки з одного походження можуть отримати доступ до всієї інформації веб-сторінки. Незважаючи на те, що таке обмеження є тягарем для сервера, заголовки HTTP можуть послабити це суворе розділення на стороні сервера, дозволяючи документу стати клаптиком інформації, отриманої з різних доменів; для цього можуть бути навіть причини пов'язані з безпекою.

- Аутентифікація Деякі сторінки можуть бути захищені, щоб отримати до них доступ могли лише певні користувачі. Базову автентифікацію можна надавати за допомогою HTTP, використовуючи заголовки WWW-Authenticate та подібні, або шляхом встановлення конкретного сеансу за допомогою файлів cookie HTTP.

- Проксі та тунельні сервери або клієнти часто знаходяться в інтрамережах і приховують свою справжню IP-адресу від інших комп’ютерів. Потім HTTP-запити проходять через проксі, щоб подолати цей мережевий бар’єр. Не всі проксі-сервери є проксі-серверами HTTP. Наприклад, протокол SOCKS працює на нижчому рівні. Ці проксі можуть обробляти інші протоколи, наприклад FTP.

- Сеанси Використання файлів cookie HTTP дозволяє зв’язувати запити зі станом сервера. Це створює сеанси, незважаючи на те, що базовий HTTP є протоколом без стану. Це корисно не лише для кошиків для покупок електронної комерції, а й для будь-якого сайту, що дозволяє користувачеві налаштувати вихідні дані.

Коли клієнт хоче спілкуватися з сервером, кінцевим сервером або проміжним проксі-сервером, він виконує такі дії:

1. Відкриття TCP-з'єднання: TCP-з'єднання використовується для надсилання запиту або кількох запитів та отримання відповіді. Клієнт може відкрити нове з'єднання, повторно використовувати існуюче або відкрити кілька TCP-з'єднань із серверами.
2. Надіслати повідомлення HTTP: За допомогою протоколу HTTP/2 ці прості повідомлення інкапсульовані у фрейми, що унеможливлює їх безпосереднє читання, але принцип залишається незмінним. Наприклад:

```
GET / HTTP/1.1
Host: developer.mozilla.org
Accept-Language: fr
```

Рисунок 1.2 – фрейм протоколу HTTP/2

3. Прочитайте відповідь, надіслану сервером, наприклад:

```
HTTP/1.1 200 OK
Date: Sat, 09 Oct 2010 14:28:02 GMT
Server: Apache
Last-Modified: Tue, 01 Dec 2009 20:18:22 GMT
ETag: "51142bc1-7449-479b075b2891b"
Accept-Ranges: bytes
Content-Length: 29769
Content-Type: text/html
```

Рисунок 1.3 – відповідь сервера на HTTP запит

Звичайно при роботі з HTTP можуть виникати несправності. Повідомлення, передані через HTTP, можуть не працювати з кількох причин:

- Помилка користувача.
- Несправність веб-браузера або веб-сервера.
- Помилки при створенні веб -сторінок.
- Тимчасові збої в мережі.

При виникненні цих збоїв протокол фіксує причину збоїв і повідомляє браузеру код помилки під назвою рядок/код стану HTTP. Помилки починаються з певної цифри, яка вказує, що це за помилка. Наприклад, помилки з кодом помилки, що починається з чотирьох, вказують на те, що запит на сторінку неможливо виконати належним чином, або що запит містить неправильний синтаксис. Наприклад, помилка 404 означає, що веб-сторінку неможливо знайти; деякі веб-сайти навіть пропонують цікаві користувацькі сторінки з помилкою 404.

REST(Representational State Transfer) означає передача репрезентативного стану . Вперше він був описаний у відомій нині дисертації Роя Філдінга. У цій роботі Філдінг

виклав своє бачення ідеальної архітектури програмного забезпечення для всесвітньої павутини.

REST не є стандартом або специфікацією. Натомість Філдінг описав REST як архітектурний стиль для розподілених гіпермедійних систем. Він вважав, що існує кілька атрибутів архітектури RESTful, які роблять її ідеальною для великих взаємопов'язаних систем, таких як Інтернет.

(Далі ми розглянемо атрибути, які визначають системи RESTful.)

Основними будівельними блоками систем RESTful є ресурси. Ресурсом може бути, наприклад, веб-сторінка, відеопотік або зображення. Ресурс може бути навіть абстрактним поняттям, таким як список усіх користувачів у базі даних або прогноз погоди для певного місця розташування. Єдиним реальним обмеженням є те, що кожен ресурс у системі унікально ідентифікується.

Крім того, ресурси можуть бути доступні у кількох представленнях. У моделі клієнт-сервер, сервер відповідає за управління станом ресурсу, але клієнт може вибрати, з яким представленням він бажає взаємодіяти.

Уніфікований інтерфейс є одним з відмінних атрибутів систем RESTful. Він вимагає від клієнтів доступу до всіх ресурсів за допомогою того самого набору стандартних операцій. Перевагою єдиного інтерфейсу полягає у простоті створення реалізацій, не пов'язаних із наданими ними послугами. Це дозволяє розвивати послуги, не впливаючи на клієнтів. Компроміс полягає в тому, що єдиний інтерфейс може накласти непотрібні обмеження на деякі системи або вимагати, щоб сервери працювали менш ефективно, ніж вони могли б із спеціалізованими операціями.

Усі операції в системі RESTful не мають зберігати якийсь стан. Це означає, що кожен виклик від клієнта до сервера не повинен залежати від будь-якого спільного стану. Це також означає, що кожен запит до сервера повинен містити всі необхідні дані, щоб сервер виконав запит.

Вимога не зберігати стан в REST поступається місцем кільком ключовим властивостям:

- Видимість: Кожен запит можна аналізувати окремо, щоб відстежувати стан системи та швидкість реагування
- Надійність: легше відновити системні збої
- Масштабованість: Просто додати більше ресурсів сервера для обробки більшої кількості запитів

Однак також йде компроміс у тому, що запити до сервера є більшими і містять повторювані дані, коли клієнт має багато взаємодій з сервером.

На відміну від REST, протокол передачі гіпертексту (HTTP) є стандартом з чітко визначеними обмеженнями. HTTP - це протокол зв'язку, який забезпечує більшість наших повсякденних взаємодій в Інтернеті:

- Веб-браузери завантажують веб-сторінки
- Потокове передавання відео
- Використання мобільного пристрою для вимкнення освітлення вдома
    - Отже, чи REST те саме, що HTTP? Коротка відповідь - ні.

HTTP - це протокол, який підтримується Робочою групою Інтернет -інженерії. Хоча це не те саме, що REST, він демонструє багато особливостей системи RESTful. Це не випадково, оскільки Рой Філдінг був одним з перших авторів RFC для HTTP.

Важливо пам’ятати, що використання системи HTTP не є обов’язковим для системи RESTful. Так сталося, що HTTP - хороший старт, оскільки він проявляє багато якостей RESTful.

Давайте детальніше розглянемо деякі якості, які роблять HTTP протоколом RESTful.

У світі HTTP ресурсами зазвичай є файли на віддаленому сервері. Це можуть бути HTML, CSS, JavaScript, зображення та всі інші файли, що містять сучасні веб-сторінки. Кожен файл розглядається як окремий ресурс, який можна адресувати за допомогою унікальної URL-адреси.

Однак HTTP не тільки для файлів. Термін ресурс також може позначати довільні дані на віддаленому сервері: клієнтів, продукти, налаштування конфігурації та багато іншого. Ось чому HTTP став популярним для створення сучасних API. Він забезпечує послідовний та передбачуваний спосіб доступу та віддаленого керування даними.

Крім того, HTTP дозволяє клієнтам вибирати різні представлення для деяких ресурсів. У HTTP це обробляється за допомогою заголовків та різноманітних відомих типів носіїв.

Наприклад, веб-сайт про погоду може надавати представлення HTML і JSON для одного прогнозу погоди. Один підходить для відображення у веб-браузері, а інший - для обробки іншою системою, яка архівує історичні дані про погоду.

Інший спосіб, яким HTTP дотримується принципів REST, полягає в тому, що він надає однаковий набір методів для кожного ресурсу. Хоча існує майже десяток доступних методів HTTP, більшість служб використовують перш за все чотири, які відображають операції CRUD: POST, GET, PUT та DELETE.

Знання цих операцій завчасно полегшує створення клієнтів, які користуються веб-сервісом. Порівняно з протоколами, такими як SOAP, де операції можуть бути налаштовані та необмежені, HTTP зберігає відомий набір операцій мінімальним і послідовним.

Також окремі веб-служби можуть заборонити певні методи для деяких ресурсів. Або вони вимагають автентифікації для важливих ресурсів. Незважаючи на це, набір можливих методів HTTP добре відомий і не може змінюватися від одного веб-сайту до іншого.

Однак для всіх способів, якими HTTP реалізує принципи RESTful, існує кілька способів, якими він також може їх порушувати. По-перше, REST не є комунікаційним протоколом, тоді як HTTP є. Наступне, що є найбільш суперечливим є те, що більшість сучасних веб-серверів використовують файли cookie та сеанси для збереження стану. Коли вони використовуються для посилання на стан на стороні сервера, це порушує принцип не зберігання стану. Нарешті, використання URL-адрес, як визначено в IETF, може дозволити веб-серверу порушити єдиний інтерфейс. Наприклад, розглянемо таку URL -адресу:

https://www.foo.com/api/v1/customers?id=17&action=clone

(make image)

Хоча цю URL-адресу потрібно запитувати за допомогою одного із заздалегідь визначених методів HTTP, наприклад GET, вона використовує параметри запиту для надання додаткових операцій. У цьому випадку ми вказуємо дію з назвою clone, яка очевидно недоступна для всіх ресурсів у системі. Також незрозуміло, якою буде відповідь без більш детального знання послуги.

Хоча багато людей продовжують використовувати терміни REST і HTTP як взаємозамінні, правда в тому, що це різні речі. REST відноситься до набору атрибутів певного архітектурного стилю, тоді як HTTP-це чітко визначений протокол, який демонструє багато функцій системи RESTful.

1.2 Огляд SOAP протоколу

SOAP-це протокол на основі XML для доступу до веб-служб по протоколу HTTP. Він має деякі специфікації, які можна використовувати у всіх додатках. SOAP відомий як Простий протокол доступу до об'єктів, але пізніше його просто скоротили до SOAP v1.2. SOAP - це протокол або, іншими словами, це визначення того, як веб-сервіси спілкуються один з одним або спілкуються з клієнтськими програмами, які їх викликають. SOAP був розроблений як проміжна мова, щоб додатки, створені на різних мовах програмування, могли легко спілкуватися один з одним і уникнути надзвичайних зусиль у розробці.

У сучасному світі існує величезна кількість програм, побудованих на різних мовах програмування. Наприклад, може бути веб-додаток, розроблений на Java, інший у .Net та інший у PHP. Обмін даними між додатками є вирішальним у сучасному мережевому світі. Але обмін даними між цими неоднорідними додатками був би складним. Так само код буде досить складним для здійснення такого обміну даними. Одним із методів боротьби з цією складністю є використання XML (Extensible Markup Language) як проміжної мови для обміну даними між програмами. Кожна мова програмування може розуміти мову розмітки XML. Отже, XML використовувався як основний засіб обміну даними. Але не існує стандартних специфікацій щодо використання XML у всіх мовах програмування для обміну даними. Саме тут

з'являється протокол SOAP. SOAP був розроблений для роботи з XML через HTTP і мав певну специфікацію, яку можна використовувати у всіх додатках.

SOAP - це протокол, який використовується для обміну даними між програмами. Нижче наведено деякі з причин, чому використовується SOAP:

— При розробці веб-сервісів на основі SOAP вам потрібно мати певну мову, яку можна використовувати для веб-сервісів для спілкування з клієнтськими додатками. SOAP - це ідеальне середовище, розроблене для досягнення цієї мети. Цей протокол також рекомендується консорціумом W3C, який є керівним органом для всіх веб-стандартів.

— SOAP-це легкий протокол, який використовується для обміну даними між програмами. Важливо звернути увагу на ключове слово «легкий». Оскільки програмування SOAP базується на мові XML, яка сама по собі є легкою мовою обміну даними, отже, SOAP як протокол, який також належить до тієї ж категорії.

— SOAP розроблений як незалежний від платформи, а також незалежний від операційної системи. Таким чином, протокол SOAP може працювати з будь - якими програмами на основі мов програмування як на платформі Windows, так і на Linux.

— Він працює за протоколом HTTP – SOAP працює за протоколом HTTP, який є протоколом за замовчуванням, який використовується всіма веб-програмами. Отже, для роботи веб-сервісів, створених на основі протоколу SOAP, для роботи у всесвітній павутині не потрібно ніяких налаштувань.

Специфікація SOAP визначає щось відоме як «повідомлення SOAP», яке надсилається веб-службі та клієнтській програмі. Нижче наведена діаграма архітектури SOAP(рис) показує різні будівельні блоки повідомлення SOAP.

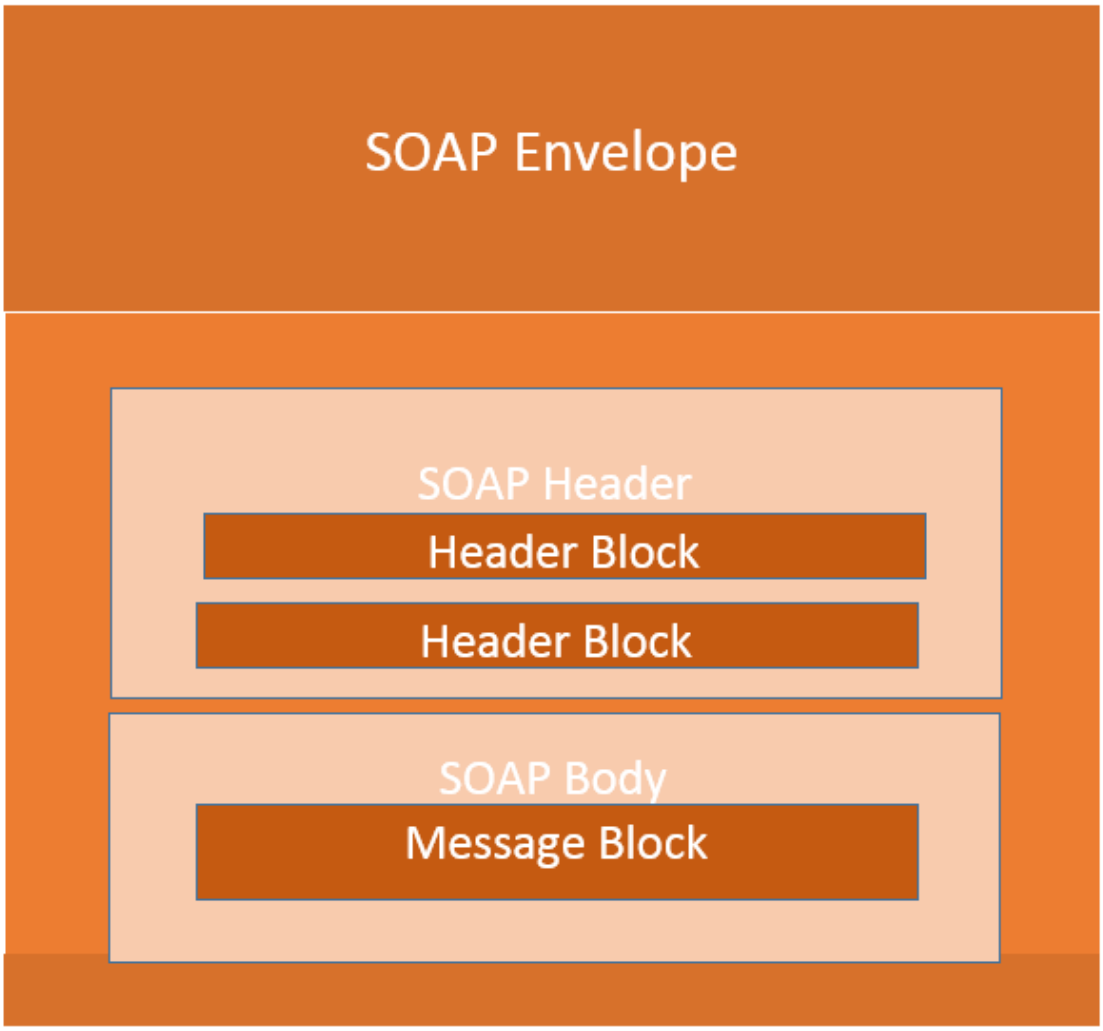


Рисунок 1.4 – SOAP конверт

Повідомлення SOAP - це XML документ, який містить наведені нижче компоненти:

— Елемент Envelope, який ідентифікує XML -документ як повідомлення SOAP - це частина, що містить повідомлення SOAP, і використовується для інкапсуляції всіх деталей у повідомленні SOAP. Це кореневий елемент у повідомленні SOAP.

— Елемент заголовка, який містить інформацію заголовка – елемент заголовка може містити таку інформацію, як облікові дані автентифікації, які можуть використовуватися програмою, що викликає. Він також може містити визначення складних типів, які можна використовувати у повідомленні SOAP. За замовчуванням повідомлення SOAP може містити параметри, які можуть бути простих типів, таких як рядки та числа, але також можуть бути складними типами об’єктів.

— Елемент Body, що містить інформацію про виклик та відповідь - Цей елемент містить фактичні дані, які необхідно надіслати між веб -службою та викликовою програмою. Нижче наведено приклад веб -служби SOAP для тіла SOAP, який насправді працює на складному типі, визначеному в розділі заголовка. Ось відповідь назви навчального посібника та опису підручника, які надсилаються до виклику програми, яка викликає цю веб -службу.(рис)

```
<soap:Body>
   <GetTutorialInfo>
               <TutorialName>Web Services</TutorialName>
               <TutorialDescription>All about web services</TutorialDescription>
   </GetTutorialInfo>
</soap:Body>
```

Рисунок 1.4 – тіло SOAP запиту

Створим в якості прокладу такий запит. Припустимо, ми хочемо надіслати структурований тип даних, який містив би комбінацію “Назва навчального посібника” та “Опис підручника”, тоді ми б визначили складний тип, як показано нижче(рис). Складний тип визначається тегом елемента <xsd:complexType>. Потім усі необхідні

елементи структури разом із відповідними типами даних визначаються у колекції складних типів.

```
<xsd:complexType>
 <xsd:sequence>
        <xsd:element name="Tutorial Name" type="string"/>
        <xsd:element name="Tutorial Description"  type="string"/>
  </xsd:sequence>
</xsd:complexType>
```

Рисунок 1.5 – комплексний тип SOAP запиту

Варто відзначити, що повідомлення SOAP зазвичай автоматично генеруються веб-службою під час її виклику. Щоразу, коли клієнтська програма викликає метод у веб-службі, веб-служба автоматично генерує повідомлення SOAP, яке містить необхідну інформацію про дані, які надсилатимутися із веб-служби до клієнтської програми. Як згадувалося раніше, просте повідомлення SOAP має такі елементи:

— Елемент Конверт

— Елемент заголовка і

— Елемент тіла

— Елемент несправності (необов’язково)

Рисунок 1.6 – елементи SOAP запиту

Важливо визначити призначення кожної частини повідомлення SOAP. Як видно з наведеного вище повідомлення SOAP(рис), перша частина повідомлення SOAP є елементом конверта, який використовується для інкапсуляції всього повідомлення SOAP. Наступним елементом є тіло SOAP, яке містить деталі фактичного повідомлення. Наше повідомлення містить веб-сервіс, який має назву «WebService». “Webservice” приймає параметр типу “int” і має назву TutorialID. Тепер наведене вище повідомлення SOAP буде передано між веб-сервісом і клієнтською програмою. Можете побачити, наскільки ця інформація корисна для клієнтської програми. Повідомлення SOAP повідомляє клієнтській програмі, яка назва веб-сервісу, а також які параметри вона очікує, а також тип кожного параметра, який приймає веб-сервіс.

Перший блок SOAP повідомлення - це конверт SOAP. Конверт SOAP використовується для інкапсуляції всіх необхідних деталей повідомлень SOAP, якими обмінюються веб-служба та клієнтська програма. Елемент конверта SOAP використовується для вказівки початку та кінця повідомлення SOAP. Це дозволяє

клієнтській програмі, яка викликає веб-службу, знати, коли SOAP-повідомлення закінчується.

На основі отриманої інформації можна відзначити наступні моменти конверта SOAP:

— Кожне повідомлення SOAP має мати кореневий елемент конверта. Абсолютно обов’язково, щоб SOAP-повідомлення містило елемент конверта.

— Кожен елемент конверта повинен мати принаймні один елемент тіла SOAP запита.

— Якщо елемент Envelope містить елемент заголовка, він повинен містити не більше одного, і він повинен відображатися як перший дочірній елемент конверта, перед елементом body.

— Конверт змінюється при зміні версій SOAP.

— Процесор SOAP, сумісний з v1.1, генерує помилку при отриманні повідомлення, що містить простір імен конверта v1.2.

— Процесор SOAP, сумісний з v1.2, генерує помилку Version Mismatch, якщо отримує повідомлення, яке не включає простір імен конверта v1.2.

Нижче на (рис) наведено приклад API SOAP версії 1.2 елемента конверта SOAP.

```
<?xml version="1.0"?>
<SOAP-ENV:Envelope xmlns:SOAP-ENV="http://www.w3.org/2001/12/soap-envelope" SOAP-
ENV:encodingStyle=" http://www.w3.org/2001/12/soap-encoding">
          <soap:Body>
        <WebService xmlns="http://tempuri.org/">
                  <TutorialID>int</TutorialID>
                </WebService>
          </soap:Body>
</SOAP-ENV:Envelope>
```

Рисунок 1.7 – SOAP конверт версії 1.2

Коли надходить запит до веб-служби SOAP, повернена відповідь може мати дві форми, які є успішною, або відповідь на помилку. Коли успіх генерується, відповідь від сервера завжди буде повідомленням SOAP. Але якщо генеруються помилки SOAP, вони повертаються як помилки “HTTP 500”.

Повідомлення про помилку SOAP складається з таких елементів:

- <faultCode>– це код, який позначає код помилки. Код несправності може мати будь -яке з наведених нижче значень:
    - SOAP-ENV:VersionMismatch – це коли зустрічається недійсний простір імен для елемента
    - SOAP Envelope. SOAP-ENV:MustUnderstand – Безпосередній дочірній елемент елемента Header з атрибутом mustUnderstand, встановленим на «1», не зрозумілий.
    - SOAP-ENV: Клієнт-повідомлення неправильно сформоване або містить неправильну інформацію.
    - SOAP-ENV: Сервер-сталася проблема з сервером, тому повідомлення не вдалося продовжити.

- <faultString> - Це текстове повідомлення, яке містить детальний опис помилки.
- <faultActor> (необов'язково) – це текстовий рядок, який вказує, хто спричинив помилку.
- <detail>(необов’язково) – це елемент для повідомлень про помилки, що стосуються конкретної програми. Таким чином, програма може мати конкретне повідомлення про помилку для різних сценаріїв бізнес-логіки.

Нижче на (рис) наведено приклад повідомлення про помилку. Помилка генерується, якщо сценарій, у якому клієнт намагається використати метод під назвою TutorialID у класі GetTutorial. Повідомлення про помилку нижче генерується у випадку, якщо метод не існує у визначеному класі.

```
<?xml version='1.0' encoding='UTF-8'?>
<SOAP-ENV:Envelope xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:xsi="http://www.w3.org/1999/XMLSchema-instance"
xmlns:xsd="http://www.w3.org/1999/XMLSchema">
      <SOAP-ENV:Body>
         <SOAP-ENV:Fault>
         <faultcode xsi:type="xsd:string">SOAP-ENV:Client</faultcode>
        <faultstring xsi:type="xsd:string">
            Failed to locate method (GetTutorialID) in class (GetTutorial)
         </faultstring>
    </SOAP-ENV:Fault>
   </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Рисунок 1.8 – SOAP конверт версії 1.2

Коли доводиться зіштовхнутися з наведеною відповіддю, її слід читати як «Не вдалося знайти метод (GetTutorialID) у класі (GetTutorial)».

Вся комунікація через SOAP здійснюється за протоколом HTTP. До SOAP багато веб-сервісів використовували стандартний стиль RPC (Remote Procedure Call) для спілкування. Це був найпростіший тип спілкування, але він мав багато обмежень. Припустимо що на сервері розміщена веб -служба, яка надала 2 методи:

- GetEmployee – це дозволить отримати всю інформацію про співробітника
- SetEmployee - Це дозволить відповідно встановити значення таких деталей, як відділ працівників, заробітна плата тощо.

У звичайному спілкуванні в стилі RPC клієнт просто викликає методи у своєму запиті і надсилає необхідні параметри на сервер, а потім сервер надсилає потрібну відповідь.

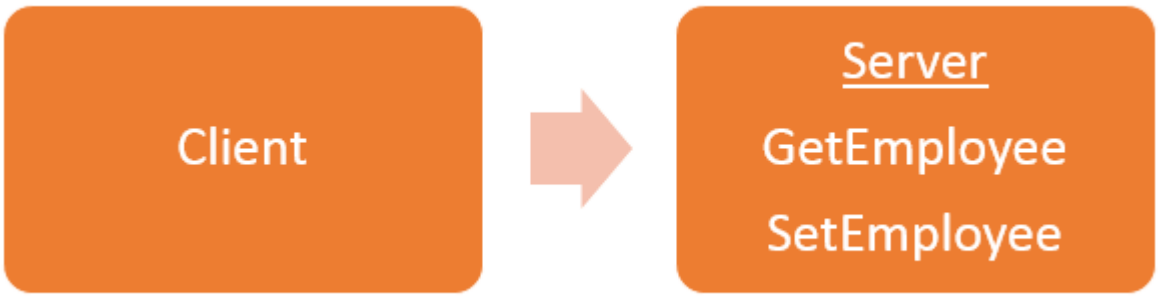


Рисунок 1.9 – RPC стиль спілкування клієнта з сервером

Наведена на (рис) вище модель комунікації має наведені нижче серйозні обмеження:

— Незалежний від мови – сервер, на якому розміщуються методи, буде певною мовою програмування, і зазвичай виклики до сервера будуть лише цією мовою програмування.

— Не стандартний протокол – коли здійснюється виклик до віддаленої процедури, виклик не виконується за стандартним протоколом. Це було б проблемою, оскільки переважно вся комунікація через Інтернет має здійснюватися за протоколом HTTP.

— Брандмауери - оскільки виклики RPC не йдуть через звичайний протокол, на сервері повинні бути відкриті окремі порти, щоб клієнт міг спілкуватися з сервером. Зазвичай усі брандмауери блокують такий вид трафіку, і, як правило, потрібно багато конфігурації, щоб забезпечити роботу такого типу зв'язку між клієнтом і сервером.

Щоб подолати всі обмеження, зазначені вище, SOAP буде використовувати наведену нижче на (рис) комунікаційну модель.

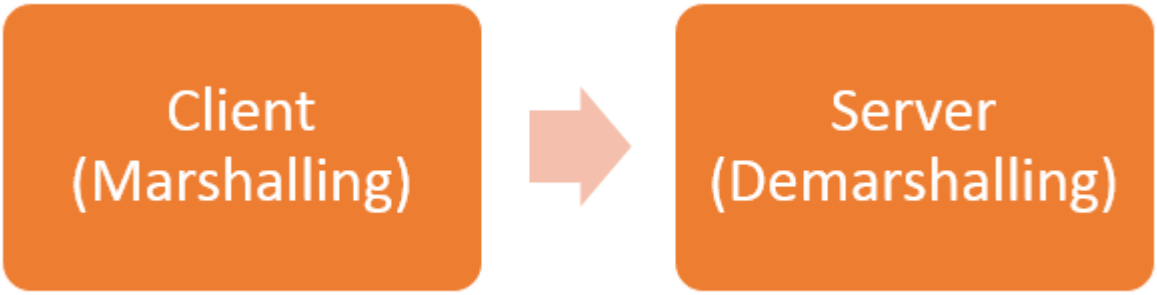


Рисунок 1.10 – комунікаційна модель SOAP протоколу

Клієнт відформатує інформацію про виклик процедури та будь -які аргументи у повідомлення SOAP і надсилає їх на сервер як частину HTTP запиту. Цей процес інкапсуляції даних у повідомлення SOAP був відомий як маршаллінг. Потім сервер розгортає повідомлення, надіслане клієнтом, дивиться, що запитує клієнт, а потім надсилає відповідну відповідь клієнту у вигляді SOAP-повідомлення. Практика розгортання запиту, надісланого клієнтом, відома як демаршалінг.

2 РОЗРОБКА RESPONCE MOCKING БІБЛІОТЕКИ

2.1 Визначення технічного завдання

Існує проблема відсутності можливості контролювати відповіді від сторонніх серверів, запити до яких надсилаються за допомогою REST або SOAP принципу.

Для вирішення цієї проблеми необхідно створити бібліотеку, яка могла надити певний інтерфейс контролю відповіді до стороннього сервісу. Дане рішення має працювати в зв'язку зі Spring Framework так як даний фреймворк являється лідером на ринку веб-додатків написаних мовою програмування Java. Також бібліотека має брати до уваги, що послідовність параметрів запиту може змінюватися із часом і цей аспект має ігноруватися для підтримко консистентності моків.

Окрім перехоплення запиту має проходити операції підміни відповіді, що має буде забезпечене сконфігурованим моком.

Також бібліотека має надавати можливість бути виключена під час виконання для можливості повернення штатного режиму роботи додатку.

Для вирішення попередньої вимоги можна надати користувачу веб-інтерфейс через, який також буде необхідно надати контроль моків.

Бібліотека має працювати з Spring HTTP-клієнтом, а саме RestTemplate.

Станом на Spring Framework 5 поряд зі стеком WebFlux Spring представила новий HTTP-клієнт під назвою WebClient. WebClient - це сучасний, альтернативний HTTP-клієнт RestTemplate. Він не тільки забезпечує традиційний синхронний API, але й підтримує ефективний неблокувальний та асинхронний підхід.

Почнемо з простого і поговоримо про запити GET із коротким прикладом використання API getForEntity () на рисунку 2.1

```
RestTemplate restTemplate = new RestTemplate();
String fooResourceUrl
  = "http://localhost:8080/spring-rest/foos";
ResponseEntity<String> response
  = restTemplate.getForEntity(fooResourceUrl + "/1", String.class);
assertThat(response.getStatusCode(), equalTo(HttpStatus.OK));
```

Рисунок 2.1 – GET запит створений за допомогою RestTemplate

Важливо звернути увагу, що у нас є повний доступ до відповіді HTTP, тому ми можемо робити такі дії, як перевірка коду стану, щоб переконатися, що операція пройшла успішно, або працювати з фактичним тілом відповіді як це показано на рисунку 2.2

```
ObjectMapper mapper = new ObjectMapper();
JsonNode root = mapper.readTree(response.getBody());
JsonNode name = root.path("name");
assertThat(name.asText(), notNullValue());
```

Рисунок 2.2 – приклад роботи з відповіддю на запит

Ми працюємо з тілом відповіді як стандартним рядком і використовуємо Джексона, і структуру вузла JSON, яку надає Джексон, для перевірки деяких деталей. Також із RestTemplate можливо зіставити відповідь безпосередньо з DTO ресурсу (рисунок 2.2)

Також RestTemplate надає можливість використання HEAD, для цього використовується API headForHeaders () як це показано на рисунку 2.3.

```
HttpHeaders httpHeaders = restTemplate.headForHeaders(fooResourceUrl);
assertTrue(httpHeaders.getContentType().includes(MediaType.APPLICATION_JSON));
```

Рисунок 2.3 – створення HEAD через RestTemplate

Для того, щоб створити новий ресурс в API, можливо використовувати наступні API: postForLocation (), postForObject () або postForEntity (). Перший повертає URI щойно створеного ресурсу, а другий повертає сам ресурс. Приклад використання на рисунку 2.4.

```
RestTemplate restTemplate = new RestTemplate();

HttpEntity<Foo> request = new HttpEntity<>(new Foo("bar"));
Foo foo = restTemplate.postForObject(fooResourceUrl, request, Foo.class);
assertThat(foo, notNullValue());
assertThat(foo.getName(), is("bar"));
```

Рисунок 2.4 – приклад використання функції postForObject()

Аналогічно, RestTemplate надає інтерфейс, який замість повернення повного ресурсу просто повертає розташування цього новоствореного ресурсу, рисунок 2.5

```
HttpEntity<Foo> request = new HttpEntity<>(new Foo("bar"));
URI location = restTemplate
  .postForLocation(fooResourceUrl, request);
assertThat(location, notNullValue());
```

Рисунок 2.5 – отримання шляху до ресурсу через RestTemplate

Звичайно RestTemplate також окрім GET методу, надає можливість створювати POST запит, приклад на (рис)

```
RestTemplate restTemplate = new RestTemplate();
HttpEntity<Foo> request = new HttpEntity<>(new Foo("bar"));
ResponseEntity<Foo> response = restTemplate
  .exchange(fooResourceUrl, HttpMethod.POST, request, Foo.class);

assertThat(response.getStatusCode(), is(HttpStatus.CREATED));

Foo foo = response.getBody();

assertThat(foo, notNullValue());
assertThat(foo.getName(), is("bar"));
```

Рисунок 2.6 – POST запит створений за допомогою RestTemplate

Також за допомогою RestTemplate можливо подати форму за допомогою методу POST. Спочатку нам потрібно встановити заголовок Content-Type на application/x-www-form-urlencoded, як це зроблено на рисунку 2.7. Це гарантує, що на сервер може бути надіслано великий рядок запиту, що містить пари ім’я/значення, розділені “&”

```
HttpHeaders headers = new HttpHeaders();
headers.setContentType(MediaType.APPLICATION_FORM_URLENCODED);
```

Рисунок 2.7 – встановлення заголовку Content-Type

Ми можемо обернути змінні форми в LinkedMultiValueMap, як це показано на рисунку 2.8. Далі ми створюємо запит за допомогою екземпляра HttpEntity

(рисунок2.9). Нарешті, можливо підключитися до служби REST, викликавши restTemplate.postForEntity () у кінцевій точці: /foos /form (рисунок 2.10)

```
MultiValueMap<String, String> map= new LinkedMultiValueMap<>();
map.add("id", "1");
```

Рисунок 2.8 – встановлення заголовку Content-Type

```
HttpEntity<MultiValueMap<String, String>> request = new HttpEntity<>(map, headers);
```

Рисунок 2.9 – створення сутності запиту

```
ResponseEntity<String> response = restTemplate.postForEntity(
  fooResourceUrl+"/form", request , String.class);

assertThat(response.getStatusCode(), is(HttpStatus.CREATED));
```

Рисунок 2.10 – приклад підключення до REST служби

Для відправлення SOAP запитів використовується WebServiceTemplate, тому бібліотека має займатися підміною запитів відправлених саме цим API. WebServiceTemplate – інтерфейс Spring Framework Для конфігурації WebServiceTemplate використовується плагін maven-jaxb2-plugin для створення анотованих класів-заглушок JAXB. Для цього додається цей плагін maven в pom.xml проекту, як це показано на рисунку 2.11.

```
<plugin>
    <groupId>org.jvnet.jaxb2.maven2</groupId>
    <artifactId>maven-jaxb2-plugin</artifactId>
    <version>0.14.0</version>
    <executions>
         <execution>
              <goals>
                  <goal>generate</goal>
              </goals>
         </execution>
    </executions>
    <configuration>
          <schemaLanguage>WSDL</schemaLanguage>
          <generateDirectory>${project.basedir}/src/main/java</generateDirectory>
          <generatePackage>com.springsoap.client.gen</generatePackage>
          <schemaDirectory>${project.basedir}/src/main/resources</schemaDirectory>
          <schemaIncludes>
             <include>countries.wsdl</include>
          </schemaIncludes>
    </configuration>
</plugin>
```

Рисунок 2.11 – конфігурація maven-jaxb2-plugin

Для роботи з SOAP сервісом, створюються клієнт SOAP за допомогою WebServiceTemplate Наприклад, створюються клас під назвою SOAPConnector.java, який буде діяти як загальний клієнт веб-служби для всіх запитів до веб-служби (рисунок 2.12).

```
import org.springframework.ws.client.core.support.WebServiceGatewaySupport;

public class SOAPConnector extends WebServiceGatewaySupport {

    public Object callWebService(String url, Object request){
        return getWebServiceTemplate().marshalSendAndReceive(url, request);
    }
}
```

Рисунок 2.12 – визначення SOAP клаєнту

Клас SOAPConnector розширює WebServiceGatewaySupport, який в основному вводить один інтерфейс із внутрішньою реалізацією WebServiceTemplate, яка доступна через метод getWebServiceTemplate (). Цей WebServiceTemplate використовується для виклику служби SOAP. Цей клас також очікує, що один введений Spring Bean під назвою Marshaller та Unmarshaller, буде наданий класом конфігурації.

Для роботи WebServiceTemplate його необхідно сконфігурувати для цього створюються клас конфігурації з коментуванням @Configuration, який матиме необхідні визначення компонентів, необхідні для SOAPConnector, щоб цей клас працював належним чином. Приклад конфігурації показаний на рисунку 2.14

```
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.oxm.jaxb.Jaxb2Marshaller;

@Configuration
public class Config {
    @Bean
    public Jaxb2Marshaller marshaller() {
        Jaxb2Marshaller marshaller = new Jaxb2Marshaller();
        // this is the package name specified in the <generatePackage> specified in
        // pom.xml
        marshaller.setContextPath("com.example.howtodoinjava.schemas.school");
        return marshaller;
    }

    @Bean
    public SOAPConnector soapConnector(Jaxb2Marshaller marshaller) {
        SOAPConnector client = new SOAPConnector();
        client.setDefaultUri("http://localhost:8080/service/student-details");
        client.setMarshaller(marshaller);
        client.setUnmarshaller(marshaller);
        return client;
    }
}
```

Рисунок 2.13 – конфігурація SOAP клаєнта

WebServiceGatewaySupport вимагає Marshaller та Unmarshaller, які є екземплярами класу Jaxb2Marshaller. Цей Spring Bean використовується як маршаллер/розкривач для компонента SOAPConnector.

2.2 Опис сценарію взаємодії користувача з бібліотекою

З точку зору користувача бібліотеки від нього буде вимагатися лише декілька кроків для конфігурації бібліотеки.

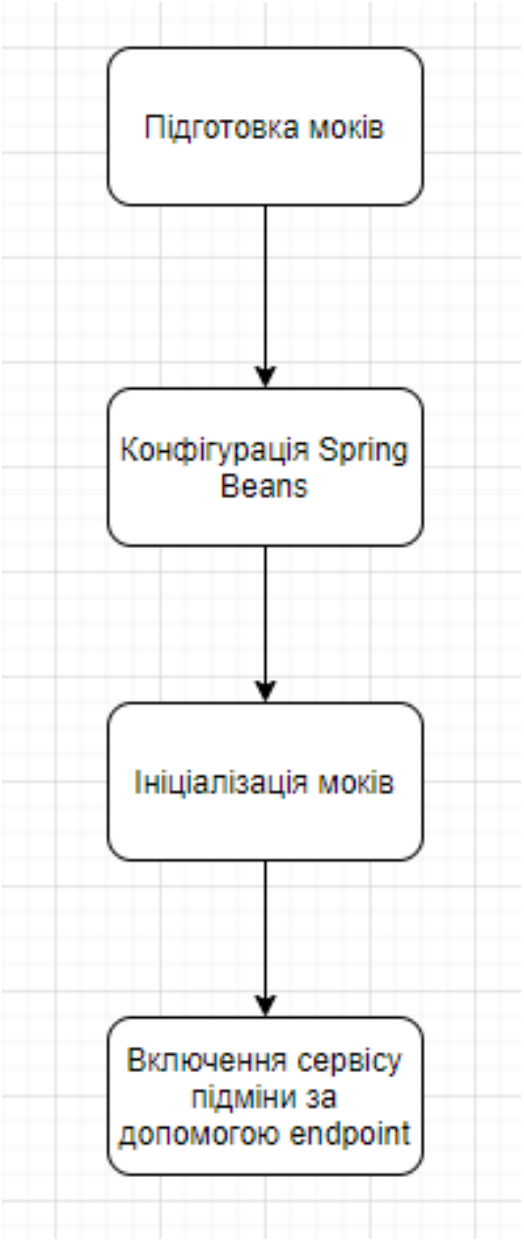


Рисунок 2.14 – алгоритм взаємодії з бібліотекою

Кроки конфігурації:

— Підготовка моків у зручному вигляді для користувача, наприклад у вигляді файлів і тд. Мок із себе представляє об’єкт, який має в собі URL запиту та відповідь для REST моків або конверт SOAP запиту і відповідно відповідь на нього для SOAP моків.

— Наступним етапом необхідно сконфігурувати сутності для Spring Framework. Головною умовою на цьому етапі буде ініціалізувати правильно сутності перехоплювачів запитів.

— Далі користувач має передати в бібліотеку моки зручним йому способом через сервіс бібліотеки.

— Після попередніх етапів додаток, який використовує бібліотеку буде працювати у штатному режимі. Для її активації необхідно буде надіслати REST запит до додатку

2.3 Огляд концепції бінів у Spring Framework

Bean є важливою концепцією Spring Framework, і тому при використанні Spring, важливо зрозуміти, що це таке і як воно працює. За визначенням, Spring Bean — це об’єкт, який складає основу програми і керується контейнером Spring IoC. Bean — це об’єкт, який створюється, збирається та іншим чином керується контейнером Spring IoC. В іншому випадку bean є просто одним із багатьох об’єктів у програмі. Beans і залежності між ними відображаються в метаданих конфігурації, які використовуються контейнером.

Усередині Spring Bean використовує функцію Inversion of Control, за допомогою якої об’єкт визначає свої залежності, не створюючи їх. Цей об’єкт делегує роботу зі створення та створення екземплярів таких залежностей контейнеру IoC.

Наприклад, якщо у нас є клас Person (рисунок 2.15), який має таку властивість класу Animal. Припустимо, що тварина має дві властивості: ім’я та вік. Щоб створити екземпляр об’єкта Person, ми повинні створити через конструктор спочатку екземпляр класу Animal і уже потім Person, як це показано на рисунку 2.16

```
public class Person {
    private Animal animal;

    public Person(Animal animal) {
        this.animal = animal;
    }
```

Рисунок 2.15 – встановлення заголовку Content-Type

```
Animal animal = new Animal("fuffy", 5);
Person person = new Person(animal);
```

Рисунок 2.16 – встановлення заголовку Content-Type

Це, безсумнівно, можна зробити, але що якби була можливість керувати таким екземпляром? Насправді це означає, що два класи тісно пов’язані, і якщо клас Animal з якоїсь причини зміниться, тобто буде додано властивість, ймовірно, зміниться і клас Person. І це може стати дуже складним для редагування десятків екземплярів Person. Тут стає в нагоді Spring і його контейнер, оскільки з ними об’єкт може отримати свої залежності з контейнера IoC, замість того, щоб самостійно створювати залежності.

Все, що нам потрібно зробити, це надати контейнеру відповідні метадані конфігурації. Для цього можна використати Spring анотації @Component та @Autowired, як це показано на рисунку 2.17. Але перед цим необхідно створити створити екземпляр тварини з анотацією Bean так як це показано на рисунку 2.18. Після цих дій екземпляр Person вже отримав ін’єкцію Aminal як екземпляр, з ім’ям Fuffy та віком 5.

```
@Component
public class Person {

    @Autowired
    private Animal animal;

    // ...
}
```

Рисунок 2.17 – конфігурація класу з наданням метаданих

```
@Configuration
public class Config {
    @Bean
    public Animal getAnimal() {
        return new Animal("Fuffy", 5);
    }
}
```

Рисунок 2.18 – створення екземпляру через Spring

Методом, відповідальним за отримання екземпляра Bean з контейнера Spring, є метод BeanFactory.getBean(). Зазвичай замість BeanFactory використовується один із відомих реалізуючих класів згідно з документацією. Яку б реалізація не

використовувалася б, існують різні сигнатури, за допомогою яких ми можна отримати Bean з контейнера Spring:

— За ім'ям - надано ім'я, повертає об'єкт, якщо Bean з таким ім'ям існує в контексті програми, інакше створює виключення NoSuchBeanDefinitionException.

— За ім'ям і типом - надано ім'я та тип, повертає Bean заданого типу, якщо він існує в контексті програми; це може бути корисно, оскільки уникайте приведення повернутого об'єкта. Цей метод може викликати виключення NoSuchBeanDefinitionException, якщо такого bean-компонента немає або BeanNotOfRequiredTypeException, якщо bean-компонент з таким типом не був знайдений

— За типом - подібний до попереднього, з тією різницею, що є можливість знайти більше бінів з тим самим типом, що спричиняє виключення NoUniqueBeanDefinitionException

— За назвою з параметрами конструктора: ми можемо передати ім'я bean і параметри конструктора (рис). У цьому випадку ми отримуємо тварину на ім’я Фуффі з віком 5 років. Але якщо Bean є одноосібним, тобто це не прототип, ми можемо отримати BeanDefinitionStoreException.

— За типом з параметрами конструктора: аналогічно попередньому, але замість імені передається тип і Bean не може бути однозначним

```
#by name
Object animal = context.getBean("animal");
```

Рисунок 2.19 – приклад отримання біну за ім’ям

```
#by name and type
Animal animal = context.getBean("animal", Animal.class);
```

Рисунок 2.20 – приклад отримання біну за ім’ям та типом класу

```
#by type
Animal animal = context.getBean(Animal.class);
```

Рисунок 2.21 – приклад отримання біну за типом класу

```
#by name and constructor parameters
Animal fuffyAnimal = (Animal) context.getBean("animal", "Fuffy", 5);
```

Рисунок 2.22 – приклад отримання біну за ім’ям

```
#by type and constructor parameters
Animal fuffyAnimal = (Animal) context.getBean(Animal.class, "Fuffy",
5);
```

Рисунок 2.23 – встановлення заголовку Content-Type

Це основні методи, які інтерфейс BeanFactory дає для отримання бинив з контейнера, і зазвичай використовується ApplicationContext для виклику цих методів. Більше того, щоб уникнути змішування логіки зі способом отримання bean, пов’язаним з фреймворком, розробник має покладатися на автоматичне підключення та ін’єкцію залежностей, щоб отримати Bean всередині іншого Spring bean.

Найчастіше створення бінів у Spring досягається за допомогою певних анотацій:

— @Component : анотація на рівні класу, яка за замовчуванням позначає bean з такою ж назвою, що й назва класу, з малою першою літерою. Можна вказати інше ім'я, використовуючи аргумент анотації.

— @Repository : анотація на рівні класу, що представляє рівень DAO програми.

— @Service : анотація на рівні класу, яка зазвичай знаходиться в класі із бізнес-логікою.

— @Controller: анотація на рівні класу, що представляє контролери в Spring MVC.

— @Configuration : анотація на рівні класу, щоб сказати, що вона може містити методи визначення Bean, анотовані @Bean.

Область дії bean визначає життєвий цикл і видимість цього bean в контекстах, в яких він використовується. Області дії bean-компонента можна розділити на базові та веб-обстеження. Базові області — це одиночні та прототипні, тоді як веб-доступні області — це запит, сеанс, додаток і веб-сокет.

Spring Framework має наступні області:

— Singleton : це означає, що контейнер створює один екземпляр цього bean-компонента, і за запитом, за ім'ям або типом, буде повернутий той самий об'єкт (рисунок 2.24).

— Прототип: щоразу, коли його запитують із контейнера, створюється інший екземпляр (рисунок 2.25).

— Запит : створює екземпляр bean для одного запиту HTTP (рисунок 2.26).

— Session : створює екземпляр bean для кожного сеансу HTTP (рисунок 2.27).

— Application : створює екземпляр bean для життєвого циклу ServletContext (рисунок 2.28)

— WebSocket : створює екземпляр bean для конкретного сеансу WebSocket. (рисунок 2.29)

```
@Bean
@Scope("singleton")
public Animal theAnimal() {
    return new Animal();
}
```

Рисунок 2.24 – приклад створення біну одиночки

```
@Bean
@Scope("prototype")
public Animal protoAnimal() {
    return new Animal();
}
```

Рисунок 2.25 – приклад створення біну у області прототипу

```
@Bean
@RequestScope
/*@Scope(value = WebApplicationContext.SCOPE_REQUEST,
    proxyMode = ScopedProxyMode.TARGET_CLASS)*/
public Animal requestAnimalBean() {
    return new Animal();
}
```

Рисунок 2.26 – приклад створення біну у області запиту

```
@Bean
@SessionScope
/*@Scope(value = WebApplicationContext.SCOPE_SESSION,
    proxyMode = ScopedProxyMode.TARGET_CLASS)*/
public Animal sessionAnimalBean() {
    return new Animal();
}
```

Рисунок 2.27 – приклад створення біну у області сеансу запиту

```
@Bean
@ApplicationScope
/*@Scope(value = WebApplicationContext.SCOPE_APPLICATION,
    proxyMode = ScopedProxyMode.TARGET_CLASS)*/
public Animal applicationAnimalBean() {
    return new Animal();
}
```

Рисунок 2.28 – приклад створення біну у області контексту

```
@Bean
@Scope(scopeName = "websocket", proxyMode = ScopedProxyMode.TARGET_CLASS)
public Animal websocketAnimalBean() {
    return new Animal();
}
```

Рисунок 2.29 – приклад створення біну у області сеансу веб-сокету

Для областей з підтримкою веб-додатків необхідний атрибут proxyMode, оскільки під час створення екземпляра контексту веб-програми немає активного запиту. Потім Spring створить проксі-сервер, який буде введено як залежність, і створить екземпляр цільового bean-компонента, коли він потрібен у запиті.

Spring Bean factory відповідає за управління життєвим циклом бінів, створених за допомогою контейнера Spring, а також контролює створення та знищення бінів. Для виконання певного користувацького коду він надає методи зворотного виклику, які можна розділити на дві групи:

— Методи зворотного виклику після ініціалізації

— Методи зворотного виклику перед знищенням

Spring Framework надає такі чотири способи керування подіями життєвого циклу біна:

— Інтерфейси зворотного виклику InitializingBean і DisposableBean

— Інтерфейси для певної поведінки (BeanNameAware, BeanClassLoaderAware, BeanFactoryAware, ApplicationContextAware)

— користувацькі методи init() і destroy() у файлі конфігурації bean

— Анотації @PostConstruct і @PreDestroy

Життєвий цикл Spring Bean складається з таких кроків:

— Bean Definition: bean визначається за допомогою анотацій або XML

— Bean Instantiation: Spring створює екземпляри bean-об'єктів так само, як ми вручну створюємо екземпляр об'єкта Java та завантажуємо об'єкт у ApplicationContext

— Populating Bean properties: Spring сканує bean-компонент, який реалізує інтерфейс Aware, і починає встановлювати відповідні властивості як ідентифікатор, область і значення за замовчуванням на основі визначення bean-компонента.

— Pre-Initialization: BeanPostProcessors вступають в дію на цьому етапі. Методи postProcessBeforeInitialization() виконують свою роботу. Крім того, анотовані методи @PostConstruct запускаються відразу після них.

— AfterPropertiesSet: Spring виконує методи afterPropertiesSet() компонентів, які реалізують InitializingBean

— Custom Initialization: Spring запускає методи ініціалізації, які були визначені в атрибуті initMethod анотацій @Bean

— Post-initialization: BeanPostProcessors працюють вдруге. Ця фаза запускає методи postProcessAfterInitialization().

— Ready: тепер бін створений і введений у всі залежності

— Pre-Destroy: Spring запускає анотовані методи @PreDestroy на цьому етапі

— Destroy: Spring виконує методи destroy() реалізований у DisposableBean

— Custom Destruction: цей етап існує за умови, що визначені спеціальні гачки знищення за допомогою атрибута destroyMethod в анотації @Bean, і Spring запускає їх на останньому етапі.

2.4 Визначення структури проекту

Для початку визначено структуру пакетів, яку буде мати бібліотека. Вона буде виглядати як це показано на рисунку 2.30.

Проект буде складатися з наступних пакетів:

— Exception – пакет в якому будуть зберігатися специфічні для бібліотеки виключення

— Interceptors – пакет для збереження класів, які будуть відповідати за перехоплення запиту

— Models – пакет для зберігання моделей, які будуть використовуватися бібліотекою

— Controllers – пакет для класу, який буде надавати можливість керувати бібліотекою

— Services – пакет для збереження класів сервісів, які будуть виконувати більшу частину логіки бібліотеки

— Utils – пакет в якому має зберігатися логіка спільна для всього проекту

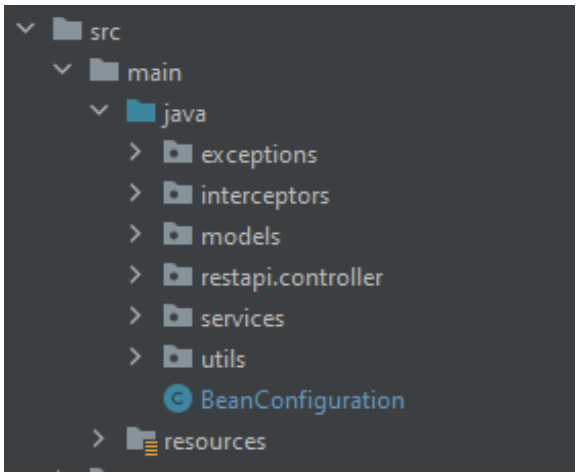


Рисунок 2.30 – структура пакетів бібліотеки

2.5 Розробка службового сервісу

В пакеті Util створимо клас RequestMockProperties (рисунок 2.31). Цей клас має в собі таку структуру даних як мапа, кожне значення якої представляє із себе ключ та його значення. Призначення цього класу зберігання збереження пропертів для можливості зміни їх під час виконання.

Клас безпосередньо складається із поля properties (рисунок 2.32) та методів для взаємодії із цим полем. Метод getProperty (рисунок 2.33) в якості вхідного значення очікує ключ за яким в мапі шукає значення і повертає. Метод setProterty в я кості вхідних значень очікує ключ та значення , які будуть записані в мапу (рисунок 2.34).

Також цей клас необхідно відмітити анотацією @Component це буде необхідно надалі на рівні сервісу для автоматичної ін’єкції залежностей.

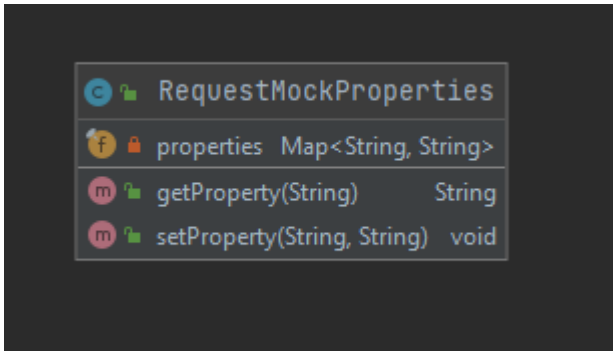


Рисунок 2.31 – UML діаграма класу RequestMockProperties

```
private final Map<String, String> properties = new HashMap<>();
```

Рисунок 2.32 – поле properties класу RequestMockProperties

```
public String getProperty(String propertyName) {
    return properties.get(propertyName);
}
```

Рисунок 2.33 – метод getProperty класу RequestMockProperties

```
public void setProperty(String propertyName, String propertyValue) {
    properties.put(propertyName, propertyValue);
}
```

Рисунок 2.34 – метод setProterty класу RequestMockProperties

2.6 Розробка моделей

В пакеті Models створимо моделі, які будуть використовуватися надалі у всьому проекті. Насамперед створимо перерахування RequestType (рисунок 2.35), яке включає в собі два значення REST та SOAP, так як запити саме цих типів будуть підмінюватися

Також в цьому пакеті створюються клас RequestMock, який має наступні поля:

- identifierOfRequest – поле строкового типу для збереження параметрів запиту будь-то REST запиту чи SOAP запиту
- response – поле строковоги типу для збереження відповіді на запит, яке потенційно може зберігати в собі будь-який текстовий формат обміну даних в мережі
- requestType – поле типу перерахування, яке було создано раніше. Відповідно до дизайну перерахування RequestType дане поле може містити в собі значення REST або SOAP

Також у класі RequestMock наявні наступні методи: getIdentifierOfRequest, getResponse, getRequestType, setIdentifierOfRequest, setResponse, setRequestType (рисунок 2.35). Це методи доступу до полів класу, вони необхідні для підтримки принципу інкапсуляції. Мета цього не просто сховати змінну, мета забезпечити приховування деталей реалізації, можливо іноді досить складних, внутрішнього стану класу і надати в інтерфейсі класу більш простий і безпечний спосіб управління цим станом. Інша сторона - це життєвий цикл програми, сьогодні ти зробив метод Set, який просто надає значення внутрішньої змінної, а завтра вимоги змінилися, і змінну потрібно обчислювати за якоюсь формулою. І ось тут інкапсуляція допоможе не змінювати в десятках місць по всій програмі привласнення на обчислення формули, а міняти його тільки в одному місці, в тілі способу. Тобто. отримуємо ще одне зниження складності у супровіді програми.

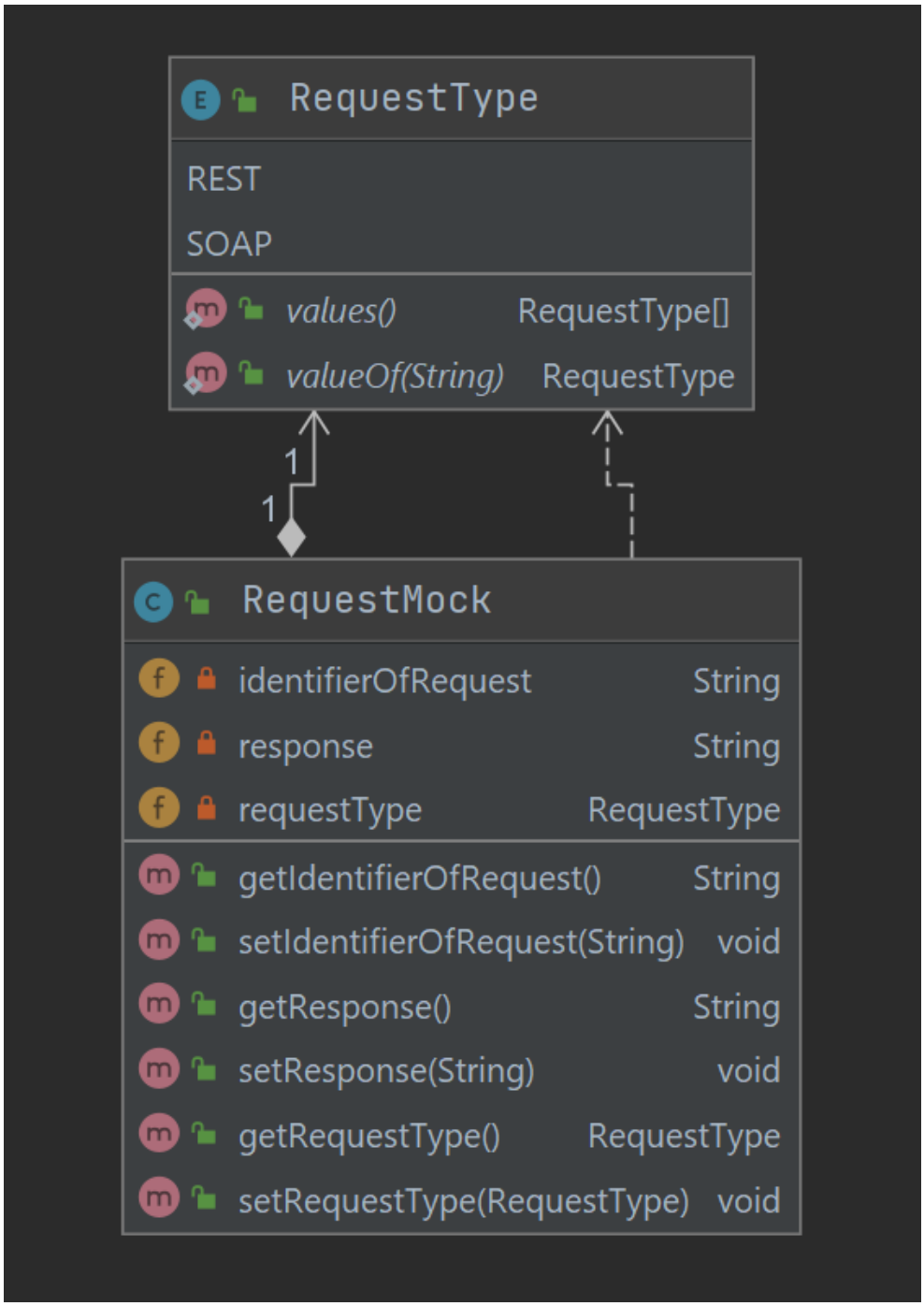


Рисунок 2.35 – UML діаграма моделей бібліотеки

2.7 Розробка сервісів

В пакеті Services будуть розміщені класи з більшою частиною основної логіки (рисунок 2.36).

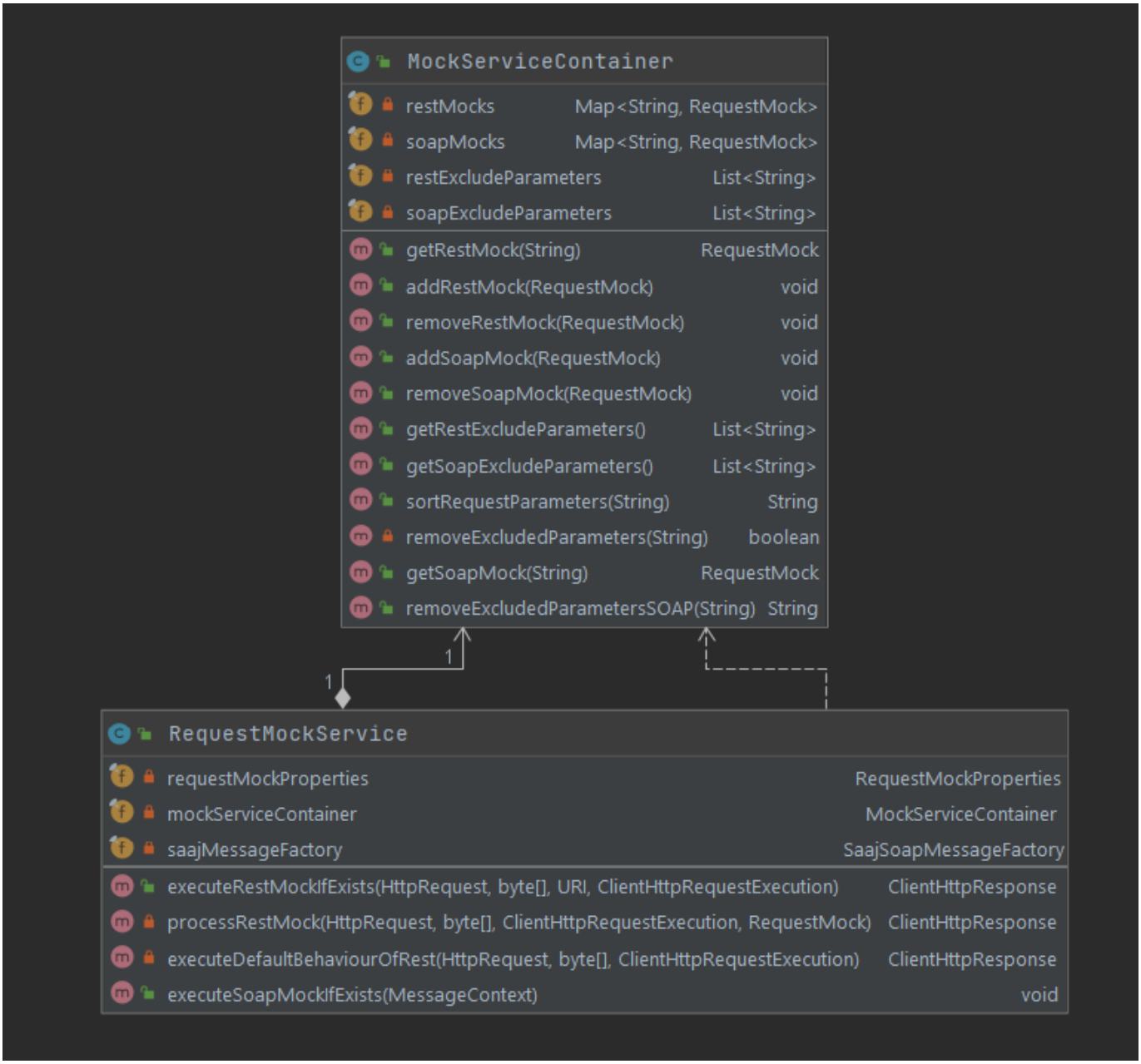


Рисунок 2.36 – UML діаграма сервісів бібліотеки

Одним із сервісів буде MockServiceContainer (рисунок 2.36), призначенням цього сервісу являється збереження моків для запитів. Цей клас має наступні поля:

— restMocks – змінна типу Map для збереження REST моків, де ключ це ідентифікатор доку, а значення безпоседньо мок

— soapMocks – змінна типу Map для збереження SOAP моків, де ключ це ідентифікатор доку, а значення безпоседньо мок

— restExcludeParameters – змінна типу List для збереження параметрів, які будуть ігноруватися при пошуку REST моків

— soapExcludeParameters - змінна типу List для збереження параметрів, які будуть ігноруватися при пошуку SOAP моків

Важливі відзначи, що клас MockServiceContainer за замовчування реалізує патерн одиночка, як Spring Bean. Тому необхідно запечпечити потокобезпечний доступ до полів класу. З цим може допомогти реалізація типу Map, як ConcurrentHashMap. На відміну від елементів HashMap, Entry в ConcurrentHashMap оголошено як volatile. Це важлива особливість, пов'язана також зі змінами в JMM. У ConcurrentHashMap також використовується покращена функція хешування. У чому потреба ускладнення хеш-функції, таблиці в хеш-мапі мають довжину, що визначається ступенем двійки. Для хеш-кодів, двійкові уявлення яких не відрізняються в молодшій та старшій позиції, ми матимемо колізії. Ускладнення хеш-функції вирішує цю проблему, зменшуючи ймовірність колізій в карті. Карта ділиться на N різних сегментів, 16 за замовчуванням, максимальне значення може бути 16-бітним і є ступенем двійки. Кожен сегмент є потокобезпечною таблицею елементів карти. Між хеш-кодами ключів і відповідними сегментами встановлюється залежність на основі застосування до старших розрядів хеш-коду бітової маски. Враховуючи псевдовипадковий розподіл хешів ключів усередині таблиці, можна зрозуміти, що збільшення кількості сегментів сприятиме тому, що операції модифікації

торкатимуться різних сегментів, що зменшить ймовірність блокувань під час виконання. ConcurrencyLevel Цей параметр впливає на використання картки пам'яті та кількість сегментів у картці. Кількість сегментів буде вибрано як найближчий ступінь двійки, більший за конкурентний рівень. Ємність кожного сегмента, відповідно, визначатиметься як відношення заокругленого до найближчого більшого ступеня двійки значення ємності карти за умовчанням, до отриманої кількості сегментів. Дуже важливо розуміти дві такі речі. Заниження рівня конкуренції веде до того, що найбільш вірогідні блокування потоками сегментів картки під час запису. Завищення показника призводить до неефективного використання пам'яті. Тому якщо лише один потік змінюватиме карту, а решта читатиме — рекомендується використовувати значення 1. Необхідно пам'ятати, що resize таблиць для зберігання всередині картки - операція, що вимагає додаткового часу і часто виконується не швидко. Тому при створенні картки потрібно мати деякі приблизні оцінки статистики виконання можливих операцій читання та запису. Тому це ідеальний кандидат для вирішення проблеми з потокобезпекою.

Сервіс має наступні методи:

— getRestMock – метод, який приймає ідентифікатор REST моку і шукає його в мапі restMocks. Якщо об’єкт не був знайдений повертається значення null

— getSoapMock - метод, який приймає ідентифікатор SOAP моку і шукає його в мапі soapMocks. Якщо об’єкт не був знайдений повертається значення null

— addRestMock – метод, який запус REST мок до мапи. Цей метод не просто запусує сутність до колекції, перед тим як її записути вона готоється для цього. А саме, створюється URI із ідентифікатора, сортуються параметри запиту і вже потім записується до колекції. Це неодхідно для того, щоб

бути впевненими, коли порядок параметрів REST запиту зміниться необхідний об’якт в будь-якому випадку знайдеться. Реалізація методу показана на рисунку 2.37.

— removeRestMock - метод, який видаляє REST мок із мапи

— addSoapMock - метод, який запус SOAP мок до мапи

— removeSoapMock - метод, який видаляє SOAP мок із мапи

— getRestExcludeParameters – метод для отримання списку параметрів, які мають бути виключені із REST запиту

— getSoapExcludeParameters - метод для отримання списку параметрів, які мають бути виключені із SOAP запиту

— sortRequestParameters – приватний метод класу необхідний для методу addRestMock. В цьому методі реалізований механіз сортування параметрів запиту. В якості вхідного параметку він отримує строку REST запиту. Для отримання параметрів спочатку отримана строка ділиться за символом “&”. Далі кожен елемент отриманого списку перевіряється методом removeExcludedParameters. Після виконання списку масив параметрів знову перетворюється в строку REST запиту, але параметри вже відсортовані. Реалізація методу показана на рисунку 2.38

— removeExcludedParameters – метод, який перевіряє навність параметру в списку REST виключень і відповідно до цього повертає логічне значення true або false. Реалізація методу показана на рисунку 2.39.

— removeExcludedParametersSOAP - метод, який перевіряє навність параметри в списку SOAP виключень і відповідно до цього форматує повідомлення. Так як робота з xml файлами досить комплексна задача для написання цього методу була використана стороння бібліотека, а саме,

jsoup. Так як проек використовує maven в якості інструменту зборки, додання цієї бібліотеки буде досягнуто зміною файлу конфігурації pom.xml. Для додання цієї бібліотеки необхідно відредагувати вершину dependencies як це показано на рисунку 2.40. Метод працює наступним чином. Спочатку створюється jsoup документ для кожної вершини якого йде перевірка чи є ця вершина xml документи у списку виключень. Якщо вершина є вона видаляється, інакше цикле йде до наступної вершини. І в кінці метод перетворює jsoup документ у строку. Реалізація методу показана на рисунку 2.41.

```
public void addRestMock(RequestMock requestMock) {
    URI request = URI.create(requestMock.getIdentifierOfRequest());
    String sortedUrl = request.getHost() + request.getPath() + sortRequestParameters(request.getQuery());
    restMocks.put(sortedUrl, requestMock);
}
```

Рисунок 2.37 – реалізація методу addRestMock класу MockServiceContainer

```
private String sortRequestParameters(String parameters) {
    return Arrays.stream(parameters.split( regex: "&"))
            .filter(this::removeExcludedParameters)
            .sorted()
            .collect(Collectors.joining( delimiter: "&",  prefix: "",  suffix: ""));
}
```

Рисунок 2.38 – реалізація методу sortRequestParameters класу MockServiceContainer

```
private boolean removeExcludedParameters(String parameterValue) {
    return !restExcludeParameters.contains(parameterValue.split( regex: "=")[0]);
}
```

Рисунок 2.39 – реалізація методу removeExcludedParametersSOAP класу MockServiceContainer

```
<!-- https://mvnrepository.com/artifact/org.jsoup/jsoup -->
<dependency>
    <groupId>org.jsoup</groupId>
    <artifactId>jsoup</artifactId>
    <version>1.14.3</version>
</dependency>
```

Рисунок 2.40 – залежність на jsoup

```
public String removeExcludedParametersSOAP(String soapEnvelope) {
    Document doc = Jsoup.parse(soapEnvelope);
    for (String excludeNode : soapExcludeParameters) {
        ArrayList<Element> els = doc.getElementsByTag(excludeNode);
        for (Element el : els) {
            el.remove();
        }
        doc = Jsoup.parse(doc.body().children().toString());
    }

    return doc.toString();
}
```

Рисунок 2.41 – реалізація методу removeExcludedParametersSOAP класу MockServiceContainer

Другим сервісом являється клас RequestMockService призначенням цього сервісу є пошук та виконання моків. Цей клас має наступні поля, інєкція яких роботися силами Spring Framework:

— requestMockProperties – поле типу RequestMockProperties із пакету Utils, основним призначення цієї залежності робота із властивостями під час виконання програми

— mockServiceContainer – поле типу MockServiceContainer, більшість методів класу MockServiceContainer використовуються саме в RequestMockService

— saajMessageFactory – поле типу SaajSoapMessageFactory, використовуэться для створення нового SOAP повідомлення.

Саме в цьому класі можна побачити всі переваги використання Spring Framework. Очевидно, що немає необхідності вручно створювати всі ці полі при зверненні до функціональних помливостей сервісу RequestMockService. Інєкція в цьому сервісі робиться за допомогою анотації @ Autowired і розміщена вона над конструктором класу, як це рекомендується робити в Spring документації (рисунок 2.42).

```
@Autowired
public RequestMockService(RequestMockProperties requestMockProperties,
                          MockServiceContainer mockServiceContainer,
                          SaajSoapMessageFactory saajMessageFactory) {
    this.requestMockProperties = requestMockProperties;
    this.mockServiceContainer = mockServiceContainer;
    this.saajMessageFactory = saajMessageFactory;
}
```

Рисунок 2.42 – ін’єкція залежностей в клас RequestMockService

Сервіс має наступні методи:

— executeRestMockIfExists – метод, який спочатку шукає REST мок за допомогою mockServiceContainer, а потім визиває метод processRestMock. Реалізація методу на рисунку 2.43.

— processRestMock – метод, який спочатку перевіряє чи був знайдений мок у попередньому методі, якщо ні то виконується метод executeDefaultBehaviourOfRest інакше із мокy береться відповідь для підміни і продувжується виконання запиту із зміненою відповіддю. Реалізація методу показана на рисунку 2.44.

— executeDefaultBehaviourOfRest – цей метод виконується у разі, якщо мок не був знайдений. В такій ситуації запит іде до справжнього сервісу без зміни відповіді.

— executeSoapMockIfExists – метод, який спочатку зчитує SOAP повідомлення з класу MessageContext переданої у метод в якості вхідних параметрів. Далі за допомогою mockServiceContainer шукається мок для цього SOAP запиту. Якщо мок був знайдений створюється нове повідомлення і перезаписує вже існуєче повідомлення в об’єкті типу MessageContext, інакше відповідь на SOAP повідомлення залишається таким, яким була встановлена сторонньому сервісі.

```
public ClientHttpResponse executeRestMockIfExists(HttpRequest httpRequest, byte[] defaultResponse,
                                                   URI uri, ClientHttpRequestExecution clientHttpRequestExecution) throws IOException {
    RequestMock mock = mockServiceContainer.getRestMock(
            mockIdentifier: uri.getHost() + uri.getPath() +
                    mockServiceContainer.sortRequestParameters(uri.getQuery()));
    return processRestMock(httpRequest, defaultResponse, clientHttpRequestExecution, mock);
}
```

Рисунок 2.43 – реалізація методу executeRestMockIfExists класу RequestMockService

```
private ClientHttpResponse processRestMock(HttpRequest httpRequest, byte[] defaultResponse, ClientHttpRequestExecution client
    if (Objects.nonNull(requestMock)) {
        clientHttpRequestExecution.execute(httpRequest, requestMock.getResponse().getBytes(StandardCharsets.UTF_8));
    }
    return executeDefaultBehaviourOfRest(httpRequest, defaultResponse, clientHttpRequestExecution);
}
```

Рисунок 2.44 – реалізація методу processRestMock класу RequestMockService

```
public void executeSoapMockIfExists(MessageContext messageContext) throws IOException, SOAPException {
    OutputStream outputStream = new ByteArrayOutputStream();
    messageContext.getRequest().writeTo(outputStream);
    String soapRequest = outputStream.toString();

    RequestMock mock = mockServiceContainer.getSoapMock(mockServiceContainer.removeExcludedParametersSOAP(soapRequest));
    if (Objects.nonNull(mock)) {
        InputStream is = new ByteArrayInputStream(mock.getResponse().getBytes());
        SaajSoapMessage message = saajMessageFactory.createWebServiceMessage();
        message.setSaajMessage(MessageFactory.newInstance().createMessage( headers: null, is));
        messageContext.setResponse(message);
    }
}
```

Рисунок 2.45 – реалізація методу executeSoapMockIfExists класу RequestMockService

2.8 Розробка інтерсепторів

В пакеті Interceptors будуть розміщені класи, які відповідають за логіку перехоплення запитів (рисунок 2.46). У більшості парадигм програмування перехоплювачі є важливою частиною, яка дозволяє програмістам контролювати виконання, перехоплюючи його.

Spring Framework також підтримує різноманітні перехоплювачі для різних цілей. Для REST запитів Spring RestTemplate дозволяє нам додавати перехоплювачі,

які реалізують інтерфейс ClientHttpRequestInterceptor. Метод intercept(HttpRequest, byte[], ClientHttpRequestExecution) цього інтерфейсу перехопить даний запит і поверне відповідь, надавши нам доступ до запиту, тіла та об’єктів виконання. Аргументи методу intercept інтерфейсу ClientHttpRequestExecution можливо використовувати, щоб виконати фактичне виконання, і передамо запит до наступного ланцюжка процесів. Оскільки метод intercept() включив запит і тіло як аргументи, також можна внести будь-які зміни в запит або навіть відмовити у виконанні запиту на основі певних умов.

Перехоплювач бібліотеки RestInterceptor (рисунок 2.46) буде викликатися для кожного вхідного запиту, і він буде перевіряти чи включена система підміни запиту. Якщо так тоді буде викликатися метод executeRestMockIfExists із сервісу RequestMockService. Реалізація методу intercep класу RestInterceptor на рисунку 2.47. Також в цьому інтерсепторі буде використаний механіз автоматичної ін’єкції залежностей для наступних полів інтерсептора:

— requestMockProperties – використовується для перевірки чи активна логіка підміни запиту в момент виклику інтерсептора.

— requestMockService – використовується для виклику метода executeRestMockIfExists.

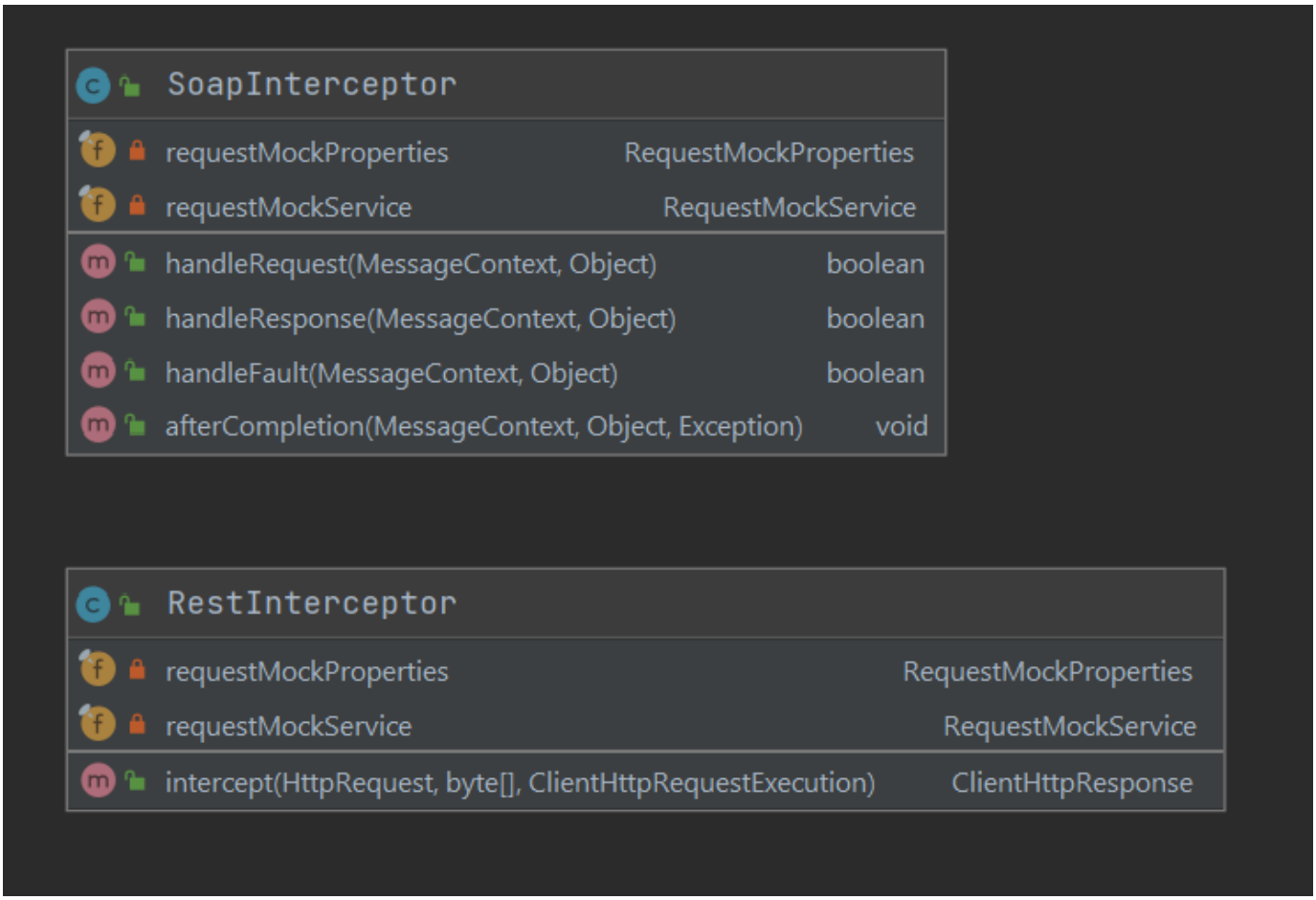


Рисунок 2.46 – UML діаграма інтерсепторів бібліотеки

```
@Override
public ClientHttpResponse intercept(HttpRequest httpRequest, byte[] bytes, ClientHttpRequestExecution clientHttpRequestExecution) throws IOException {

    if (Boolean.getBoolean(requestMockProperties.getProperty("is.mocking.enables"))) {
        return requestMockService.executeRestMockIfExists(httpRequest, bytes, httpRequest.getURI(), clientHttpRequestExecution);
    }
    return clientHttpRequestExecution.execute(httpRequest, bytes);
}
```

Рисунок 2.47 – реалізація методу intercept класу RestInterceptor

Для того, щоб інтерсептор RestInterceptor працював його необхідно зареєструвати. Для цього створимо в пакеті Configurations клас BeanConfiguration та анотаємо його анотаціює @Configuration. В цьому класі створюємо бін класу

RestTemplate. В ньому отримуємо бін інтерсептора RestInterceptor із контексту і додаємо його до інтерсепторів у RestTemplate (рисунок 2.48).

```
@Bean
public RestTemplate restTemplate() {
    RestTemplate restTemplate = new RestTemplate();
    restTemplate.setInterceptors(Collections.singletonList(context.getBean(RestInterceptor.class)));
    return restTemplate;
}
```

Рисунок 2.48 – ініціалізація RestInterceptor

Для перехоплення SOAP запитів можна використати інтерфейс EndpointInterceptor. Він має чотири методи handleRequest, handleResponse, handleFault, afterCompletion. Для реалізації логіки бібліотеки необхідно модифікувати тіло методу handleResponse для зміни відповіді на запит. Для цього створимо бін SoapInterceptor, який імплементує інтерфейс EndpointInterceptor. В методі як і в RestInterceptor перевіряє чи активна система підміни. Якщо так то викликається метод executeSoapMockIfExists, інакше нічого не змінюється. Реалізація методу handleResponse класу SoapInterceptor на рисунку 2.49. Також в цьому інтерсепторі буде використаний механіз автоматичної ін’єкції залежностей для наступних полів інтерсептора:

— requestMockProperties – використовується для перевірки чи активна логіка підміни запиту в момент виклику інтерсептора.

— requestMockService – використовується для виклику метода executeSoapMockIfExists.

```
@Override
public boolean handleResponse(MessageContext messageContext, Object o) throws Exception {

    if (Boolean.getBoolean(requestMockProperties.getProperty("is.mocking.enables"))) {
        requestMockService.executeSoapMockIfExists(messageContext);
    }

    return true;
}
```

Рисунок 2.49 – реалізація методу handleResponse класу SoapInterceptor

Для того, щоб інтерсептор SoapInterceptor працював його необхідно зареєструвати. Для цього створимо в пакеті Configurations клас SoapHandlerConfiguration наслідуємо його від класу WsConfigurerAdapter та анотаємо його анотаціями @Configuration та @EnableWs. В цьому класі перевизначаємо метод addInterceptors. В якому беремо бін із контексту та додаємо до списку інтерсепторів (рисунок 2.51).

```
@Override
public void addInterceptors(List<EndpointInterceptor> interceptors)
{
    interceptors.add(context.getBean(SoapInterceptor.class));
}
```

Рисунок 2.50 – ініціалізація SoapInterceptor

2.9 Огляд принципу роботи Spring Controller

Spring MVC – це веб-фреймворк Spring. Він дозволяє створювати веб-сайти або RESTful сервіси (наприклад, JSON/XML) і добре інтегрується в екосистему Spring, наприклад, він підтримує контролери та REST контролери. Він має наступні компоненти:

- DispatcherServlet
- Handler Mapper
- Controller
- ViewResolver
- View

На рисунку 2.51 показано, як крок за кроком обробляється запит HTTP від початку до кінця в фреймворку Spring MVC.

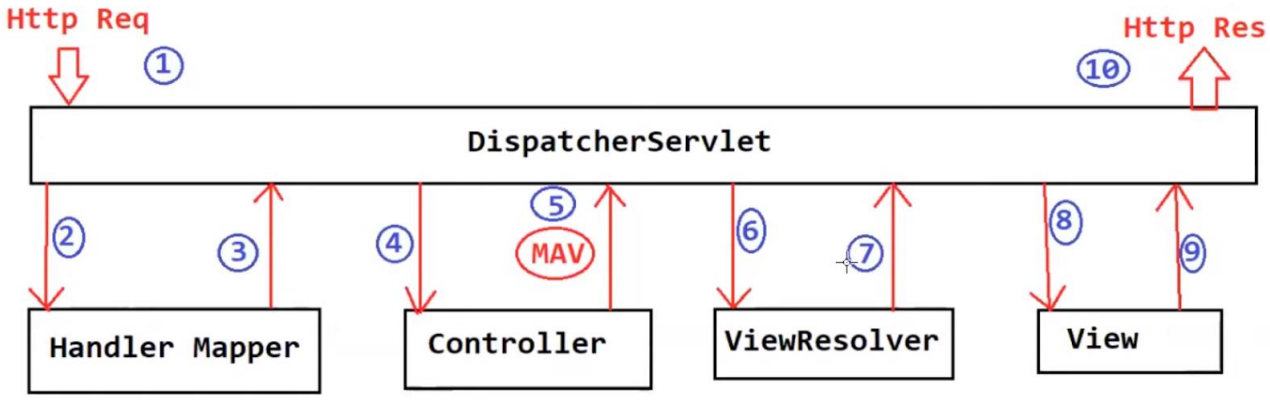


Рисунок 2.51 – алгоритм обробки HTTP запиту у Spring MVC

Обробка проходить через наступні кроки:

— Крок 1 - Коли клієнт (браузер) надсилає HTTP запит на певну URL-адресу. DispatcherServlet Spring MVC отримує запит.

— Крок 2 - DispatcherServlet звертається до HandlerMapper, щоб визначити, який контролер відповідає за обробку HTTP запиту.

— Крок 3 - HandlerMapper вибирає контролер, який зіставляється з URL-адресою вхідного запиту, і повертає вибраний обробник і контролера в DispatcherServlet.

— Крок 4 - Тепер DispatcherServlet знає, який контролер відповідає за обробку запиту, тому DispatcherServlet перешле цей запит відповідному контролеру для обробки запиту.

— Крок 5. Тепер контролер обробляє запит, перевіряє запит і створює модель з даними. Нарешті, контролер повертає логічне ім’я представлення та моделі в DispatcherServlet.

— Крок 6 – DispatcherServlet звертається до ViewResolver, щоб розв’язати логічне представлення з фізичним представленням, яке існує в програмі.

— Крок 7 - ViewResolver відповідає за відображення логічного вигляду з фактичним виглядом і повернення фактичних деталей перегляду назад у DispatcherServlet.

— Крок 8 - Тепер DispatcherServlet надсилає подання та модель до компонента View.

— Крок 9. Компонент View об'єднує подання та модель і формує звичайний вихідний HTML. Нарешті, компонент View надсилає вихідні дані HTML назад у DispatcherServlet.

— Крок 10 – DispatcherServlet нарешті надсилає вихідний HTML-код як відповідь назад до браузера для відтворення.

У традиційному підході додатки MVC не орієнтовані на сервіси, тому існує View Resolver, який відтворює остаточні уявлення на основі даних, отриманих від контролера.

Програми RESTful розроблені так, щоб бути сервісно-орієнтованими та повертати вихідні дані, як правило JSON/XML. Оскільки ці програми не виконують візуалізацію представлення, не існує View Resolver, зазвичай очікується, що контролер надсилатиме дані безпосередньо через відповідь HTTP. Налаштування для програми Spring RESTful таке ж, як і для програми MVC, з тією лише різницею, що немає View Resolvers і немає карти моделі. API, як правило, просто повертає необроблені дані назад клієнту – як правило, представлення XML та JSON – і тому DispatcherServlet обходить View Resolvers і повертає дані прямо в тілі відповіді HTTP.

Для використання цієї функціональності як правило використовуються анотації: @Controller, @RestController, @ResponseBody із Spring.

2.10 Розробка контролера

В пакеті Controllers буде розміщений клас, який відповідає за логіку взаємодії із бібліотекою через API інтерфейс (рисунок 2.52).

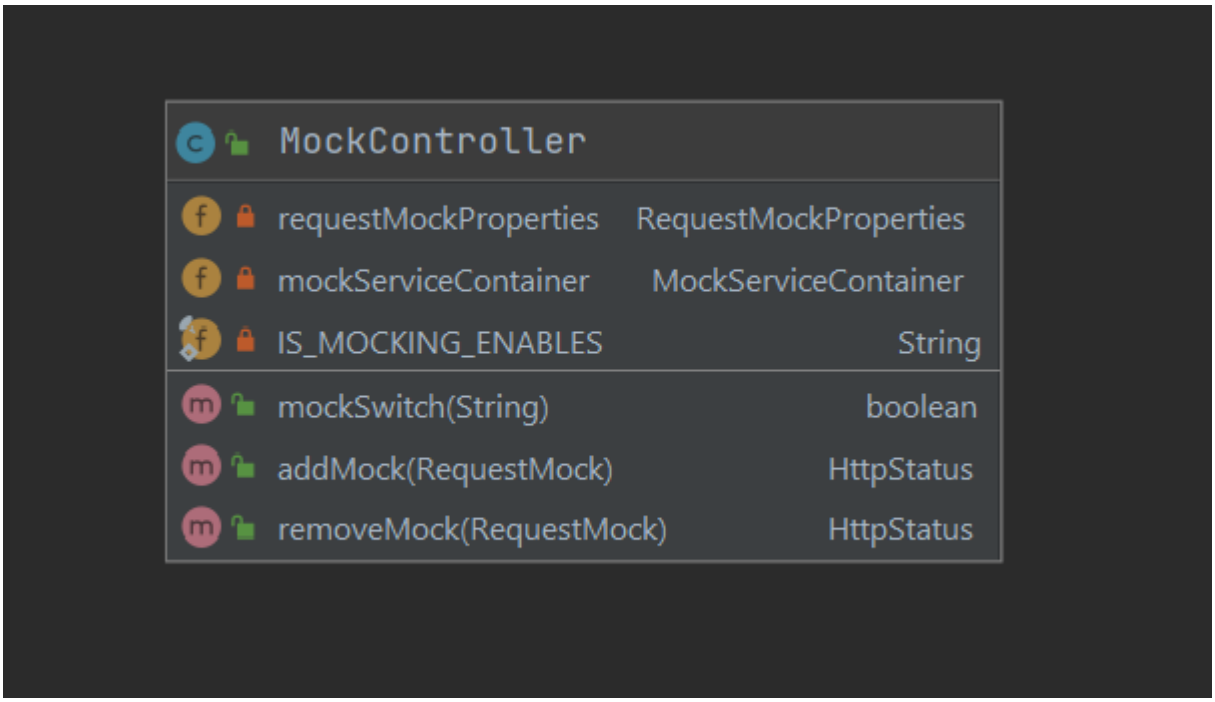


Рисунок 2.52 – UML представлення MockController

Контролером в цьому пакеті буде клас MockController. Для того, що його зареєструвати як контролер цей клас анотується анотацією @RestController("/mock"), в якості параметру в контролер передаєтьтся строка “/mock”, це означає, що для звернення до контролера необхідно починати шлях запиту із цього слова. Також до цього класу додається анотація @CrossOrigin без параметрів, щоб дозволити всі адресам взаємодіяти з цим контролером. Клас має наступнв полі, ін’єкція, яких буде зроблена силами Spring Framework (рисунок 2.53):

— requestMockProperties – використовується в класі для включення, виключення логіки підміни запиту

— mockServiceContainer – використовується для ініціалізації моків та їх видалення за необхідності.

Контролер має нуступні методи:

— mockSwitch – метод для включення, виключення логіки підміни запитів, анотований анотацією @GetMapping("/switch/{switchValue}"). Ця анотація змушує співставляти цей метод із GET запитом, який іде із шляхом “/mock/switch/”, і значення “{switchValue }” взято у дужки для того щоб Handler Mapper очікував запит із змінною у шляху, тобто повний запит до цього методу виглядатиме, як ““/mock/switch/true”. Якщо змінна передана у шлях являється логічним значення true або false тоді змінюється змінна в requestMockProperties, інакше створюється виключення HttpClientErrorException із статусом 400, тобто BAD_REQUEST. Реалізація методу на рисунку 2.54.

— addMock - метод для завантаження нових моків, анотований анотацією @PostMapping("/add"). Ця анотація змушує співставляти цей метод із POST запитом, який іде із шляхом “/mock/add”. Також цей метод очікую отримати об’єкт типу RequestMock у тілі запиту. Після того як запит прийшов перевіряється чи відповідає поле requestType із RequestMock одному із дозволених значень. Якщо так то новий об’єкт записується до списку моків через mockServiceContainer, інакше створюється виключення із пакету Exceptions UnknownMockTypeException. Реалізація методу зображена на рисунок 2.55.

— removeMock - метод для видалення нових моків, анотований анотацією @PostMapping("/remove"). Ця анотація змушує співставляти цей метод із POST запитом, який іде із шляхом “/mock/ remove”. Також цей метод очікую отримати об’єкт типу RequestMock у тілі запиту. Після того як запит прийшов перевіряється чи відповідає поле requestType із RequestMock одному із дозволених значень. Якщо так то новий об’єкт видаляється із списку моків через mockServiceContainer, інакше створюється виключення із пакету Exceptions

UnknownMockTypeException. Реалізація методу зображена на рисунок 2.56

```
@Autowired
public MockController(RequestMockProperties requestMockProperties, MockServiceContainer mockServiceContainer) {
    this.requestMockProperties = requestMockProperties;
    this.mockServiceContainer = mockServiceContainer;
}
```

Рисунок 2.53 – ін’єкція залежностей у MockController

```
@GetMapping("/switch/{switchValue}")
public boolean mockSwitch(@PathVariable String switchValue) {
    if (switchValue.equalsIgnoreCase( anotherString: "true") || switchValue.equalsIgnoreCase( anotherString: "false")) {
        requestMockProperties.setProperty(IS_MOCKING_ENABLES, switchValue);
        return Boolean.parseBoolean(requestMockProperties.getProperty(IS_MOCKING_ENABLES));
    }
    throw new HttpClientErrorException(HttpStatus.BAD_REQUEST, "Received not boolean path variable");
}

@PostMapping("/add")
```

Рисунок 2.54 – реалізація методу mockSwitch класу MockController

```
@PostMapping("/add")
public HttpStatus addMock(@RequestBody RequestMock requestMock) {
    if (RequestType.REST.equals(requestMock.getRequestType())) {
        mockServiceContainer.addRestMock(requestMock);
        return HttpStatus.OK;
    } else if (RequestType.SOAP.equals(requestMock.getRequestType())) {
        mockServiceContainer.addSoapMock(requestMock);
        return HttpStatus.OK;
    }

    throw new UnknownMockTypeException("Unknown type of mock : " + requestMock.getRequestType() + " Mock can have type REST or SOAP");
}
```

Рисунок 2.55 – реалізація методу addMock класу MockController

```
@PostMapping("/remove")
public HttpStatus removeMock(@RequestBody RequestMock requestMock) {
    if (RequestType.REST.equals(requestMock.getRequestType())) {
        mockServiceContainer.removeRestMock(requestMock);
        return HttpStatus.OK;
    } else if (RequestType.SOAP.equals(requestMock.getRequestType())) {
        mockServiceContainer.removeSoapMock(requestMock);
        return HttpStatus.OK;
    }

    throw new UnknownMockTypeException("Unknown type of mock : " + requestMock.getRequestType() + " Mock can have type REST or SOAP");
}
```

Рисунок 2.56 – реалізація методу removeMock класу MockController

ВИСНОВКИ

В рамках даної випускної кваліфікаційної роботи був проведений аналіз протоколів передачі даних, визначені їх параметри та особливості. Також в рамках цього аналізу були виведені вимоги до бібліотеки, які при розробці враховувалися. На основі проведеного аналізу був визначений процес взаємодії розробника з готовою бібліотекою.

На основі розробленого сценарію взаємодії були визначені необхідні функціональні модулі проекту. На основі цього списку був обраний найбільш підходящий інструмент в обличчі Spring Framework. На основі цього вибору був проведений огляд цієї технології, особливостей її роботи та переваг, які можливо отримати при розробці. Окрім цього було визначено призначення кожного функціонального модуля.

Після визначення технічного завдання, сценарію взаємодії та структури проекту залишилася їх реалізація. Модуль за модулем був розроблений базис бібліотеки, починаючи зі службового модуля, який має підтримувати можливість конфігурації під час виконання. Наступним етапом були визначені моделі з якими буде працювати бібліотека та описані у вигляді класів. Далі була розроблена основна логіки програми розділена на два логічні сервіси, один з них являється контейнером для моків, інший реалізує логіку пошуку необхідного мока та підміну запиту. Але для використання цих сервісів необхідно спочатку перехопи запит, ця логіка реалізована в модулі інтерсепторів, і базується вона на інтерфейсах Spring Framework. І останній розроблена логіка, яка надає можливість взаємодіяти із бібліотекою через API під час виконання програми. Ця логіка реалізована за допомогою такого інструменту Spring Framework як контролер.

Результатом магістерської роботи є програмно продукт - система підміни вихідного запиту із програми до стороннього сервісу, що має необхідний набір функціоналу для вирішення поставлених перед нею завдань. Бібліотека реалізована за допомогою Spring Framework.

Дана модель бібліотеки підміни запиту є прототипом або основою для більш комплексного рішення. Вона має такі очевидні мінуси, як відсутність гнучкої конфігурації, система не надає можливість зберігати моки в разі виключення програми. Бібліотека розроблена для підтвердження працездатності такого інструменту.

ПЕРЕЛІК ПОСИЛАНЬ

1. HTTP: The Definitive Guide автор Браян Тотті і Девід Гурлі 2002
2. HTTP/2 in Action автор Баррі Поллард 2019
3. Learning HTTP/2: A Practical Guide for Beginners автор Стівен Людін і Хав'єр Гарса 2017
4. High Performance Browser Networking автор Ілля Григорик 2013
5. HTTP Pocket Reference: Hypertext Transfer Protocol автор Клінтон Вонг 2000
6. REST API Design Rulebook Автор: Mark H. Massé 2011
7. RESTful Web APIs: Services for a Changing World автор Леонард Річардсон і Сем Рубі 2013
8. RESTful API Design автор Маттіас Біїл 2016
9. Designing Web APIs: Building APIs That Developers Love автори Амір Шеват, Бренда Цзінь, Саурабх Сахні 2018
10. Hands-On RESTful API Design Patterns and Best Practices автори Петуру Радж і Харіхара Субраманян 2019
11. Spring in Action автор Craig Walls 2021
12. Spring Boot in Action Craig Walls 2021
13. Spring Microservices in Action John Carnell 2020
14. Reactive Spring Kindle Edition Josh Long 2020
15. Spring Security in Action Laurentiu Spilca 2020
16. Expert Spring MVC and Web Flow by Seth Ladd, Darren Davison, Steven Devijver, Colin Yates 2006
17. Spring 5 Recipes — A problem-solution approach by Marten Deinum , Daniel Rubio , Josh Long 2017
18. ProfessionalJava Development with the Spring Framework by Rod Johnson 2005
19. Pro Spring 5: An In-Depth Guide to the Spring Framework and Its Tools 5th ed. Edition by Iuliana Cosmina, Rob Harrop, Chris Schaefer , Clarence Ho 2017
20. Spring Boot: Up and Running: Building Cloud Native Java and Kotlin Applications 1st Edition by Mark Heckler 2021

ДЕМОНСТРАЦІЙНІ МАТЕРІАЛИ (Презентація)

Об'єкт дослідження: процес створення бібліотеки мовою програмування Java.

Предмет дослідження: підхід підміни REST, SOAP запиту до певного сервісу

Мета роботи: дослідити процес створення бібліотеки мовою програмування Java та створити прототип бібліотеки , ціллю якої є підміна відповіді на REST, SOAP запит до певного сервісу.

Завдання:

1. Аналіз протоколів передачі даних для mocking бібліотеки
2. Розробка responce mocking бібліотеки

Вступ

В сучасних умовах все більше сторонніх сервісів використовуються при розробці програмних продуктів. І як правило впливати на них не має можливості, що стає справжнім викликом, коли один з таких сервісів перестаю працювати через можливі власні проблеми зі стабільність, із сервером або деякі інші технічні проблеми.

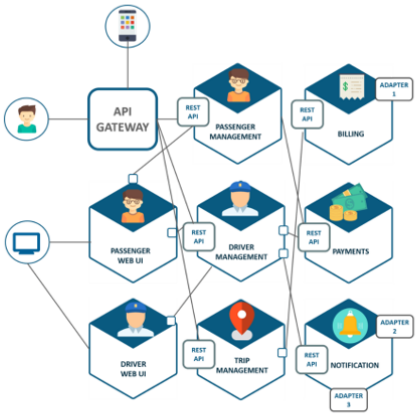


Рисунок 3.1 – мікросервісна архітектура

Виникнення таких збоїв зупиняє розробку продуктів, що спираються на такі сервіси. І в результаті цілі команди розробників заблоковані несправністю на яку вони не можуть вплинути.

Вступ

Окрім технічних негараздів існує проблема контролю даних у сторонньому сервісі. При використанні таких сервісів вони віддають ті дані, які на даний момент часу вони мають, але не завжди при розробці певного продукти необхідні актуальні дані.

Наприклад, є десять сценаріїв роботи стороннього сервісу і на ці всі сценарії розроблюваний програмний продукт має мати відповідну поведінку. І виходить, що на певний момент часу розробник може покрити один сценарій, а інші дев’ять залишать не перевіреними та не покритими необхідною логікою.

Постановка технічного завдання

Існує проблема відсутності можливості контролювати відповіді від сторонніх серверів, запити до яких надсилаються за допомогою REST або SOAP принципу.

Для вирішення цієї проблеми необхідно створити бібліотеку, яка могла надити певний інтерфейс контролю відповіді до стороннього сервісу. Дане рішення має працювати в зв’язку зі Spring Framework так як даний фреймворк являється лідером на ринку веб-додатків написаних мовою програмування Java.

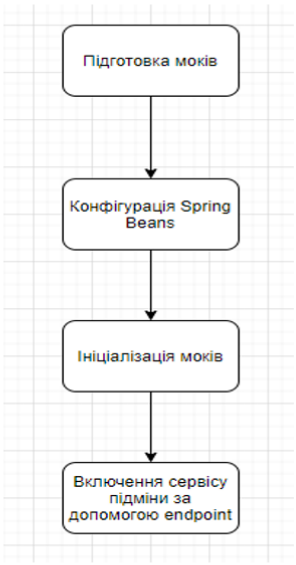
Окрім перехоплення запиту має проходити операції підміни відповіді, що має буде забезпечене сконфігурованим моком.

Також бібліотека має надавати можливість бути виключена під час виконання для можливості повернення штатного режиму роботи додатку.

Бібліотека має працювати з Spring HTTP-клієнтом, а саме RestTemplate.

Для відправлення SOAP запитів використовується WebServiceTemplate, тому бібліотека має займатися підміною запитів відправлених саме цим API.

Алгоритм взаємодії з бібліотекою



Кроки конфігурації:

— Підготовка моків у зручному вигляді для користувача, наприклад у вигляді файлів і тд. Мок із себе представляє об'єкт, який має в собі URL запиту та відповідь для REST моків або конверт SOAP запиту і відповідно відповідь на нього для SOAP моків.

— Наступним етапом необхідно сконфігурувати сутності для Spring Framework. Головною умовою на цьому етапі буде ініціалізувати правильно сутності перехоплювачів запитів.

— Далі користувач має передати в бібліотеку моки зручним йому способом через сервіс бібліотеки.

— Після попередніх етапів додаток, який використовує бібліотеку буде працювати у штатному режимі. Для її активації необхідно буде надіслати REST запит до додатку

Структура проекту

Проект складається з наступних пакетів:

— Exception – пакет в якому будуть зберігатися специфічні для бібліотеки виключення

— Interceptors – пакет для збереження класів, які будуть відповідати за перехоплення запиту

— Models – пакет для зберігання моделей, які будуть використовуватися бібліотекою

— Controllers – пакет для класу, який буде надавати можливість керувати бібліотекою

— Services – пакет для збереження класів сервісів, які будуть виконувати більшу частину логіки бібліотеки

— Utils – пакет в якому має зберігатися логіка спільна для всього проекту

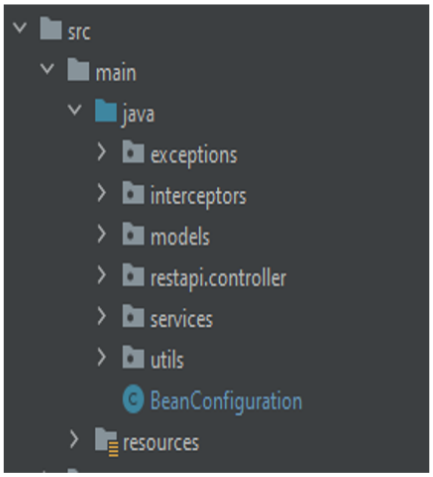


Рисунок 7.1 – структура проекту

Пакет Util

В пакеті Util створимо клас RequestMockProperties (рисунок 8.1). Цей клас має в собі таку структуру даних як мапа, кожне значення якої представляє із себе ключ та його значення. Призначення цього класу зберігання збереження пропертів для можливості зміни їх під час виконання.

Рисунок 8.1 – Клас RequestMockProperties

Пакет Models

В пакеті Models створено моделі, які будуть використовуватися надалі у всьому проекті. Насамперед створимо перерахування RequestType яке включає в собі два значення REST та SOAP, так як запити саме цих типів будуть підмінюватися.

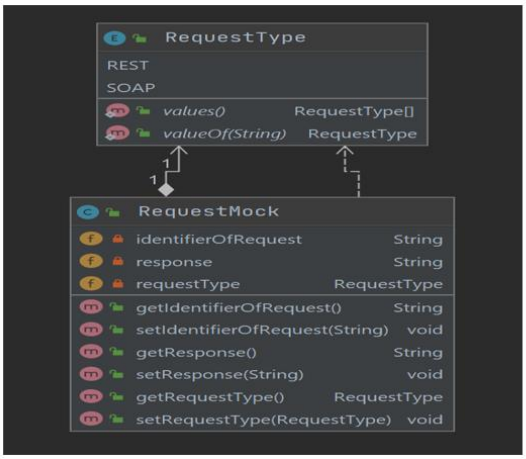


Рисунок 9.1 – Класи RequestMock та RequestType

Пакет Services

В пакеті Services розміщені класи з більшою частиною основної логіки.

Одним із сервісів буде MockServiceContainer, призначенням цього сервісу являється збереження моків для запитів

Другим сервісом являється клас RequestMockService призначенням цього сервісу є пошук та виконання моків

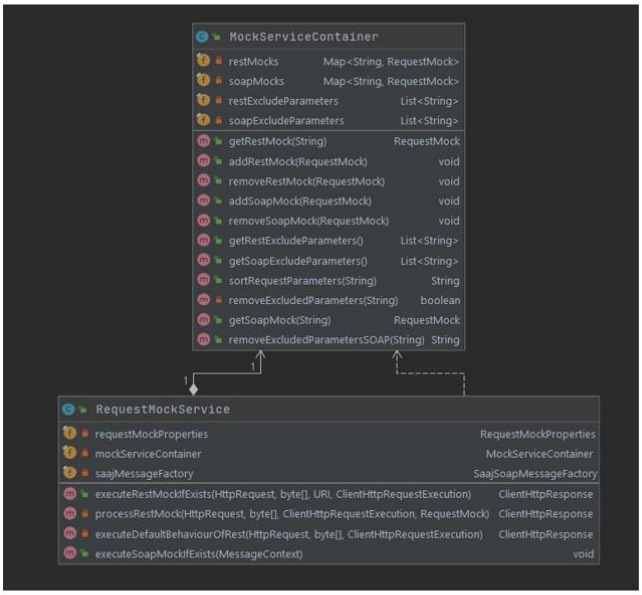


Рисунок 10.1 – Класи MockServiceContainer та RequestMockService

Пакет Interceptors

В пакеті Interceptors будуть розміщені класи, які відповідають за логіку перехоплення запитів. У більшості парадигм програмування перехоплювачі є важливою частиною, яка дозволяє програмістам контролювати виконання, перехоплюючи його.

Перехоплювач бібліотеки RestInterceptor буде викликатися для кожного вхідного запиту, і він буде перевіряти чи включена система підміни запиту. Якщо так тоді буде викликатися метод executeRestMockIfExists із сервісу RequestMockService

SoapInterceptor, клас який імплементує інтерфейс EndpointInterceptor. В SoapInterceptor як і в RestInterceptor перевіряє чи активна система підміни. Якщо так то викликається метод executeSoapMockIfExists, інакше нічого не змінюється.

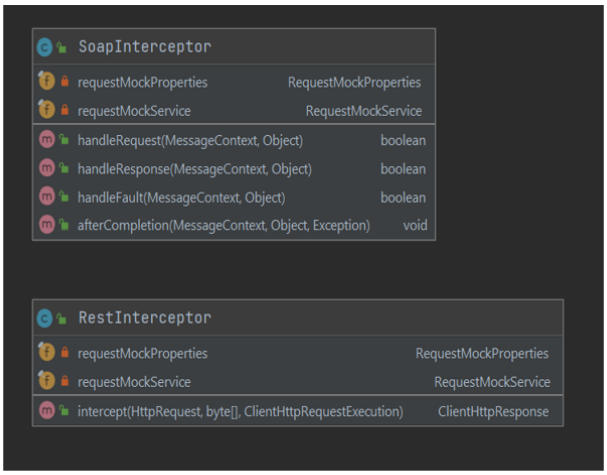


Рисунок 10.1 – Класи MockServiceContainer та RequestMockService

Пакет Api.Controller

В пакеті Controllers буде розміщений клас, який відповідає за логіку взаємодії із бібліотекою через API інтерфейс

Контролер має наступні методи:

- mockSwitch – метод для включення, виключення логіки підміни запитів
- addMock - метод для завантаження нових моків
- removeMock - метод для видалення нових моків

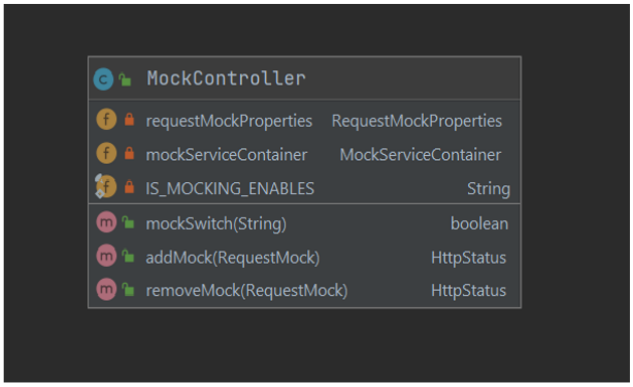


Рисунок 12.1 – Клас MockController

ВИСНОВКИ

В рамках даної випускної кваліфікаційної роботи був проведений аналіз протоколів передачі даних, визначені їх параметри та особливості. Також в рамках цього аналізу були виведені вимоги до бібліотеки, які при розробці враховувалися. На основі проведеного аналізу був визначений процес взаємодії розробника з готовою бібліотекою

Результатом магістерської роботи є програмно продукт - система підміни вихідного запиту із програми до стороннього сервісу, що має необхідний набір функціоналу для вирішення поставлених перед нею завдань. Бібліотека реалізована за допомогою Spring Framework.

Дана модель бібліотеки підміни запиту є прототипом або основою для більш комплексного рішення. Вона має такі очевидні мінуси, як відсутність гнучкої конфігурації, система не надає можливість зберігати моки в разі виключення програми. Бібліотека розроблена для підтвердження працездатності такого інструменту.