Міністерство освіти і науки України
НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ
«КИЄВО-МОГИЛЯНСЬКА АКАДЕМІЯ»
Факультет інформатики
Кафедра мережних технологій

Магістерська робота

Освітній ступінь: магістр

На тему: «Створення бібліотек на основі Spring Boot для уніфікації розробки інформаційної системи з мікросервісною архітектурою**»**

Виконав: студент 2 року навчання
Спеціальності
121 Інженерія програмного забезпечення
Скрипнік Андрій Олегович
Науковий керівник
доцент Франчук О.В.

Рецензент _. _. _.
Магістерська робота захищена
з оцінкою_____
Секретар ЕК С.А. Мелещенко
«___» _____2023

Київ 2023

Національний університет «Києво-Могилянська Академія»

Кафедра мережних технологій

ЗАТВЕРДЖУЮ

Зав. кафедри мережних технологій,

професор, д.ф.-м.н.

_____Г. І. Малашонок

“____”_____ 2023 р.

ІНДИВІДУАЛЬНЕ ЗАВДАННЯ

на магістерську роботу

студенту Скрипніку Андрію 2-го курсу факультету інформатики

ТЕМА: Створення бібліотек на основі Spring Boot для уніфікації розробки інформаційної системи з мікросервісною архітектурою

Зміст ТЧ до курсової роботи:

Анотація

Вступ

1. Проблеми розробки продуктів з мікросервісною архітектурою
2. Виділення спільних компонентів сучасних розподілених систем
3. Практична імплементація бібліотек для уніфікації та прискорення розробки системи

Висновки

Список джерел

Додатки (за необхідністю)

Дата видачі “___” _____ 2023 р.

Керівник _____ Завдання отримано _____

Календарний план виконання курсової роботи

Тема: Створення бібліотек на основі Spring Boot для уніфікації розробки інформаційної системи з мікросервісною архітектурою

№ п/п	Назва етапу курсового проекту (роботи)	Термін виконання етапу	Примітка
1.	Отримання завдання на магістерську роботу	жовтень 2022 р.	
2.	Огляд документації та літератури за темою роботи	жовтень - листопад 2022 р.	
3.	Оволодіння практичними навичками фреймворку Spring Boot	листопад - січень 2023 р.	
4.	Написання теоретичної частини курсової роботи	січень - квітень 2023 р.	
6.	Реалізація практичної частини роботи	лютий - квітень 2023 р.	
7.	Надання роботи керівнику для перевірки, демонстрація практики	квітень 2023 р.	
8.	Корегування роботи за результатами перевірки керівником	травень 2023 р.	
9.	Остаточне оформлення теоретичної частини та слайдів доповіді	травень 2023 р.	
10.	Захист магістерської роботи	червень 2023 р.	

Студент Скрипнік А.О.

Керівник Франчук О.В.

“_____” _____ _____ р.

Зміст

Анотація

Роботу присвячено дослідженню проблематики при використанні мікросервісної архітектури в розробці інформаційних систем, а також, імплементації ряду бібліотек, і пропозиції загального підходу щодо розв’язку виявлених проблем на основі фреймворку Spring Boot.

Практичним результатом роботи є 5 бібліотек, які дозволяють привести різні компоненти системи до уніфікованого вигляду в таких аспектах як: логування, обробка помилок, безпека, асинхронна комунікація, інтеграційне тестування.

Вступ

Після стрімкого росту акцій технологічних компаній в 2020-2022 роках, коли на фоні шалено зростаючої капіталізації на фондовому ринку техногіганти стрімко розширювали свій штат, настав період спаду. В кінці 2022 – на початку 2023 ай-ті світ сколихнули декілька хвиль скорочень. Наприклад, після шаленого зростання на 61% в кількості працівників за 3 роки з 2019 по 2022 [1], в січні 2023 Google оголосила скорочення 12000 співробітників, що складає приблизно 6% від штату компанії [2]. І це далеко не єдиний випадок, а тренд, що охопив всю індустрію в цілому.

Ще декілька років тому компанії змагалися за кандидатів, пропонуючи непомірно великі зарплати, і вони справді могли собі це дозволити. Зараз, на момент травня 2023 року, коли ставка Федерального Резервної Системи США складає 5%, інвестори не мають ресурсу до наповнення компаній грошима, ці самі компанії починають процес оптимізації витрат.

Час розробника став дуже дорогим. Компанії прагнуть використовувати цей час максимально ефективно. До того ж, бізнес хоче створювати конкурентний продукт, і швидко доставляти його кінцевому користувачу.

Мікросервісна архітектура стала однією з провідних технологій в розробці веб-систем. В останні роки, вона стала рішенням за замовченням як для стартапів, так і для великих стійких бізнесів. Її концепція полягає в тому, щоб розбити великі програмні додатки на незалежні компоненти, які називаються мікросервісами. Кожен мікросервіс виконує окрему функцію і може бути розроблений, розгорнутий та масштабований незалежно від інших. Переваги мікросервісної архітектури полягають у її гнучкості та масштабованості. Проте разом із перевагами з`являються і складності такі

як: налагодження координації між сервісами, версіонування та сумісність, налаштування безпеки та моніторингу.

Розв’язок цих проблем вимагає часу та координації між командами. Час розробника зараз дуже дорогий, його потрібно використовувати максимально ефективно. Якщо на початкових етапах розробки проекту не закласти стійкий фундамент у вигляді загальних для всіх команд правил та найкращих практик, то це може вилитись в великі витрати в майбутньому, та неможливість ефективно підтримувати й розвивати систему.

Метою роботи є ідентифікація компонентів сучасної складної інформаційної системи, що мають потенціал до уніфікації у вигляді окремих бібліотек, поведінка і функціонал яких може бути параметризованою. Створити бібліотеки як кінцевий продукт у вигляді програмного коду з використанням мови Java та фреймворку Spring Boot.

Текстова частина складається із трьох розділів.

У першому розділі розглядається переваги й недоліки сучасних розподілених систем, відомі патерни, що використовуються для вирішення певних проблем, та можливості Spring Boot фреймворку в контексті побудови інформаційної системи з мікросервісною архітектурою.

У другому розділі пропонується архітектура екстракції спільних частин системи в окремі модулі для уніфікації та прискорення розробки розподіленої системи. Такі речі як обробка помилок чи безпека мають бути однакові серед всіх компонентів системи для чистоти API, економії часу, зменшення дублювання коду.

Третій розділ присвячено опису реалізації бібліотек з використанням фреймворку Spring Boot. Детально розглядається функціональність кожного артефакту, можливості для параметризації в рамках окремих мікросервісів, та варіації щодо подальшого розширення функціоналу.

1 Проблеми розробки продуктів з мікросервісною архітектурою

1.1 Чому мікросервісна архітектура стала популярною

Для повного розуміння причин зростання популярності розподілених систем слід визначити, яку проблему вони вирішують. Чому в певній момент часу монолітна архітектура веб-застосунків перестала задовольняти потреби бізнесу.

Сучасні системи орієнтовані на велику кількість користувачів, тож вони мають бути надійними, масштабованими та мають легко підтримуватись [3].

Перш за все треба говорити про масштабованість. З кожним роком кількість користувачів різноманітних веб-сервісів зростає: від соціальних мереж до доставки їжі через застосунок, від онлайн банкінгу до онлайн освіти. Масштабованість монолітних застосунків може бути обмежена, оскільки вся програма повинна масштабуватися як єдине ціле. Це далеко не завжди ефективно, якщо окремі компоненти системи часто використовуються, а інші – зовсім рідко. Архітектура мікросервісів дозволяє незалежно масштабувати окремі компоненти системи, полегшуючи горизонтальне масштабування та балансування навантаження.

Говорячи про надійність, монолітний застосунок є єдиною точкою відмови. Якщо програміст зробить фатальну помилку в коді чи станеться витік пам’яті, то є ризик, що весь застосунок буде недоступним, в той час як мікросервісна архітектура дозволяє розподілити компоненти системи на різні сервери або контейнери, зменшуючи ризик збоїв [4]. У разі помилки принаймні певна частина системи залишатиметься доступною при коректному розділенні системи на компоненти та використанні відомих патернів мікросервісної комунікації.

В питанні підтримки системи є переваги й недоліки у обох типів системи. Розгортання монолітного застосунку можна налаштувати один

раз, коли кожен мікросервіс вимагатиме окремої конфігурації. Втім, це можна шаблонізувати для економії часу. Робити швидкі зміни легше робити в розподіленій системі, бо самі сервіси менші, відповідно збірка, тестування та розгортання відбуваються швидше, ніж в моноліті. Проте значний за обсягом функціонал в мікросервісах не завжди зручно робити, коли виникає питання крос-командної залежності. Якщо є необхідність одночасних змін в сервісах, що підтримуються різними командами, це може бути довго й проблематично.

В загальному, розподілена система дозволяє командам розробників працювати над окремими сервісами незалежно один від одного. Це сприяє паралельному розвитку та швидкому впровадженню нових функцій та змін. До того ж, набагато легше мати окремі невеликі репозиторії, а не одну величезну кодову базу, яку одночасно наповнює команда з 50 розробників.

Можливість використання різних мов програмування для різних компонентів системи також є однією з причин популяризації мікросервісного підходу. Розробники можуть підбирати мову, яка краще підходить під ту чи іншу задачу, що дійсно робить розв'язок бізнесових проблем більш гнучким та оптимальним в певних ситуаціях.

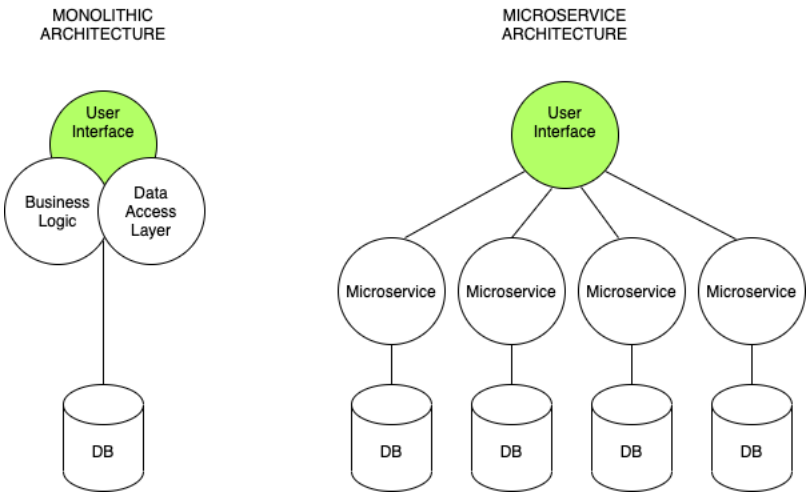


Рисунок 1.1.1 – архітектурні підходи: моноліт та мікросервіси

Отже, мікросервісний підхід асоціюється з швидкістю, гнучкістю, масштабованістю та високою доступністю – це відповідає вимогам сучасного світу, тому

Однією з перших компаній, яка почала використовувати архітектуру мікросервісів та поширила її використання, є Netflix. Процес міграції Netflix до мікросервісної архітектури розпочався близько 2009 року, коли вони зіткнулися з проблемами масштабованості та постійного розгортання в монолітній системі. Вони зрозуміли, що розділення своєї потокової платформи на менші, незалежні сервіси може допомогти їм подолати ці виклики.

Успіх мікросервісної трансформації Netflix надихнув інші компанії використовувати подібний підхід. Netflix також внесли значний внесок у розробку інструментів та фреймворків для управління та оркестрації мікросервісів, таких як Netflix OSS (Open Source Software) [5], що включає різні проекти, наприклад, Eureka для так званого “service discovery” та Ribbon для балансування навантаження.

1.2 **Проблеми, що виникають при розробці застосунку з мікросервісною архітектурою**

На жаль, розподілені системи – це не срібна куля, яка вирішує всі задачі, що постають перед бізнесом та розробниками. Зважаючи на безумовні переваги цієї архітектури, виникає ряд труднощів для її безболісного впровадження.

1. Перш за все, розбиття застосунку на багато окремих мікросервісів призводить до збільшення складності системи. Це вимагає додаткових зусиль у вирішенні проблем координації та комунікації між сервісами. Навіть просто розбити застосунок на малі компоненти є досить нетривіальною задачею. Іноді навіть

спосіб розбиття є питанням дискусій: використовувати функціональне розбиття чи розділення за доменами. Тобто проблеми починаються вже з моменту дизайну системи та створення проекту в системі контролю версій.

2. Контроль над консистентністю даних є викликом у розподілених системах. Коли певна бізнес-операція залучає кілька компонентів системи, дані можуть бути оновлені частково, якщо якийсь компонент недоступний. Відповідно до CAP теореми [6], в розподілених системах тільки дві з трьох вимог можуть бути задоволені: консистентність, доступність, стійкість до розбиття. Відповідно, дуже часто жертвують саме консистентністю. Проте це не означає, що в розподілених системах не можна покладатись на коректність даних. Зазвичай вони стають “eventually consistent”, тобто консистентними з плином певного часу. Для вирішення цієї проблеми розроблені патерни такі як: Saga, Two-phase Commit, Transactional Outbox. Проте знання цих патернів та вміння їх застосовувати вимагає високого рівня кваліфікації архітектора системи та розробників.
3. Налагодження та моніторинг сервісів може бути викликом зі зростанням кількості сервісів та їх взаємодії. Дуже важливо слідкувати за системою та отримувати сповіщення при виявленні аномалій. Контроль за часом відповіді та пропускною здатністю набагато складніший, коли в системі є 100 різних сервісів, які активно комунікують між собою.
4. У мікросервісній архітектурі важливо належним чином налаштувати версіонування API, враховуючи backward-compatibility та forward-compatibility [7]. В умовах крос-командних залежностей треба завжди брати до уваги безпечність тієї чи іншої зміни в публічному API. Якщо не належним чином керувати

версіями, можуть виникнути несумісності між сервісами та некоректна поведінка системи. Розв'язанням цієї проблеми є використання стратегій версіонування API, наприклад, використання номеру версії в url шляху, чи використовуючи content-negotiation підхід.

5. Проблема управління конфігурацією: кожен сервіс може мати власну конфігурацію, яка включає параметри, змінні середовища, налаштування автоматичного масштабування та зовнішні залежності. Управління цими конфігураціями стає складним, особливо при зміні конфігураційних параметрів та їх неузгодженому оновленні. Централізоване управління конфігураціями, використання конфігураційних файлів або конфігураційних сервісів можуть бути використані для полегшення цієї проблеми.

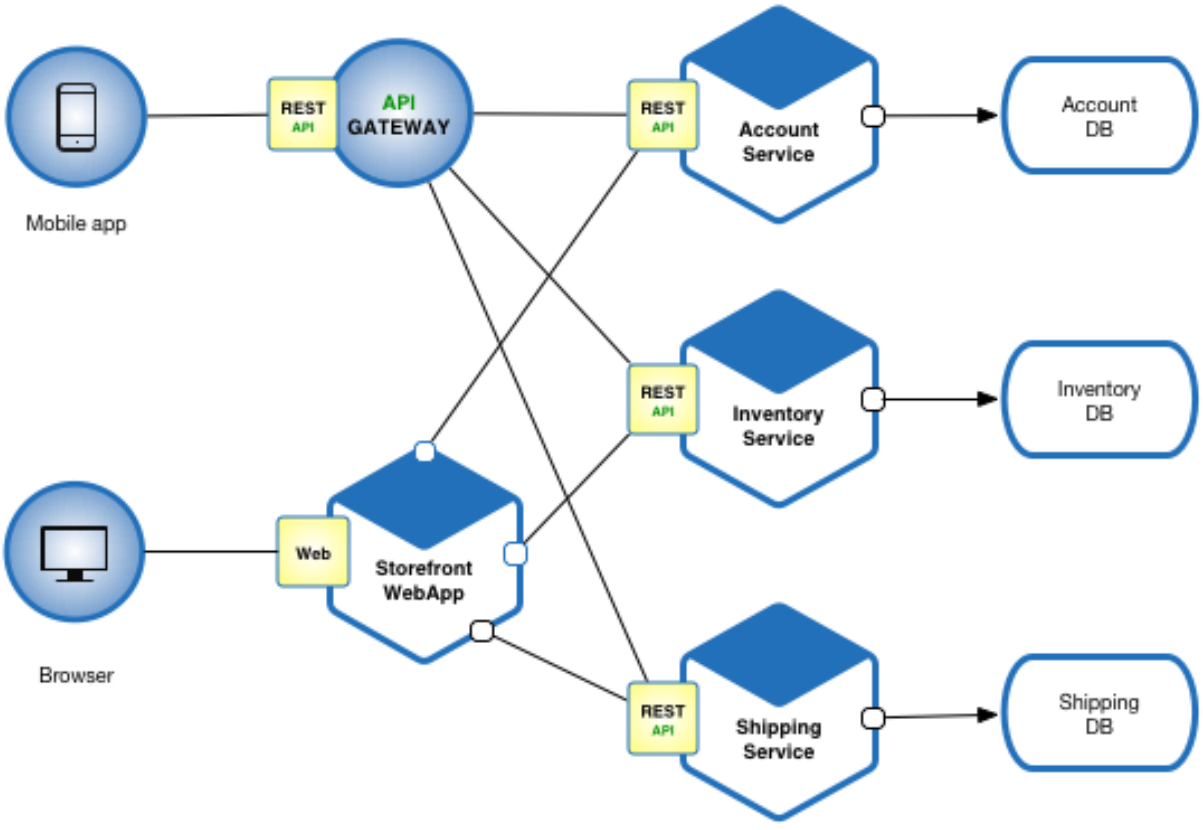


Рисунок 1.2.1 – веб-магазин з мікросервісною архітектурою [8]

Це далеко не вичерпний список проблем, що виникають при розробці застосунку, що складається з багатьох окремих компонентів. Відповідні патерни та архітектурні рішення дозволяють їх розв'язати, проте це вимагає поступової і злагодженої координації проектувальників системи, менеджерів, що визначають пріоритети, та команд. Неправильні рішення на початкових етапах роботи можуть призвести до значно більших витрат в часі і грошах в майбутньому.

1.3 **Що гальмує написання коду розподіленої системи**

Крім вищезазначених питань, треба розуміти, що розробники в рамках різних команд стикатимуться з питанням уніфікації. Проблема уніфікації в розподілених системах виникає тоді, коли різні компоненти системи мають потребу у спільній функціональності або бібліотеках, але вони не здатні ефективно обмінюватися цими ресурсами через свою розподілену природу. Основна причина виникнення цієї проблеми полягає у тому, що компоненти системи можуть бути розгорнуті на різних фізичних серверах або навіть в різних розподілених середовищах, можуть бути написані різними мовами, використовуючи різні підходи в назвах ресурсів, різні формати логування, різні формати обміну повідомленнями такі як: REST, gRPC, брокер повідомлень.

Уявімо, що команда працює над монолітним застосунком, тоді в проекті має бути модуль, що відповідає за безпеку, наприклад, перевіряє валідність токену та його підпис, якщо використовувати стандартні протоколи, такі як OAuth або JSON Web Tokens (JWT). Можливо, слід використати централізовану систему керування аутентифікацією. До прикладу, розглянемо патерн API Gateway, який являє собою єдину точку входу до системи. API Gateway буде перенаправляти запити від клієнтів до відповідних мікросервісів. Перед перенаправленням, API Gateway може

перевіряти наявність і валідність токенів доступу. До того ж там же можна керувати авторизацією.

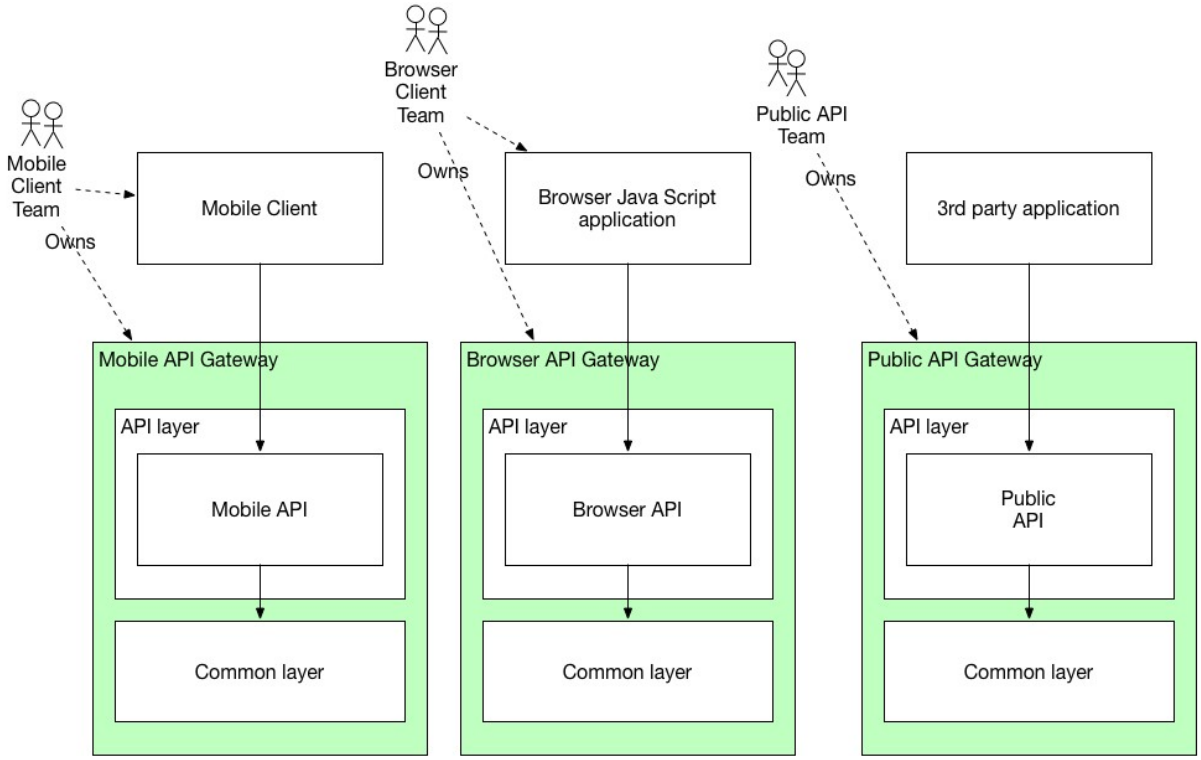


Рисунок 1.3.1 – патерн API Gateway [9]

Втім, якщо було зроблено вибір не використовувати даний патерн, як ще можна забезпечити безпеку в системі? Звісно, дублювати код перевірки токену в кожному сервісі було б нелогічно. Дуже важливо налаштувати безпеку в системі таким чином, щоб розробник певного сервісу міг сконцентруватись на безпосередніх задачах цього сервісу, і не торкався питань безпеки. Воно має бути вирішене один раз на початку проекту, уніфіковано.

Обробка помилок в розподілених системах може бути викликом, оскільки кожен сервіс може відповідати за обробку помилок власним способом. Проблема полягає в тому, як уніфікувати цей процес, щоб забезпечити єдиний стандарт обробки помилок у всій системі. Кожна окрема команда не має витрачати час на написання шаблонів чи структурних класів для помилок. Необхідна стандартизація структури

помилок. Це може бути JSON-об'єкт з полями, такими як "повідомлення", "код помилки", "деталі", "стек викликів" тощо [10]. Ця структура дозволить однорідно представляти помилки у всіх сервісах, що буде зручним та зрозумілим для клієнтів API.

Також критично важливим є питання логування (журналювання), яке має бути централізованим та в єдиному форматі. Хоча розподілена система має кілька незалежних компонентів, в плані логування вони мають бути зорганізовані в єдиний формат для однорідності та спрощення обробки даних. Для централізованого журналювання необхідно мати механізм збору журнальних даних з усіх мікросервісів. Це може бути досягнуто за допомогою спеціалізованих систем для збору журналів, таких як ELK Stack (Elasticsearch, Logstash, Kibana), Graylog, Splunk тощо. Звісно, розробник не має налаштовувати цей процес для кожного нового сервісу.

Налаштування комунікації між сервісами може сильно загальмувати розробку, особливо коли структура повідомлень кодується руками зі сторони як клієнту, так і серверу. Будь-яка зміна, наприклад, зі сторони серверу може призвести до непередбачуваних наслідків. До того ж це складно контролювати, якщо ці сервіси розробляються різними командами, вони мають синхронізувати свій час для цієї зміни. Інша проблема: команда А залежить від API команди В, який ще не готовий. Команда А заблокована. Одним з популярних підходів є використання схем даних або контрактів. Схема даних визначає структуру запиту у форматі, який може бути використаним як клієнтом, так і сервером. Ця схема може бути описана у форматах, таких як JSON Schema або Protobuf, або ж використовувати інші формати, які найкраще підходять для вашого випадку використання. Для асинхронного спілкування можна використати реєстр схем – централізований компонент, який зберігає та керує схемою даних.

1.4 **Чому уніфікація важлива для розробки розподіленої системи**

При розробці монолітного застосунку не виникає питань щодо підходу до обробки помилок чи логування, бо зазвичай створюється певний інтерфейс чи конфігурація, які можна використовувати при написанні нового функціоналу, не задумуючись про ці моменти. Розробник може сконцентруватись на безпосередньому функціоналі. Якщо в розподіленій системі нема загального підходу, то на кожного окремого сервісу має бути затрачений час на написання відповідного коду. Існує ризик, що різні команди по-різному бачать формат помилки, формат чи рівні логування, навіть формат HTTP запиту або відповіді може бути різним, починаючи від регістру (верблюдячий регістр чи верхній верблюдячий регістр). Від цього можуть страждати вже клієнти загального API, зазвичай фронтенд-розробники. Більше того, робота бекенд-розробників дублюється через відсутність уніфікації. Кожне дублювання – це по-перше, відступ від принципу DRY [11], а по-друге, просто витрата дорогоцінного часу програміста.

Досягти того ж самого рівня перевикористання вже написаного коду можна за допомогою винесення загальних компонентів в окремі сервіси або бібліотеки. Маючи одну загальну бібліотеку для оброки помилок і конвертації кінцевої відповіді на HTTP запит в єдиний документований формат, можна легко підтримувати всю систему цілісною і зрозумілою для клієнта. Те ж саме із логуванням: маючи єдину конфігурацію, яку можна імпортувати у вигляді бібліотеки, можна досягнути уніфікованого формату логування по всій розподіленій системі. Це лише єдиний наведений приклад покращення й пришвидшення розробки, але в складній

багатокомпонентній системі кількість коду та функцій, що потребують уніфікації тільки зростає.

Узагальнюючі, уніфікація в розподілених системах має наступні переваги:

1. Спрощення розробки: уніфікація дозволяє стандартизувати процес розробки та використання загальних компонентів, інтерфейсів, бібліотек. Це спрощує розробку нових сервісів і значно зменшує кількість коду, що потрібно написати. Внаслідок цього полегшується підтримка і розширення системи.
2. Зручність супроводу: уніфікована система має зрозумілу структуру, кожна функціональність описана в одному місці, а не розділена тонким шаром по всій системі, що дозволяє легко здійснювати супровід, виявляти та виправляти помилок, а також впроваджувати новий функціонал.
3. Масштабованість: уніфікована архітектура сприяє зручному масштабуванню системи. Стандартизовані компоненти та інтерфейси дозволяють легко додавати нові вузли, розподілені сервери або сервіси, що полегшує масштабування системи з ростом навантаження або обсягу даних.

Звісно, уніфікація стосується не тільки вузьких питань щодо написання коду, але й у більш широких проблемах. До прикладу, краще мати спільний стандарт комунікації між сервісами: SOAP, REST, gRPC. Зручніше розробляти сервіси на одній мові програмування або ж хоча б на мовах, що мають спільну платформу, до прикладу, JVM (Java, Scala, Kotlin). Використання централізованої системи керування конфігурацією, такої як Consul або ZooKeeper [12], дозволяє уніфікувати управління конфігурацією мікросервісів. Ці глобальні питання мають бути вирішені на старті проекту системними архітекторами в залежності від вимог системи.

2. Винесення компонентів розподіленої системи в бібліотеки

2.1 Можливості мови Java та фреймворку Spring Boot для побудови застосунку з мікросервісною архітектурою

Такі властивості мови програмування Java, як строга типізація та об`єктно-орієнтованість роблять її широко застосованою в багатьох сучасних сферах. Відомі світові й українські банківські додатки написані цією мовою (Privat24, Monobank, Revolut). Дуже популярною ця мова є в сфері розробки CRM- та ERP-систем, страхування, медицині, фінансах, торгових платформах та аналітичних інструментах. Враховуючи широкий спектр функціональності, сильну підтримку і велику спільноту розробників, Java є популярною мовою для розробки веб-застосунків з мікросервісною архітектурою:

- Платформна незалежність: Java працює на віртуальній машині Java (JVM), що дає можливість запускати Java-застосунки на різних операційних системах без потреби перекомпіляції.
- Багата екосистема: Java має велику та розвинену екосистему з великою кількістю бібліотек, фреймворків та інструментів. Це дозволяє розробникам ефективно будувати та підтримувати мікросервісні архітектури за допомогою готових компонентів та рішень.
- Висока продуктивність: Java відома своєю високою продуктивністю та ефективністю. Вона використовується в багатьох великих підприємствах та веб-додатках, де потрібна висока швидкодія та масштабованість.
- Міцна підтримка веб-розробки: Java має широку підтримку для розробки веб-додатків. Фреймворки, такі як Spring, Spring Boot та Java EE, надають потужні інструменти для створення веб-

застосунків з мікросервісною архітектурою. Вони дозволяють швидко та ефективно розгортати та управляти мікросервісами.

Для зручності розробки веб-застосунків у 2003 році було створено фреймворк Spring [13]. Це булі і є альтернатива традиційним підходам до розробки Java-додатків. Творець фреймворку Род Джонсон хотів створити інструмент, який би спрощував розробку, підтримку та розгортання додатків, враховуючи кращі практики індустрії. Початково Spring був заснований на принципах інверсії керування (IoC) і аспектно-орієнтованого програмування (AOP) [14]. Його основна мета полягала в тому, щоб відокремити бізнес-логіку від деталей реалізації та залежностей між компонентами, що забезпечувало більшу гнучкість, тестованість та повторне використання коду.

Spring Boot є розширенням фреймворку Spring. Він надає можливості автоматичної конфігурації, що дозволяє зосередитись на розробці функціональності. Він має вбудовану підтримку для веб-сервера Tomcat, що дозволяє запускати додатки без необхідності налаштування окремого сервера. Spring Boot також має інструменти, які спрощують роботу з розгортанням, моніторингом та управлінням додатками. Серед його ключових можливостей можна навести наступні:

- Конфігурація через код та зовнішні файли: Spring Boot дозволяє конфігурувати застосунок за допомогою коду або зовнішніх файлів (наприклад, properties або YAML файлів). Це дозволяє зручно встановлювати параметри конфігурації для кожного мікросервісу.
- Завантаження компонентів на вимогу: Spring Boot підтримує автоматичне завантаження залежностей компонентів на вимогу. Це дозволяє розбити функціональність на окремі модулі або мікросервіси і завантажувати їх при необхідності.

- Вбудована підтримка мікросервісного оточення: Spring Boot надає підтримку для стандартних протоколів та бібліотек, які використовуються в мікросервісному оточенні. Наприклад, він має вбудовану підтримку для Spring Cloud, що дозволяє використовувати такі необхідні в мікросервісній архітектурі компоненти, як service registry, load balancer, circuit breaker тощо.
- Моніторинг та керування: Spring Boot надає зручні інструменти для моніторингу та керування мікросервісами. Наприклад, модуль Spring Boot Actuator дозволяє отримувати метрики, інформацію щодо кількості потоків, відсоткового значення використаної пам`яті тощо.
- Spring Boot CLI: дозволяє розробникам ефективно працювати з Spring Boot за допомогою простого та потужного інструменту командного рядка, прискорюючи розробку, розгортання та відлагодження додатків.

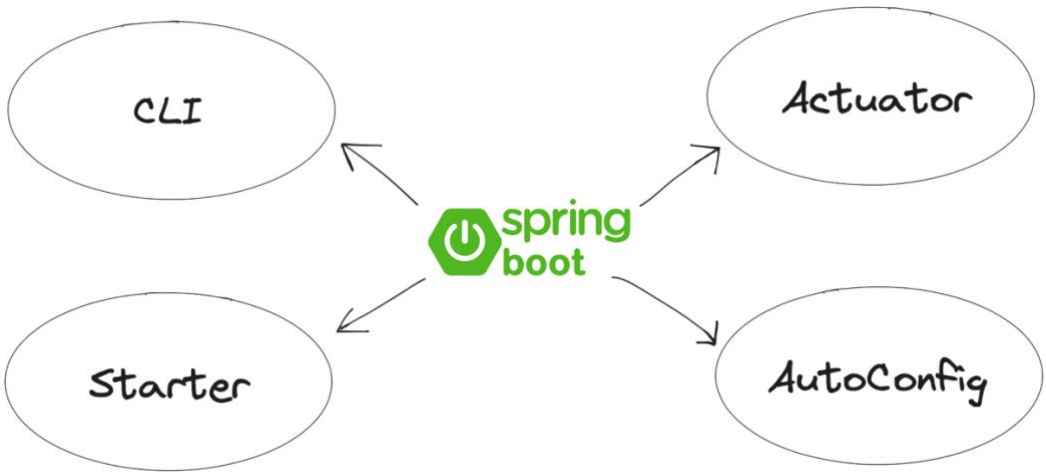


Рисунок 2.1.1 – комоненти Spring Boot

Окремо слід зупинитися на Spring Boot стартерах. Це залежність, яку можна підключити в проект, і яка містить необхідні бібліотеки, конфігурацію та інші ресурси для розробки додатку з використанням Spring

Boot. Стартер спрощує процес налаштування та підключення різних модулів і функціональності: надає готове початкове налаштування для різних компонентів, що необхідні в розробці. Наприклад, spring-boot-starter-web містить необхідні бібліотеки і конфігурацію для розробки веб-додатків, spring-boot-starter-data-jpa надає підтримку роботи з базою даних через JPA, а spring-boot-starter-test додає необхідні засоби для написання тестів.

Завдяки Spring Boot стартерам розробникам не потрібно вручну підключати кожну окрему залежність, вказувати конфігурацію та вирішувати конфлікти версій. Вони можуть просто включити в проект потрібні стартери, і Spring Boot самостійно налаштує необхідну функціональність. Крім вбудованих стартерів, також можна створювати власні стартери для специфічних потреб проекту або бібліотеки. Власні стартери дають можливість згрупувати пов'язані залежності та конфігурацію, щоб іншим розробникам було легше використовувати вашу бібліотеку. Загалом, Spring Boot стартери спрощують розробку додатків на основі Spring Boot, забезпечуючи та швидке налаштування необхідної функціональності, що економить час та зусилля розробника.

2.2 Єдина конфігурація логування та вирішення проблеми розподіленого трасування

Налаштування логування в Spring Boot може бути здійснено за допомогою конфігураційних файлів, таких як application.properties або application.yml. У цих файлів можна визначити рівень логування, формат повідомлень та місце, куди логи будуть записуватись.

Найпопулярнішими фреймворками для цієї задачі в екосистемі Spring Boot є Logback та Log4j2 [15]. Обидва фреймворки мають схожі конфігураційні можливості. Однак, Logback надає більш простий та прямолінійний спосіб налаштування за допомогою файлу logback.xml. З

іншого боку, Log4j2 надає більш гнучкі та розширені можливості конфігурації та в цілому вважається більш продуктивним та оптимізованим.

Очевидно, щоб формат повідомлень, рівень логування та інші конфігурації були уніфіковані, та могли бути оброблені централізованим рішенням для аналізу логів, наприклад ELK (Elastic, Logstash, Kibana), розробники мають використовувати певне єдине джерело, де прописана конфігурація логування. Для цієї задачі чудово підходить відповідний Spring Boot стартер, який можна створити на старті проекту, і потім включати у будь-який новий мікросервіс.

У контексті мікросервісної архітектури Spring Boot, важливим є питання розподіленого трасування (distributed tracing) [16]. Для вирішення проблем в системі, для аналізу часу виконання запиту, і в цілому для дослідження поведінки системи необхідно мати інструмент для стеження за шляхом виконання запитів, які проходять через різні мікросервіси.

В екосистемі Spring Boot існує рішення Spring Cloud Sleuth [17] - це бібліотека, яка надає підтримку для розподіленого трасування в мікросервісній архітектурі. Sleuth дозволяє стежити за шляхом виконання запиту. Sleuth генерує унікальний ідентифікатор (Trace ID) для кожного запиту, який проходить через систему. Цей ідентифікатор передається через HTTP заголовки або контекст трасування, що дозволяє ідентифікувати запит та його пов'язані взаємодії.

Проте якщо потрібно виконати більш специфічні завдання або здійснити додаткові налаштування, написання власного стартера може бути кращим варіантом. Тоді можна мати повний контроль над функціоналом та можливістю налаштування стартера відповідно до потреб проекту.

Розробники великих систем стикаються з поняттям Mapped Diagnostic Context (MDC) - це механізм, який дозволяє прив'язати контекстну (ідентифікатор сесії, ідентифікатор користувача, ім’я сервісу, trace id) інформацію до потоку виконання в програмі. Він використовується в

основному в логуванні для додавання додаткових даних до журналів, що відносяться до конкретного потоку або операції.

Маючи власну бібліотеку з налаштуваннями логування для різних середовищ проекту (девелопмент, тест, продакшн), та відповідними класами, що наповнюють Mapped Diagnostic Context, можна уніфікувати та спростити процес логування та трасування, й не витрачати на нього час в ході активної розробки бізнес-логіки.

2.3 Єдина конфігурація безпеки в мікросервісній архітектурі

Безпека – ключова складова сучасної інформаційної системи. Серйозні проекти з різних індустрій мають певні стандарти, без дотримання яких вони не можуть функціонувати. Наприклад, в сфері платіжних систем існує стандарт PCI DSS (Payment Card Industry Data Security Standard) [18]: це стандарт безпеки, розроблений і прийнятий групою платіжних систем (Visa, MasterCard, American Express, Discover, JCB) з метою захисту персональних даних власників платіжних карт. PCI DSS встановлює вимоги щодо захисту даних про картки, обробки платежів, мережевої безпеки та інших аспектів безпеки платіжних систем.

HIPAA (Health Insurance Portability and Accountability Act) [19]: Цей стандарт встановлює вимоги до безпеки та конфіденційності медичних інформаційних систем і персональних медичних даних в США. HIPAA захищає права пацієнтів та встановлює вимоги до захисту медичних записів та конфіденційності.

Існують й більш широко поширені стандарти, як GDPR (General Data Protection Regulation) [20]: Це регулятивний стандарт, який регулює захист персональних даних громадян Європейського Союзу. GDPR встановлює правила збору, зберігання, обробки та передачі персональних даних, а також права користувачів на захист своїх даних.

Навіть не беручи до уваги встановлені законом стандарти, які можуть бути й не застосовані до конкретного проекту, якщо він розробляється в сфері, яка не є врегульованою, існують певні правила та норми, які є обов'язковими для сучасних систем: використання SSL/TLS для створення шару захисту транспортного рівня, використання хешування та солі для безпечного зберігання паролів, захист від SQL-ін'єкцій, використання двофакторної автентифікації. В цілому, чим безпечніша система, тим більша в неї довіра від користувачів.

Автентифікація – стандартна задача для сучасних систем. Зазвичай користувачі автентифікуються за допомогою імені і паролю, або за допомогою соціальних мереж (Google, Facebook, LinkedIn) із використанням технології Oauth2 [21].

Виникає проблема, як сервісам системи, що відповідають за бізнес логіку, визначити, що користувач дійсно автентифікований. Навіть якщо з кожним запитом передається токен автентифікації, необхідно перевірити його цілісність, чи не вийшов строк його дії. Одним із варіантів є використання спеціального сервісу – API Gateway – який матиме відповідальність перевірки токену, і відповідно за результатом цієї перевірки контролювати доступ до ресурсів.

Проте не у всіх проектах використовується API Gateway і не завжди він може бути оптимальним рішенням проблеми. В окремих випадках, коли вимоги до продуктивності та масштабованості є дуже високими, може бути доцільним уникати зайвого навантаження на API Gateway шляхом виконання валідації токенів в окремому сервісі або компоненті. Також коли логіка автентифікації та авторизації досить складна та потребують виконання специфічного функціоналу, краще викликати цей функціонал безпосередньо в основному додатку або сервісі, а не через API Gateway.

Одним із варіантів розв'язку проблеми уніфікованого механізму автентифікації може бути бібліотека, яка може імпортована в будь-який

сервіс, що має бути відкритим до зовнішнього світу та вимагає забезпечити доступ до певного функціоналу тільки автентифікованим користувачам системи. Перш за все, ця бібліотека може перевіряти цілісність токену та час його дії. Якщо припустити, що токен представлений у вигляді JWT, то алгоритм може виглядати наступним чином:

- Розкодування токену, витягнувши з нього заголовок (header), тіло (payload) та підпис (signature).
- Перевірка підпису секретним ключем у випадку синхронного шифрування або публічним ключем при асинхронному шифруванні. Це дозволяє перевірити цілісність токену та забезпечити, що він не був змінений після підпису.
- Перевірка часу дії токену.
- Перевірка інших атрибутів: додаткова перевірка наявності та валідності інших атрибутів, таких як видавець (issuer), аудиторія (audience) та інші, які можуть бути присутніми в токені.

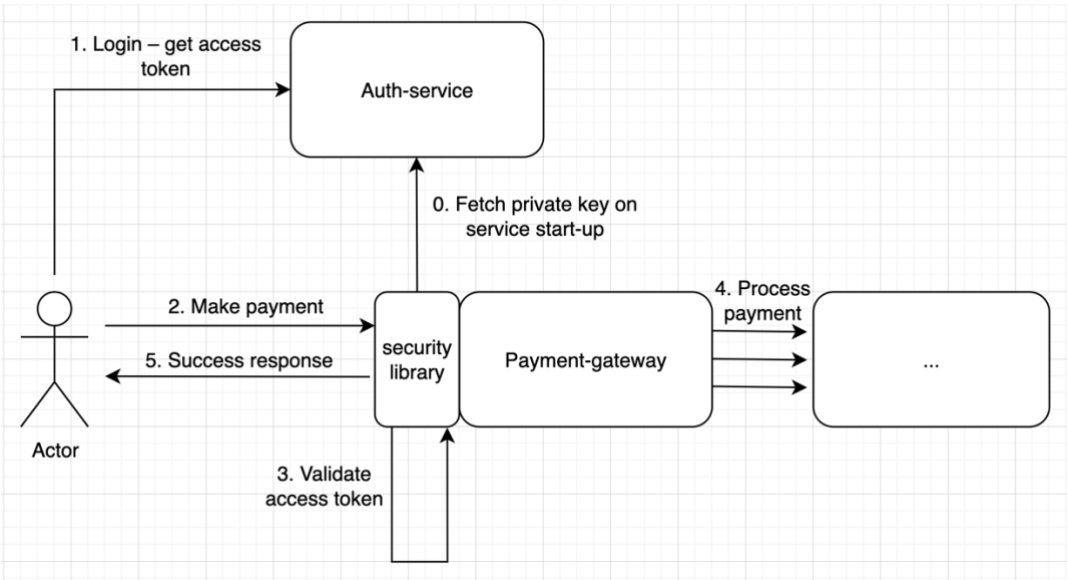


Рисунок 2.3.1 – використання бібліотеки для автентифікації

Окрім цього, бібліотека може містити зручну можливість для конфігурації доступу до ендпоінтів сервісу: чи потрібен токен для виконання

запиту до ендпоінту, чи він доступний і без токену, тобто є частиною API, що не має вимог стосовно безпеки.

Звісно, окрім вищезазначених переваг, після обробки токену, можна витягнути всю інформацію, що була в ньому закодована, наприклад, ідентифікатор користувача, та використовувати її в ході виконання програми.

Узагальнюючи, підхід до безпеки в проекті має бути уніфікований певним чином, бо це дуже важлива складова системи. Використовувати централізований підхід (API Gateway), виокремлювати функцію в окрему бібліотеку чи, можливо, йти третім шляхом – вибір, який треба зробити на початкових етапах розробки архітектури системи.

2.4 Налаштування уніфікованої структури помилок

Коли різні команди розробляють різні сервіси в рамках однієї розподіленої системи, дуже важливо мати єдиний формат помилок для зручного користування API сервісів як всередині системи, так і зовнішніми клієнтами. Неоднорідність контракту ускладнює взаємодію з різними сервісами системи, та в цілому робить код менш зрозумілим та розширюваним. Особливо може страждати клієнтська частина проекту, якщо вона є, бо веб або мобільним розробникам знадобиться ускладнювати клієнтську логіку різними модифікаціями класів обробників API помилок.

Проте навіть домовившись про єдиний контракт помилок, про перелік бізнесових кодів помилок, про структуру об’єкту помилки, може виникнути складність для розробників бекенду імплементувати цю структуру без наявності єдиного шаблону. Кожен мікросервіс – окрема

програма, в ході роботи яких можуть виникнути помилки, які треба обробляти під уніфікований формат в рамках цього ж сервісу. Для зменшення повторюваного коду та для зручності розробки можна винести загальний повторюваний код обробки помилок та код, що формує структуру помилки, в окрему бібліотеку.

Для ефективної роботи над цією важливою частиною API проекту слід слідувати наступним крокам:

- Визначення структури помилки: необхідно створити стандартну модель або клас, який описує структуру помилки. Цей клас може містити поля, такі як код помилки, повідомлення про помилку, додаткові деталі або стек виклику. Цю структуру можна запрограмувати в бібліотеці, написати зручні функції побудови цього об’єкту в залежності від класу винятку (exception) та перевикористовувати в різних сервісах системи.
- Використання HTTP статусів. HTTP статуси є стандартом для передачі інформації про тип помилки. Наприклад, статус 400 (Bad Request) означає, що запит був некоректний, або статус 500 (Internal Server Error) – це внутрішня помилка сервера.
- Документація: детальна документація про контракт помилки, включаючи структуру та значення полів, використання HTTP статусів та можливі повідомлення про помилку. Це допоможе розробникам сервісів коректно обробляти та відповідати на помилки.
- Версіювання контракту: коли потрібно внести зміни до контракту помилки, звісно потрібно закладати можливість версіювання. Це дозволяє підтримувати сумісність з попередніми версіями контракту та забезпечує поступове оновлення сервісів. Це стосується не тільки обробки помилок, а в цілому написання API.

2.5 Налаштування асинхронної комунікації між сервісами на прикладі використання Kafka

Спосіб комунікації між сервісами в розподіленій системі – неодмінна частина архітектури системи. Окрім стандартного синхронного спілкування за допомогою HTTP протоколу, популярним варіантом є асинхронне спілкування через брокери повідомлень.

Основна ідея брокера повідомлень полягає в тому, що відправник не прямо взаємодіє з одержувачем, а надсилає повідомлення до брокера, а потім брокер передає його одному або декільком одержувачам. Це дозволяє будувати слабко зв’язану систему та забезпечити асинхронну комунікацію між компонентами системи. Серед найвідоміших брокерів повідомлень можна навести приклади Apache Kafka та RabbitMQ. Kafka [22] є потоковою платформою, спрямованою на обробку великого обсягу даних з високою швидкістю та здатністю зберігання повідомлень на довготривалий термін. У свою чергу, RabbitMQ централізована черга повідомлень, яка надає надійну доставку повідомлень та підтримує різні протоколи спілкування, включаючи AMQP. Kafka працює на основі моделі публікації-підписки (publish-subscribe), в той час як RabbitMQ базується на моделі черги повідомлень (message queue).

Асинхронна комунікація надає декілька переваг [23]:

- Масштабованість: кожен сервіс може працювати незалежно та не блокувати інші сервіси. Це дозволяє масштабувати кожен сервіс окремо, забезпечуючи кращу продуктивність та використання ресурсів.
- Резистентність до помилок: асинхронна комунікація дозволяє обробляти помилки та відновлювати роботу без прив'язки до прямого взаємодії між сервісами. Якщо сервіс, що отримує

повідомлення, недоступний, повідомлення може бути збережено та оброблено пізніше, коли сервіс стає доступним.

- Зменшення залежності: асинхронність дозволяє сервісам працювати незалежно один від одного. Вони можуть розроблятися, розгортатися та масштабуватися окремо, не впливаючи на роботу інших сервісів.

Kafka досить просто інтегрується в проекти на основі фреймворку Spring Boot, якщо використовувати стартер spring-kafka. Проте із появою додаткової системи постає питання її коректного налаштування та забезпечення безперервної роботи.

Якщо пропустити проблеми пов’язані із налаштуванням та розгортанням Kafka кластеру, а сконцентруватись на помилках, які можуть виникати безпосередньо в сервісах, що продукують та поживають повідомлення, то можна виділити основні:

- Серіалізація та десеріалізація даних та їх версіонування: при передачі повідомлень між сервісами потрібно забезпечити безпомилкову десеріалізацію повідомлення в об’єкт на стороні отримувача. Для забезпечення цього використовується реєстр схеми повідомлення та його версіонування.
- Обробка помилок: коли сервіси спілкуються асинхронно, важливо правильно обробляти помилки. Недоставлені повідомлення, помилкові формати даних або інші проблеми повинні бути виявлені та оброблені. Є багато варіантів: stop-the-world стратегія при виявленні помилки, механізм повторюваних (retryable) черг, механізм dead-letter черг, міксовані стратегії.
- Розподілене трасування: потрібно забезпечити наявність Trace Id в повідомленнях, щоб мати змогу відслідковувати шлях як синхронних, так і асинхронних операцій в системі.

Бібліотека із заздалегідь прописаною схемою повідомлень, та стратегіями обробки помилок може зекономити час розробників, що використовують асинхронну комунікацію між сервісами. Класи, що описують структуру повідомлення, та їхні JSON (або іншого формату) схеми таким чином беруться з єдиного джерела. А вибір стратегії для кінцевого користувача може зводитись до визначення відповідного параметру в конфігураційному файлі application.yaml.

Trace Id може бути доданий, наприклад, до хедеру Kafka повідомлення, тож є сенс винести відповідний код до загальної бібліотеки для уніфікації та уникнення його дублювання в кожному сервісі.

2.6 Використання API-first підходу для вирішення проблеми крос-командних залежностей

При розробці складної розподіленої системи, кожна команда відповідає за певний набір сервісів, що є частиною певного домену. Втім, кожен домен по окремості не може функціонувати, і залежить від інших доменів. Спілкування між сервісами різних доменів відбувається синхронно чи асинхронно. Коли команда розробляє новий сервіс А, що потребує використання даних іншого домену, і сервіс В іншого домену працює в продакшині, то це не є проблемою для сервісу А використати API сервісу В. Проте якщо це велика й нова функціональність, яка має бути розробленою в короткі терміни, і сервіс В не є готовим, а має бути написаним з нуля, то виникає сильна залежність першої команди від другої.

Очевидно, що в такому випадку серйозна перевага мікросервісної архітектури, а саме – можливість розробки різних сервісів різними командами паралельно – нівелюється.

В монолітному застосунку можна створити окремий модуль, прописати інтерфейси, і розробляти імплементацію. Клієнт не переймається

щодо деталей імплементації, йому достатньо мати інтерфейс, який він може використовувати для розробки своєї частини програми. В цьому випадку можна організувати паралельну роботу двох команд, і не втрачати дорогоцінний час.

Втім, в розподілених системах теж існує підхід до написання інтерфейсів. Проте цей інтерфейс має вигляд не синтаксичної конструкції певної мови програмування, а найчастіше – OpenAPI специфікації [31], що визначає контракт взаємодії з сервісом. В OpenAPI специфікації описується:

1. Заголовок, версія, опис API.
2. Ендпоінти: доступні маршрути та їх шляхи (URL-посилання), параметри, HTTP-методи (GET, POST, PUT, DELETE) та типи даних, які можуть бути використані.
3. Параметри, які можуть бути передані у запиті до API, такі як параметру шляху, тіло HTTP-запиту тощо, заголовки (headers) тощо.
4. Відповіді: Вказує, які види відповідей можуть бути отримані від API, включаючи HTTP-коди статусу та типи даних, які повертаються.
5. Схеми даних: Визначає структуру та типи даних, які використовуються в запитах та відповідях API, включаючи об'єкти, масиви, рядки, числа тощо. Зазвичай використовується формат JSON Schema для опису схеми даних.
6. Безпека: Вказує, як забезпечена автентифікація та авторизація для доступу до API.

OpenAPI специфікація є важливим інструментом для документування та розробки API, вона сприяє зрозумілості, стандартизації та взаємодії між розробниками.

Маючи OpenAPI специфікацію сервісу B, обидві команди можуть стартувати роботу над реалізацією своєї частини функціональності. Тобто

сильна залежність першої команди від другої тепер не є проблемою, і робота може бути виконана паралельно.

Більше того, в контексті Spring Boot веб-сервісів OpenAPI специфікація може бути використана для генерації коду як клієнтської, так і серверної частини. Для цього можна використати спеціальний Maven плагін "openapi-generator-maven-plugin". Це дозволить команді, що розробляє сервіс А, не писати власноруч клас-клієнт для API сервісу В. Водночас команда, яка розробляє сервіс В, може не писати з нуля код для контролерів, що обробляють HTTP-запити. Це значно економить час на механічну розробку, і дозволяє сконцентруватись на бізнес-логіці.

Загалом, використання OpenAPI специфікації під час розробки мікросервісної системи зменшує крос-командні залежності, економить час розробників, дозволяє використати код-генерацію, чітко документує функціональність сервісів та значною мірою полегшує тестування системи.

2.7 Уніфікований підхід до інтеграційного тестування компонентів розподіленої системи

Продовжуючи думку із попереднього розділу, контракт сервісу має бути не тільки описаний, а й належним чином протестований. В процесі CI/CD важливо мати крок інтеграційного тестування API для забезпечення якості системи й швидкого виявлення можливих багів та помилок. В екосистемі Spring Boot є зручні інструменти для написання тестів для перевірки поведінки API.

Інтеграційне тестування контролерів зазвичай здійснюється за допомогою спеціального фреймворку для тестування, такого як JUnit або TestNG, разом з фреймворком для тестування HTTP-запитів, таким як MockServer, MockClient чи WebClient.

При інтеграційному тестуванні контролерів, тест виконує HTTP-запити до API, передає різні параметри запиту, різні варіації тіла запиту, перевіряє коди статусу відповідей, зчитує та перевіряє тіло відповідей.

Інтеграційне тестування контролерів дозволяє переконатись, що ендпоінти працюють правильно, їхня поведінка відповідає очікуванням та немає непередбачених помилок в процесі взаємодії з API. Це допомагає забезпечити якість та надійність додатку перед його розгортанням в продакшн середовищі.

Для імітації роботи власного сервісу або роботи зовнішніх залежностей власного сервісу можна використати MockServer. За допомогою MockServer можна налаштовувати очікувані HTTP-запити та відповіді, які будуть повертатися під час тестування.

MockClient, з іншого боку, це спеціальний клієнт, який використовується для створення HTTP-запитів до вашого сервісу. MockClient використовується для створення запитів з певними параметрами, методами та заголовками, які ви очікуються від реальних клієнтів. Це дозволяє перевіряти, як сервіс обробляє запити і повертає відповіді, без реальної взаємодії з залежностями.

Це не єдині інструменти для проведення інтеграційного тестування, проте вони потребують конфігурації перед їх використанням. Ця конфігурація, а також створення HTTP-запитів, перевірка коректності HTTP-відповідей – це те, що має бути продубльовано фактично в кожному сервісі, що має інтеграційні тести в CI/CD. Звісно, ця робота може бути виконана один раз й винесена в окрему бібліотеку, щоб розробник, який займається інтеграційним тестом, міг перейматися лише за функціональну частину цього тесту – створення HTTP-запиту до та перевірка відповіді.

Хоча уніфікація коду інтеграційного тестування може здаватися не першочерговим завданням, проте це також економить час розробника на написання нових й підтримку існуючих тестів.

3. Практична реалізація бібліотек з використанням можливостей Spring Boot

3.1 Реалізація бібліотеки «common-logging»

Всі бібліотеки написані з використанням мови програмування Java версії 17 та фреймворку Spring Boot версії 3. Самі бібліотеки являють собою стартери в термінології Spring Boot стартери – зручні та легко конфігуровані пакети, що включають необхідні залежності для використання певної функціональності.

Для реалізації бібліотеки, що надає можливість налаштовувати логування та вирішує проблему розподіленого трасування слід вирішити, який фреймворк використовувати. Найвідоміші – Logback та Log4j2. Для прикладу взято Log4j2.

log4j2-prod.yaml

```yaml
configuration:
  status: info

appenders:
  - console:
      name: ConsoleAppender
      target: System.out
      jsonLayout:
        complete: false
        compact: true
        eventEol: true
        properties: false
        additionalFields:
          - key: correlationId
            value: '%X{correlationId}'

loggers:
  - root:
      level: info
      appender-ref:
        - ref: ConsoleAppender

  - logger:
      name: com.skrypnik
      level: info
      additivity: false
      appender-ref:
        - ref: ConsoleAppender
```

log4j2-local.yaml

```yaml
configuration:
  status: debug

  appenders:
    - console:
        name: ConsoleAppender
        target: System.out
        layout:
          pattern: "%d{HH:mm:ss.SSS} [%thread] %-5level %logger{36} - %msg%n"

  loggers:
    - root:
        level: info
        appender-ref:
          - ref: ConsoleAppender

    - logger:
        name: com.skrypnik
        level: debug
        additivity: false
        appender-ref:
          - ref: ConsoleAppender
```

Рисунок 3.1.1 – конфігурація логування Log4j2

Для конфігурації формату логування із фреймворком Log4j2 використовуються файли в "yaml" форматі. На рисунку 3.1.1 наведено приклад різних конфігурацій для локального середовища та продакшн середовища. В локальному середовищі логи виводяться в форматі стрічки, в продакшн – в форматі JSON. Також відрізняється мінімальний рівень логування: для локального – "debug", для продакшн – "info".

Задля наповнення Mapped Diagnostic Context (MDC) даними про ідентифікатор трасування (Trace id або Correlation id), версію і ім’я сервісу та іншими даними, що важливі в контексті логування, використано OncePerRequestFilter – функціонал Spring Boot, який дозволяє включати додаткову логіку перед обробкою HTTP-запиту в класі контролеру. У випадку common-logging бібліотеки перед обробкою кожного HTTP-запиту заповнюється MDC. Якщо Correlation id присутній в заголовку запиту, то його значення береться із заголовку, якщо відсутній – то генерується випадкове значення. Також є можливість не виконувати цю операцію для таких шляхів, як "/health", які регулярно викликаються системою оркестрації контейнерів чи іншими інфраструктурними об’єктами для перевірки стану сервісу. Заповнювати MDC в цьому випадку немає сенсу.

```java
@Component
@Slf4j
@RequiredArgsConstructor
@Order(Ordered.HIGHEST_PRECEDENCE)
public class HttpRequestLoggingFilter extends OncePerRequestFilter {

    1 usage
    private static final List<String> NOT_LOGGING_URLS_PATHS = List.of( e1: "/health");

    private final MappedDiagnosticContext mappedDiagnosticContext;

    /NullableProblems/
    @SneakyThrows
    @Override
    protected void doFilterInternal(HttpServletRequest request, HttpServletResponse response, FilterChain filterChain) {
        boolean skipLogging = NOT_LOGGING_URLS_PATHS.stream()
                .anyMatch(uri -> request.getRequestURL().indexOf(uri) != -1);
        if (!skipLogging) {
            mappedDiagnosticContext.populateMDC(request);
        }
        filterChain.doFilter(request, response);
    }

}
```

Рисунок 3.1.2 – Spring Boot фільтр для заповнення MDC

```java
public void populateMDC(HttpServletRequest request) {
    populateHostData();
    setServiceName(serviceConfig.getName());
    setServiceVersion(serviceConfig.getVersion());
    setEnvironment(deploymentConfig.getEnv());
    setCorrelationId(Optional.ofNullable(request.getHeader(X_CORRELATION_ID_HEADER))
            .orElse(UUID.randomUUID().toString()));
}
```

Рисунок 3.1.3 – код заповнення MDC даними

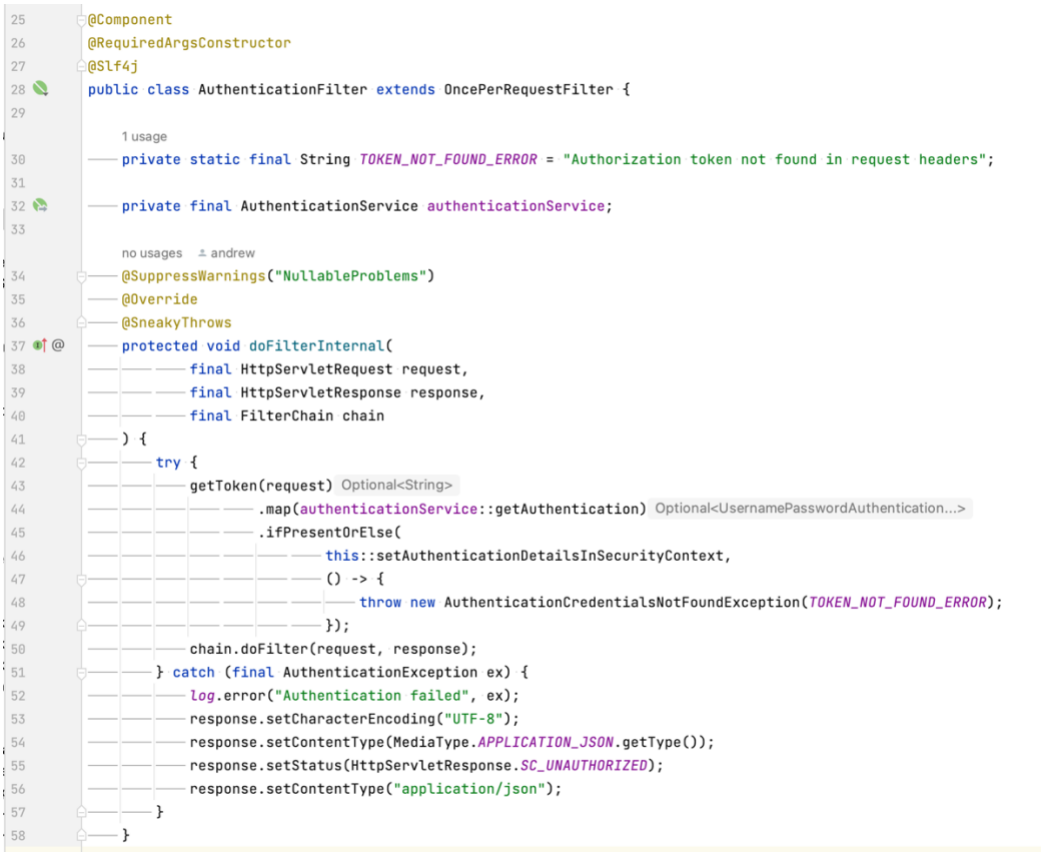
Отже, бібліотека common-logging дозволяє мати єдине місце налаштування формату й правил логування для всього проекту, а також вирішує проблему дистрибутивного трасування при синхронній комунікації між сервісами. Для розробника нового сервісу достатньо включити залежність на стартер common-logging і мати готову конфігурацію логування для будь-якого середовища.

3.2 Реалізація бібліотеки «common-security»

Бібліотека common-security реалізована для спрощення роботи з безпекою API. Якщо розподілена система використовує автентифікацію, що базується на наявності й валідності токену автентифікації, то сервіс має містити код, що може декодувати токен, перевірити його валідність, та виокремити інформацію, що він несе.

До уваги взята найпоширеніший формат токену – JWT, відповідно common-security бібліотека працює із ним. Для перевірки JWT використовується вищезгадана можливість Spring Boot створити клас-фільтр, що передує безпосередній обробці HTTP-запиту. Якщо токен не валідний або відсутній, то запит навіть не буде оброблений контролером, а клієнт отримає відповідну помилку з HTTP-кодом 401 (Unauthorized).

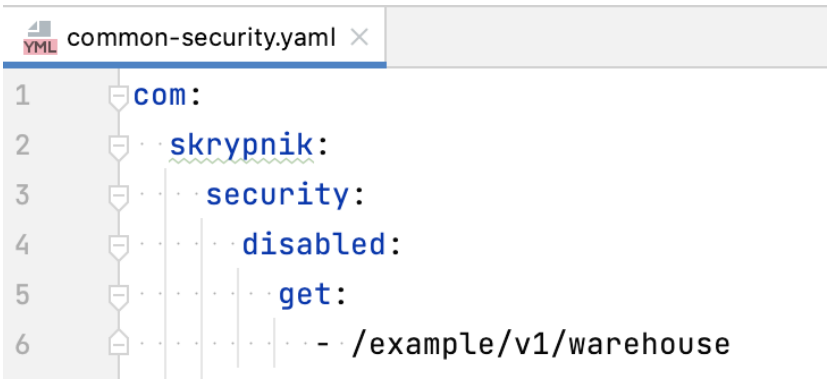
Для даного прикладу було розглянуто синхронне шифрування за алгоритмом HMAC256, але за необхідності можна розширити імплементацію бібліотеки для роботи із асинхронним шифруванням.



```java
@Component
@RequiredArgsConstructor
@Slf4j
public class AuthenticationFilter extends OncePerRequestFilter {

    1 usage
    private static final String TOKEN_NOT_FOUND_ERROR = "Authorization token not found in request headers";

    private final AuthenticationService authenticationService;

    no usages  andrew
    @SuppressWarnings("NullableProblems")
    @Override
    @SneakyThrows
    protected void doFilterInternal(
            final HttpServletRequest request,
            final HttpServletResponse response,
            final FilterChain chain
    ) {
        try {
            getToken(request) Optional<String>
                    .map(authenticationService::getAuthentication) Optional<UsernamePasswordAuthentication...>
                    .ifPresentOrElse(
                            this::setAuthenticationDetailsInSecurityContext,
                            () -> {
                                throw new AuthenticationCredentialsNotFoundException(TOKEN_NOT_FOUND_ERROR);
                            });
            chain.doFilter(request, response);
        } catch (final AuthenticationException ex) {
            log.error("Authentication failed", ex);
            response.setCharacterEncoding("UTF-8");
            response.setContentType(MediaType.APPLICATION_JSON.getType());
            response.setStatus(HttpServletResponse.SC_UNAUTHORIZED);
            response.setContentType("application/json");
        }
    }
```

Рисунок 3.2.1 – код перевірки JWT

Використовуючи можливості Spring Boot автоконфігурації, бібліотека common-security надає функціонал щодо відключення перевірки токену для певних ендпоінтів сервісу-клієнту бібліотеки. За замовченням, якщо стартер common-security включений в проект, то всі ендпоінти є безпечними – перевіряється JWT. Якщо є необхідність, зробити ендпоінт відкритим, то це можна налаштувати в конфігураційному файлі проекту application.yaml.



common-security.yaml

```yaml
com:
  skrypnik:
    security:
      disabled:
        get:
          - /example/v1/warehouse
```

Рисунок 3.2.2 – відключення перевірки JWT

Підсумовуючи, common-security надає дуже зручний функціонал для убезпечення API від доступу неавтентифікованих користувачів, та слугує єдиним місцем валідації JWT, а саме – перевірка підпису, перевірка часу дії токену та виокремлення інформації, що закодована в цьому токені в безпековий контекст, який також може бути використаний в інших частинах бізнес-логіки.

3.3 Реалізація бібліотеки «common-error-handling»

Головною метою загального підходу до обробки помилок є зрозумілість і передбачуваність API для клієнта. Помилки – природня частина інтеграції з API, наприклад, некоректний запит від клієнта найчастіше породжує 400 HTTP статус код від серверу. Або ж коли ресурс не знайдено на сервері, то клієнт отримує статус 404. Проте складні системи не можуть обмежитись лише HTTP статус кодами для визначення помилок, бо кількість причин цих помилок зростає пропорційно до кількості функціоналу, що надає система.

Так як кожен мікросервіс потенційно може розроблятись окремими людьми чи командами, то для цілісності й зрозумілості всієї системи треба забезпечити які оброблення у єдиному форматі всередині цього сервісу. Щоб уникнути дублювання коду та спростити розробку, код обробки помилок та створення структури помилок у окрему бібліотеку.

```
14  @RestControllerAdvice
15  @Slf4j
16  @RequiredArgsConstructor
17  /unchecked/
18  public class CommonExceptionHandler extends ResponseEntityExceptionHandler {
19
20      private final ExceptionProcessorStrategy exceptionProcessorStrategy;
21
        andrew
22      @ExceptionHandler(Exception.class)
23      protected ResponseEntity<ApiError> handleCommonApiException(Exception exception, WebRequest webRequest) {
24          val apiErrorResult = exceptionProcessorStrategy.getProcessor(exception).process(exception, webRequest);
25          return new ResponseEntity<>(apiErrorResult.getApiError(), apiErrorResult.getHttpStatus());
26      }
27  }
```

Рисунок 3.3.1 – клас обробник помилок

В ході виконання Java застосунку можуть бути викинуті винятки (exceptions), наприклад, "JsonParseException", "NullPointerException" або винятки, специфічні до конкретного застосунку. Фреймворк Spring Boot надає можливість створити обробника цих винятків в рамках одного класу, який відмічено анотацією @ControllerAdvice або @RestControllerAdvice. Це єдине місце в програмі, яке в залежності від типу винятку може віддати клієнту API відповідну HTTP-відповідь, модифікуючи при цьому тіло відповіді, HTTP-статус та, можливо, заголовки відповіді.

Бібліотека common-error-handling використовує патерн "стратегія" для надання клієнтам зручного способу визначати, яким чином оброблюється виняток в залежності від його типу – Java класу.

```
5 usages  1 implementation  andrew
public interface ExceptionProcessor<T extends Exception> {
    1 usage  1 implementation  andrew
    ApiErrorResult process(T exception, WebRequest webRequest);

    1 usage  1 implementation  andrew
    Class<T> getExceptionClass();

    1 usage  1 implementation  andrew
    HttpStatus httpStatus();
}
```

Рисунок 3.3.2 – інтерфейс обробник помилок

```
    1 usage  andrew
    @Override
@   public ApiErrorResult process(Exception exception, WebRequest webRequest) {
        val id = WebRequestUtil.getErrorId(webRequest);
        val apiError = ApiError.builder()
                .id(id)
                .code(defaultCode())
                .message(exception.getMessage())
                .messageFormat(exception.getMessage())
                .context(Map.of())
                .build();
        return ApiErrorResult.builder()
                .apiError(apiError)
                .httpStatus(HttpStatus.INTERNAL_SERVER_ERROR)
                .build();
    }
```

Рисунок 3.3.3 – імплементація обробника за замовченням

Для обробки специфічного для конкретного сервісу класу помилок розробнику достатньо імплементувати інтерфейс "ExceptionProcessor" (див. Рисунок 3.3.2), зазначивши, як екземпляр помилки можна конвертувати в HTTP-відповідь, а саме: який HTTP-статус та яке тіло відповіді отримає клієнт API. Тіло відповіді має єдину структуру, яку уособлює клас "ApiError". Цей клас має набір полів, що описують помилку: id – ідентифікатор помилки, за яким можна відслідкувати хід програми в логах; code – бізнес код помилки; message – повідомлення для користувача API; messageFormat – повідомлення з плейсхолдерами замість певних ключових даних, що можна використати для локалізації; context – мапа ключ-значення, де ключ – це плейсхолдер в messageFormat, а значення – дані системи (наприклад, "userId" – "ce98f79c-fe5c-11ed-be56-0242ac120002").

3.4 Реалізація бібліотеки «common-kafka»

Для демонстрації важливості спільної конфігурації брокера повідомлень було обрано Apache Kafka та існуючий Spring Boot стартер spring-kafka. По суті, common-kafka є розширенням цього стартеру із додатковими можливостями налаштування стратегій обробки помилок.

За замовченням використовується "standard" стратегія, що передбачає декілька додаткових спроб у разі виникнення помилки, та перенаправлення повідомлення у dead letter topic. Для dead letter topic [33] може існувати інший обробник або може бути сконфігурована система сигналів, що сигналізують про те, що система не змогла обробити повідомлення. При правильному налаштуванні така стратегія дозволяє не блокувати споживача повідомлень, якщо в брокері з’явилась повідомлення, що викликає непередбачувану поведінку системи.

Бібліотека common-kafka дозволяє налаштувати кількість спроб після виникнення помилки, визначати тип винятків ("neverRetryExceptions"),

після появи яких повідомлення одразу спрямовується в dead letter topic. Також клієнт може налаштувати зростання проміжків між повторними спробами обробити повідомлення для кожного топіку.

Можна налаштувати іншу стратегію оброки повідомлень – "stop-the-word". При цій стратегії у разі помилки обробки повідомлення система нескінченно намагається спробувати обробити повідомлення ще раз. Така стратегія може мати місце в застосунках, яким критично важлива консистентність й цілісність даних, наприклад, банківським установам. Така система має бути ретельно протестована й повідомлення, що можуть викликати "stop-the-word" ситуацію, мають бути дійсно дуже рідкісним явищем. У випадку Kafka, таке повідомлення блокує частину черги в залежності від кількості партицій в цій черзі. Тобто подальші повідомлення не будуть оброблені поки не будуть вчинені дії щодо того повідомлення, яке зупинило систему. Зазвичай є два варіанти: або робити термінову зміну в коді, яка виправить помилку, або вручну пропустити це повідомлення, збільшивши consumer-offset на один, повідомляючи брокеру, що даний сервіс пропускає обробку цього повідомлення. Звичайно, для ефективного використання переваг цієї стратегії необхідно забезпечити якісний моніторинг системи, і своєчасні сигнали на збільшення кількості помилок чи логів при виникненні "stop-the-world".

```
49  @Bean
50  public DefaultErrorHandler errorHandler(KafkaOperations<?, ?> kafkaTemplate) {
51      final var errorHandlingStrategy = Optional.ofNullable(kafkaProperties.getErrorHandlingStrategy())
52              .orElse(STANDARD_STRATEGY);
53      final DefaultErrorHandler consumerErrorHandler;
54      final var backOff = createBackOff(errorHandlingStrategy, kafkaProperties.getBackOff());
55      if (errorHandlingStrategy.equalsIgnoreCase(STOP_THE_WORLD_STRATEGY)) {
56          consumerErrorHandler = new DefaultErrorHandler(
57                  (rec, ex) -> log.warn("Message consumption failed, STW is working", ex), backOff);
58          consumerErrorHandler.setAckAfterHandle(false);
59      } else {
60          consumerErrorHandler = new DefaultErrorHandler(dltRecover(kafkaTemplate), backOff);
61          addNonRetryableExceptions(consumerErrorHandler);
62      }
63      configureTopicSpecificBackoff(errorHandlingStrategy, consumerErrorHandler);
64      return consumerErrorHandler;
65  }
```

Рисунок 3.4.1 – стратегії обробки помилок в Kafka

Підсумовуючи, бібліотека common-kafka надає зручну можливість обрати стратегію обробки помилок для окремого сервісу чи черги в брокері Kafka, звільняючи розробника від необхідності писати додатковий код, адже вся конфігурація може бути налаштована в файлі application.yaml, який зчитується бібліотекою під час ініціалізації Spring Boot сервісу.

3.5 Реалізація бібліотеки «common-integration-test»

Для імплементації бібліотеки, що уніфікує підхід до інтеграційного тестування, API використано бібліотеку «org.testcontainers:mockserver». Це інструмент, який дозволяє створювати та керувати контейнерами з ізольованими середовищами для інтеграційного тестування. У контексті MockServer, ця бібліотека використовується для створення і запуску ізольованого контейнера з MockServer, що дозволяє емулювати зовнішні залежності або сторонні сервіси, які використовуються у додатку під час тестування. Це дозволяє контролювати тестове середовище та надавати підроблені відповіді від MockServer для перевірки поведінки додатку у різних сценаріях тестування.

Для взаємодії з MockServer використано бібліотеку «org.mock-server:mockserver-client-java». За допомогою неї можна створювати налаштовувати очікувані HTTP-запити та відповіді, перевіряти, що очікувані запити були зроблені та багато іншого. Ця бібліотека дозволяє зручно використовувати MockServer у Java-проекті для ефективного тестування імітованих HTTP-серверів.

Вищезазначені інструменти чудово підходять для валідації поведінки веб-сервісу, що створений виходячи із OpenAPI специфікації. Для тестування треба створити клас, в якому необхідно налаштувати контейнер MockServer. Використовуючи MockServerClient, можна налаштувати очікувані запити та відповіді. В тестових методах викликається API та

перевіряється, чи відповідають отримані результати очікуваним. Звісно, конфігурація серверу, визначення поведінки API можуть бути уніфіковані й приведені до єдиного стандарту.

```
public abstract class WithRequestBodyIntegrationTest<Request, Response> extends CommonIntegrationTest<Response> {
    no usages  andrew
    @Test
    void executeMockServerClientTest() {
        TestRequestResponse<Request, Response> given = prepareRequest();
        Response response = executeRequest(given.getRequestBody());
        assertResponse(given.getHttpRequest(), response, given.getResponseBody());
    }

    1 usage  andrew
    protected TestRequestResponse<Request, Response> prepareRequest() {
        mockServerClient = defaultMockServerClient();
        final var requestDefinition = mockBuilder(mockServerClient)
                .withRequestBody(stringFileContent(requestJsonPath()))
                .mock();
        return new TestRequestResponse<>(
                requestDefinition,
                typedFileContent(requestJsonPath(), requestClass()),
                typedFileContent(responseJsonPath(), responseClass())
        );
    }

    1 usage  andrew
    protected Response executeRequest(Request request) {
        return testedApi().apply(request);
    }

    2 usages  andrew
    protected abstract String requestJsonPath();

    1 usage  andrew
    protected abstract Function<Request, Response> testedApi();
```

Рисунок 3.5.1 – абстрактний клас інтеграційного тесту

Методи конфігурації серверу і клієнту, емуляція поведінки API та перевірка результату винесенні в абстрактні класи CommonIntegrationTest, WithRequestBodyIntegrationTest. Використовується патерн проектування «шаблонний метод». Розробник, створюючи клас-імплементацію тесту має перевизначити абстрактні методи. Приклад: метод requestJsonPath() визначає локацію файлу, в якому прописане тіло HTTP-запиту, який зараз тестується. Метод testedApi() реалізує HTTP-запит до API, що тестується за допомогою REST-клієнту.

Використання бібліотеки «common-integration-test» робить процес написання інтеграційним тестів простим й дистильованим від зайвих конфігурацій тестового середовища. Програмісту достатньо наслідувати

готовий абстрактний клас та перевизначити декілька методів, щоб створити тест для нового API.

3.6 Демонстраційний проект з використанням розроблених бібліотек

Для демонстрації роботи та користі вищеописаних підходів та бібліотек було розроблено розподілену систему на базі мови програмування Java 17 та фреймворку Spring Boot 3.

Для прикладу було взято предметну доменну модель грошових переказів. Система дозволяє робити переказ – створювати транзакцію – з одного рахунку на інший. Система складається з чотирьох сервісів: сервіс автентифікації «user-management-service», сервіс керування рахунком «account-service», сервіс керування транзакціями «transaction-service» та сервіс нотифікацій «notification-service».

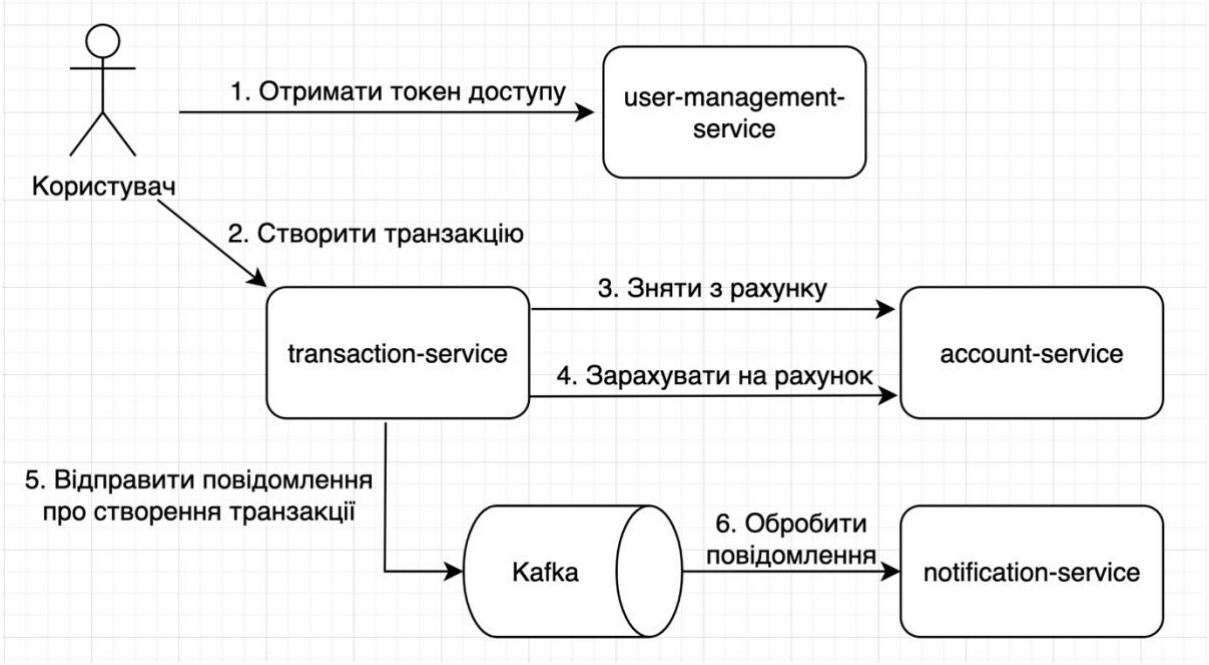


Рисунок 3.6.1 – архітектура демонстраційної системи

Перш за все користувач задля здійснення транзакції має бути автентифікованим в систему. Він вводить свої дані (ім’я користувача та пароль), і отримує токен доступу. Для подальших запитів до сервісу

керування транзакціями він має надати цей токен. Для валідації токену використовується бібліотека «common-security». У випадку відсутності необхідності перевірки токену для певних ендпоінтів системи це можна легко налаштувати шляхом конфігурація файлу application.yaml.

Для комунікації між сервісами керування транзакціями та керування рахунками використовується синхронний підхід – HTTP-запити. Для умовної незалежної розробки цих сервісів був використаний API-first підхід: першопочатково була написана OpenAPI специфікація, та згенерований код серверу і клієнту для сервісу керування рахунками за допомогою Maven плагіну «openapi-generator-maven-plugin». Це забезпечило можливість паралельної розробки двох сервісів.

Інтеграційне тестування API було проведено за допомогою бібліотеки «common-integration-test», яка дозволила не витрачати час на налаштування тест-середовища, розгортання docker-контейнерів та написання логіки зчитування тест-даних із файлу. Більше того, для виконання запитів до серверу був використаний автоматично-згенерований клієнт з OpenAPI специфікації. В цілому, написання тесту зводиться до створення JSON-файлів із тест даними та перевизначення декількох методів із заготовлених абстрактних класів.

В усіх сервісах використовується бібліотека «common-error-handling» для єдиного способу обробки винятків та перетворення їх в HTTP-відповідь уніфікованого формату, що робить систему цілісною та передбачуваною.

Єдиний формат логування та можливість трасування запитів забезпечується за допомогою бібліотеки «common-logging».

Бібліотека «common-kafka» використана для асинхронної комунікації між сервісом керування транзакціями та сервісом нотифікацій. Для того щоб зчитати чи відправити повідомлення з параметрами конфігурації споживача за замовчення, нічого додатково налаштовувати не потрібно, достатньо просто додати бібліотеку в проект.

Підсумовуючи, використання розроблених бібліотек в демонстраційному проекті показало, що для прискорення, паралелізації й уніфікації розробки розподіленої системи слід розглянути запропоновані вище підходи. Бібліотеки значно економлять час, коли йдеться про розробку супутніх до бізнес-логіки речей таких як: логування, безпека, інтеграційне тестування. Розроблені у форматі Spring Boot старетрів бібліотеки дозволяють зробити конфігураційний файл сервісу application.yaml єдиним місцем налаштування асинхронного спілкування, обробки помилок та безпеки, що також робить розробку простішою й швидшою.

Висновки

В ході виконання роботи проаналізовано й досліджено проблематику уніфікації коду й підходів до розв’язання проблем в розробці систем з мікросервісною архітектурою. Були визначені компоненти, реалізація яких має бути єдиною та уніфікованою в усіх частинах розподіленої системи задля ефективної роботи над кодом, його підтримкою і розширенням.

У результаті, за допомогою мови програмування Java та можливостей фреймворку Spring Boot розроблено прототипи бібліотек, які слугують єдиним місцем налаштування важливих аспектів розподіленої системи: логування, обробка помилок, безпека, асинхронна комунікація й інтеграційне тестування. Для перевірки результатів роботи та коректності роботи бібліотек було створено демонстраційний проект з мікросервісною архітектурою.

Було оцінено актуальність використання кожної бібліотеки та доступність їх впровадження та можливостей додаткового налаштування під конкретні потреби їх клієнтів.

Список використаних джерел

1. Sam Newman (2014) "Building Microservices: Designing Fine-Grained Systems"
2. Alphabet: Number of Employees 2010-2023 | GOOGL. Електронний ресурс. https://www.macrotrends.net/stocks/charts/GOOGL/alphabet/number-of-employees
3. Martin Klepmann (2017) "Designing Data-Intensive Applications"
4. Chris Richardson (2018) "Microservices Patterns: With Examples in Java"
5. John Carnell, Illary Huaylupo Sánchez (2021) "Spring Microservices in Action"
6. George Coulouris, Jean Dollimore, and Tim Kindberg (2011) "Distributed Systems: Concepts and Design"
7. Michael T. Nygard (2007) "Release It!: Design and Deploy Production-Ready Software"
8. Chris Richardson (2018) "Microservices Patterns: With Examples in Java"
9. Chris Richardson (2018) "Microservices Patterns: With Examples in Java"
10. Martin Reddy (2011) "API Design for C++"
11. Robert C. Martin (2008) "Clean Code: A Handbook of Agile Software Craftsmanship"
12. Flavio Junqueira and Benjamin Reed (2013) "ZooKeeper: Distributed Process Coordination"
13. Clarence Ho, Rob Harrop,Chris Schaefer (2017) "Pro Spring 5: An In-Depth Guide to the Spring Framework and Its Tools"
14. Craig Walls (2018) "Spring in Action"
15. Apache Log4j™ 2. Електронний ресурс https://logging.apache.org/log4j/2.x/
16. Yuri Shkuro (2019). "Mastering Distributed Tracing: Analyzing performance in microservices and complex systems"

17. Spring Cloud Sleuth. Електронний ресурс https://spring.io/projects/spring-cloud-sleuth
18. Standards Overview. Електронний ресурс https://www.pcisecuritystandards.org/standards/
19. Health Insurance Portability and Accountability Act of 1996 (HIPAA). Електронний ресурс https://www.cdc.gov/phlp/publications/topic/hipaa.html
20. General Data Protection Regulation. Електронний ресурс. https://gdpr-info.eu/
21. Justin Richer, Antonio Sanso (2017). "OAuth 2 in Action"
22. Neha Narkhede, Gwen Shapira, Todd Palino (2017). "Kafka: The Definitive Guide"
23. Sam Newman (2014) "Building Microservices: Designing Fine-Grained Systems"