КИЇВСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ

ІМЕНІ ТАРАСА ШЕВЧЕНКА

Факультет комп'ютерних наук та кібернетики

Кафедра інтелектуальних програмних систем

Кваліфікаційна робота

на здобуття освітнього рівня бакалавра

За спеціальністю 121 Програмна інженерія

На тему:

РОЗРОБКА БІБЛІОТЕКИ ДЛЯ РЕАЛІЗАЦІЇ ШАБЛОНУ ПРОЕКТУВАННЯ «MULTITENANCY»

Виконала студентка 4-го курсу
Юлія НОРТМАН _____

Науковий керівник:
доцент, кандидат фіз.-мат. наук
Олександр ГАЛКІН _____

Засвідчую, що в цій роботі немає запозичень з праць інших авторів без відповідних посилань.
Студентка _____

Роботу розглянуто й допущено до захисту
На засіданні кафедри інтелектуальних програмних систем
«___» _____ 2021р.,
протокол № 14 від 28.05.2021р
Завідувач кафедри
Олександр ПРОВОТАР _____

Київ – 2021

РЕФЕРАТ

Обсяг роботи 50 сторінок, 7 ілюстрацій, 21 джерело посилань.

HIBERNATE, SAAS, SPRING BOOT, БАГАТООРЕНДНІСТЬ, БАЗА ДАНИХ, ОРЕНДАТОР, ХМАРА

Об’єктом роботи є дослідження особливостей архітектури багатоорендності (мультиорендності) у SaaS застосунках та можливостей її реалізації з використанням фреймворків Spring 5 та Hibernate. Предметом роботи є бібліотека для підтримки багатоорендності.

Метою роботи є створення Spring бібліотеки для реалізації багатоорендної поведінки у додатках, які написані з використанням фреймворків Spring 5 та Hibernate, і дослідження ефективності реалізації.

Інструментами розробки є безкоштовне вільно поширюване інтегроване середовище розробки Eclipse 4.19 для корпоративної та веб розробки Java застосунків, мова Java 11, система міграцій баз даних Liquibase, база даних PostgreSQL 13.2 та засіб для автоматизації роботи з проектами Apache Maven 3.8. Ручне тестування програми відбувалося за допомогою системи Postman.

Результати роботи: виконано загальний огляд підходів до реалізації багатоорендної архітектури, а також переваг та недоліків, які властиві кожному з них. Розроблено Spring Boot Starter для реалізації багатоорендної поведінки.

Створений програмний продукт може використовуватися при розробці хмарних сервісів, які потребують підтримки багатоорендності.

В подальшому бібліотека може бути удосконалена з метою покращення часової ефективності її роботи. Крім того, можливе додавання функціоналу для підтримки масштабованості сервісів, а також інтеграції з іншим ORM фреймворком, відмінним від Hibernate.

ЗМІСТ

СКОРОЧЕННЯ ТА УМОВНІ ПОЗНАЧЕННЯ

ASP – Application Service Provider, постачальник програмних сервісів

DI – Dependency Injection, інверсія залежностей

IaaS – Infrastructure as a Service, інфраструктура як послуга

JVM – Java Virtual Machine, віртуальна машина Java

ORM – Object-relational mapping, об’єктно-реляційне відображення

PaaS – Platform as a Service, платформа як послуга

POJO – Plain Old Java Object

SaaS – Software as a Service, програмне забезпечення як послуга

SQL – Structured Query Language, мова структурованих запитів

ООП – Об’єктно-орієнтоване програмування

СУБД – Система управління базами даних

ВСТУП

Оцінка сучасного стану об'єкта дослідження або розробки. З появою та розвитком хмарних технологій змінився підхід до вирішення бізнес проблем. На сьогоднішній день більше ніж 69% сервісів використовують хмарні обчислення і, згідно зі статистичними дослідженнями, ця цифра продовжує зростати і досягне відмітки у 90% до кінця 2021го року[1)]. Хмарні сервіси надають великий спектр послуг: від відкритих та загальнодоступних публічних сервісів до приватних хмар з високим рівнем надійності та безпеки.

Ключовим моментом хмарних технологій, який відрізняє їх від традиційного підходу є перетворення ізольованих даних у загальнодоступний пул ресурсів. У більшості таких ситуацій виникає потреба у створенні багатоорендного середовища, що дає змогу ввести поняття спільних ресурсів. У багатоорендних SaaS застосунках кожен клієнт функціонує у віртуальній ізоляції по відношенню до інших. В той час як індивідуальні дані кожного орендатора залишаються приватними і доступними лише для нього, насправді всі клієнти розділяють веб сервери, інфраструктуру, сховища даних та пам'ять. У даній роботі основна увага буде сконцентрована на реалізації мультиорендності у контексті баз даних.

Актуальність роботи та підстави для її виконання. Мова Java з року в рік займає найвищі позиції в рейтингу найбільш використовуваних мов програмування[2)] як для створення автономних додатків так і для написання великих корпоративних застосунків. Згідно з опитуванням JVM Ecosystem Report,

---

[1)] За даними досліджень науково-консультативної фірми в області IT технологій та фінансів Gartner (https://www.gartner.com/en/newsroom/press-releases/2018-09-12-gartner-forecasts-worldwide-public-cloud-revenue-to-grow-17-percent-in-2019), дослідницької фірми у галузі технологій 451 Group (https://451research.com/images/Marketing/press_releases/Pre_Re-Invent_2018_press_release_final_11_22.pdf) і технологічної компанії Cisco (https://www.cisco.com/c/en/us/solutions/collateral/executive-perspectives/annual-internet-report/white-paper-c11-741490.html).

[2)] Згідно з даними, представленими компанією Statista (https://www.statista.com/statistics/793628/worldwide-developer-survey-most-used-languages) та індексом популярності мов програмування - PYPL (https://pypl.github.io/PYPL.html).

опублікованому компанією Snyk у 2020му році найпопулярнішим фреймворком для Java вважається Spring Framework, а 50% з цих розробників використовують Spring Boot. Однією з ключових особливостей у використанні Spring Boot є можливість простого вирішення задач, які часто зустрічаються, за допомогою набору Spring Boot Starter, що легко інтегруються у будь-який архітектурний рівень додатку.

Іншим сучасним розповсюдженим засобом, який найчастіше використовується в комбінації із Spring Boot є ORM фреймворк Hibernate. Починаючи з версії 4.0, у ньому з'явилася підтримка мультиорендності, але для забезпечення її повноцінної роботи розробникам все ще необхідно створювати велику кількість шаблонного коду. Крім того, написання такого коду вимагає глибокого розуміння структури модуля Spring Data і порушує головний принцип фреймворку – угода головніша за конфігурацію (Convention over Configuration). Ще одним значним недоліком є той факт, що Hibernate реалізує лише два з трьох підходів до багатоорендності, що планувалося виправити з виходом нової версії, але так і не було зроблено.

Мета й завдання роботи. Метою роботи є створення Spring Boot Starter для реалізації мультиорендності [1], використовуючи для цього можливості, передбачені фреймворками Spring 5 та Hibernate. Для досягнення цієї мети було поставлено наступні цілі:

- Дослідити і вивчити поняття архітектурного шаблону багатоорендності, розглянути можливі підходи до його реалізації.
- Дослідити внутрішню реалізацію модуля Spring Data та Hibernate.
- Створити програму та дослідити її на ефективність і надійність у використанні.

Можливі сфери застосування. Створений програмний продукт може використовуватися для реалізації багатоорендної поведінки у SaaS сервісах, написаних з використанням фреймворків Spring та Hibernate.

РОЗДІЛ 1 ТЕОРЕТИЧНА ЧАСТИНА

1.1 Хмарні технології та їх характеристика

Хмарні обчислення - це модель, яка надає зручний доступ по мережі до спільних ресурсів, таких як сервери, сховища, платформи для розгортання, програми та служби, з можливістю їх подальшого налаштування. Їх виділення та звільнення відбувається з мінімальною взаємодією з боку постачальника сервісу. [2] У порівнянні з традиційним підходом, така модель надає низку переваг таких як вартість, швидкість, масштабованість, безпечність, надійність та продуктивність. Поняття «хмара» описує сукупність серверів, до яких можна отримати доступ через мережу, а також програмне забезпечення, яке на них встановлено.

1.1.1 Типи хмарних обчислень

Існує три типи моделі хмарних обчислень. Публічна хмара надає доступ до інфраструктури, яка не належить кінцевому користувачу. Традиційним підходом до розташування серверів було лише використання хмари, тоді як зараз деякі постачальники можуть використовувати дата центри користувачів. Додатки, які працюють у таких хмарах, як наприклад соціальні мережі Facebook або LinkedIn, зазвичай не потребують складної процедури реєстрації для користування, але в свою чергу не гарантують надійного збереження даних.

У випадку приватної хмари клієнтом є один або група користувачів і тільки вони, тобто доступ до ресурсів обмежений лише однією організацією. На відміну від попереднього випадку інфраструктуру приватних хмар навпаки прийнято розміщувати у дата центрах, які належать організації, однак пізніше з’явилися підходи до використання центрів постачальників хмарних сервісів, що спричинило появі нових підтипів приватних хмар [3]: ті що управляються стороннім постачальником та внутрішніх, тобто таких, які існують в межах більшої, приватної чи публічної хмари.

Третім типом хмарної інфраструктури є гібридний або змішаний тип, який являє собою поєднання характеристик двох попередніх. Прикладом може слугувати організація, яка має свою власну приватну хмару і крім того додатково

розділяє свої ресурси з іншою публічною хмарою. Це дозволяє використовувати приватні хмари для надійного збереження конфіденційної інформації, але для інших потреб використовувати більш швидкі публічні ресурси. Згідно із щорічним звітом компанії Flexera[1)], вже декілька років підряд такий тип є найбільш поширеним серед компаній, а спосіб поєднання різних типів хмар залежить від бізнес потреб користувачів.

Іноді виділяють ще декілька типів такі як мультихмарність (розгортання різних відносно незалежних компонентів одного сервісу на різних хмарах від різних постачальників) та спільні хмари (community clouds) [2], які є різновидом приватних хмар, що призначені для використання декількома організаціями.

1.1.2 Сервіси хмарних обчислень

Іншою класифікацією хмарних обчислень є розділення їх на три основні моделі: інфраструктура як послуга (Infrastructure as a Service, IaaS), платформа як послуга (Platform as a Service, PaaS) та програмне забезпечення як послуга (Software as a Service SaaS). Їх взаємозв'язок та різні рівні контролю часто зображають у вигляді піраміди (Рисунок 1.1).

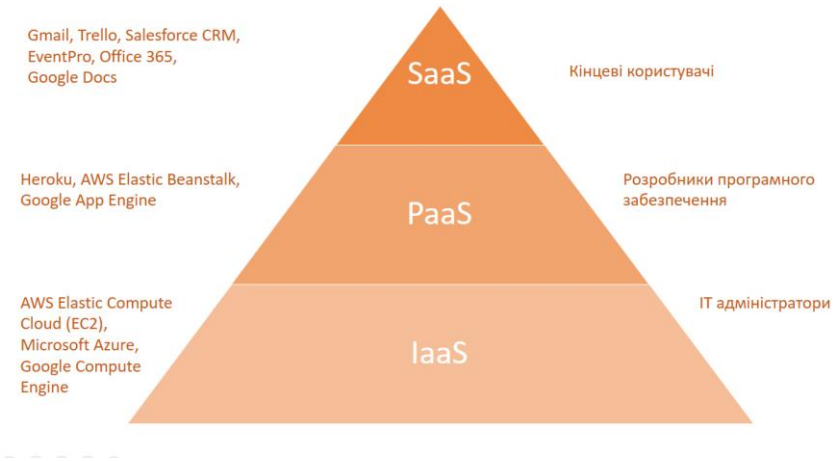

Рисунок 1.1 – Сервіси хмарних обчислень

[1)] Flexera 2020 State of the Cloud Report (https://info.flexera.com/SLO-CM-REPORT-State-of-the-Cloud-2020)

IaaS модель надає користувачу можливість користування віртуалізованим апаратним забезпеченням, тобто серверами, сховищами тощо. Кожен ресурс доступний як окремий компонент сервісу, і клієнт платить лише за ту частину якою він користується. При такому підході відсутня необхідність самостійно купляти, встановлювати та керувати серверами. Постачальник сервісу хмарних обчислень, як наприклад Amazon Web Services (AWS) або Google Cloud Platform (GCP), управляє інфраструктурою в той час як клієнт може самостійно обирати, встановлювати та налаштовувати все необхідне програмне забезпечення таке як віртуальні машини, операційні системи, бази даних тощо.

PaaS це інша форма хмарних обчислень, що являє собою платформу для розгортання, тестування та керування програмним забезпеченням. Користувачами платформи як сервісу є розробники ПЗ. Вони не мають прямого доступу до хмарної інфраструктури, але контролюють застосунки, які були розгорнуті на платформі і можуть користуватися вже наявними засобами розробки або управління контейнерами, такими як Kubernetes. Ця модель вважається найскладнішою з усіх трьох складових хмарної структури. Її прикладами є Salesforce, Heroku, Red Hat, AWS тощо.

SaaS є найбільшим сегментом ринку хмарних обчислень[1)] і дозволяє користувачам отримати доступ до додатку через клієнтський застосунок (наприклад браузер), встановивши з'єднання по мережі, замість самостійного локального встановлення на машину. Це є важливою особливістю, оскільки користування таким додатком можливе з будь-якого пристрою, на якому встановлений клієнт, включаючи мобільні телефони та планшети. У SaaS моделі розробник не продає ліцензію на продукт. Натомість кожен покупець платить певну суму і отримує доступ до компонентів додатку, при цьому отримавши статус орендатора сервісу і можливість зберігати свої дані в системі. Правильно

[1)] За даними досліджень науково-консультативної фірми в області IT технологій та фінансів Gartner (https://www.gartner.com/en/newsroom/press-releases/2018-09-12-gartner-forecasts-worldwide-public-cloud-revenue-to-grow-17-percent-in-2019)

спроектована SaaS система володіє трьома ключовими особливостями: масштабованість, можливість налаштування та ефективна реалізація багатоорендності.

Використання SaaS платформ характеризується певними перевагами у порівнянні з традиційним завантаженням і встановленням програм, оскільки не потребує спеціальних вимог до апаратного забезпечення користувача і є платформо незалежним. Ще однією особливістю є те, що користувач не прив'язаний до конкретного пристрою, а всі його дані зберігаються на сервері, що гарантує їх цілісність навіть у разі відмови комп'ютера чи мобільного телефону. Крім того, користувач не повинен дбати про оновлення програми, оскільки ця функція покладається на постачальника програмного забезпечення.

Можливість масштабування під потреби користувача є іншою ключовою особливістю SaaS застосунків. Це дозволяє клієнту мінімізувати витрати, якщо у системі наявні зайві ресурси або навпаки, збільшити розмір сховища або об'єм оперативної пам'яті, при цьому не турбуючись про необхідність придбання додаткового апаратного забезпечення.

1.1.3 Переваги та недоліки використання хмарних обчислень

Конкретний перелік переваг варіюється від системи до системи, але в цілому найголовнішою перевагою є те, що компаніям більше не потрібно купувати та встановлювати свою власну інфраструктуру. На сьогоднішній день значна кількість компаній переходить на використання хмарних технологій для того, щоб зменшити вартість та складність розміщення програмних продуктів. Наприклад використання PaaS платформи дозволяє уникнути необхідності виявлення таких помилок, які виникають внаслідок відсутності потрібного конфігураційного файлу, оскільки це є завданням самої системи. Іншою перевагою використання цього рівня абстракції є відсутність необхідності у спеціалістах проміжного програмного забезпечення.

Можливість збільшувати ємність ресурсів за необхідності дозволяє впроваджувати розширення сервісів набагато ефективніше в економічному плані,

тому що не вимагає встановлення необхідної інфраструктури. Наприклад, така ситуація може виникнути при періодичному використанні певного сервісу, або якщо очікується тимчасове збільшення навантаження.

Однак використання хмарних ресурсів не завжди буде більш дешевою опцією, оскільки інколи вартість оренди на довгий проміжок часу може перевищити вартість потрібного апаратного забезпечення. Тому якщо заздалегідь відомі ресурсні потреби додатку, які не зміняться з часом, то використання хмарних платформ в такій ситуації буде менш оптимальним з економічної точки зору. Також деякі компанії можуть відмовитися від використання хмар для збереження приватних даних клієнтів.

Крім того, в той час як розгортання нових сервісів з використанням хмарних технологій є відносно легкою процедурою, міграція вже існуючого продукту може стати достатньо непростою і затратною задачею.

1.2 Багатоорендність у SaaS застосунках

Поняття одноорендності описує ситуацію, коли у кожного клієнта є своя одиниця програми, тобто кожен орендатор має свою базу даних та власні ресурси, і доступ іншим клієнтам до них відсутній. Для того, щоб додати нового клієнта потрібно розгорнути нову програму, створити базу даних і інколи може виникнути необхідність внесення змін в код для того, щоб забезпечити потреби орендатора. Перевагами такої системи є надійність сервісу (оскільки він доступний для використання одним клієнтом, а отже ресурси будуть доступні безперервно), приватність даних та можливість адаптування під потреби користувача. Тим не менш, значними недоліками такої системи будуть складність і вартість її розгортання та підтримки, оскільки весь сервіс функціонує лише для одного клієнта, а це збільшує вірогідність того, що ресурси не будуть використовуватися на всю потужність.

Багатоорендність або мультиорендність (англ. multitenancy) – архітектурний шаблон, ідея якого полягає у використанні багатьма клієнтами (або орендаторами) одного ресурсу. Це дозволяє одному додатку більш гнучко та ефективно надавати

послуги багатьом орендаторам без виділення додаткових ресурсів для програмного забезпечення, при цьому вся спільна інформація є доступною для всіх клієнтів, окрім приватних даних специфічних для кожного орендатора, які мають бути надійно ізольовані одне від одного. Орендатором (tenant) називають групу користувачів, у яких є спільний доступ з певним набором привілей до певного програмного забезпечення.

При реалізації цього шаблону необхідно звернути увагу на два ключові моменти. З одного боку краще мати у спільному використанні якомога більше для ефективнішого використання ресурсів, дешевшої вартості та більшої гнучкості. З іншої сторони потрібно критичним моментом стає приватність, конфіденційність та цілісність даних кожного орендатора. Також потрібно передбачити ситуацію, коли один орендатор може потенційно завадити належному функціонуванню інших орендаторів.

1.2.1 Підходи до реалізації

Існує три основні підходи для забезпечення мультиорендності при розробці SaaS застосунків, кожен з яких володіє певними особливостями у порівнянні з іншими.

1.2.1.1 Окрема база даних

Дані про кожного орендатора зберігаються в окремій базі даних, що забезпечує надійну ізоляцію даних. Крім того, це створює додатковий простір для оптимізації та адаптації, оскільки схема для будь-якої бази даних може змінюватися в залежності від потреб клієнта, що ніяким чином не вплине на структуру даних інших клієнтів. Це може бути корисним, якщо клієнт потребує додаткових полів, крім стандартно доступних всім орендаторам.

Використання окремої бази даних для кожного орендатора є достатньо затратною операцією з точки зору вартості та вимог до апаратного забезпечення і підтримки, однак є найбільш ефективним при забезпеченні надійності та безпеки даних і може використовуватися у таких сферах як банкінг або медицина.

Основний підхід для розв’язання цієї задачі полягає у створенні окремого з’єднання або пулу з’єднань для кожного клієнта. Вибір пулу для з’єднання з базою даних буде залежати від того, який орендатор зараз активний.

1.2.1.2 Окрема схема

Інформація знаходиться у спільній базі даних, але за кожним клієнтом закріплена своя схема. Так як і в попередньому випадку існує можливість модифікації таблиць після їх створення, таким чином адаптуючи їх згідно з потребами користувача. Ця модель забезпечує менш надійну ізоляцію даних, але дозволяє зберігати більше орендаторів на одному сервері баз даних.

Такий підхід можна реалізувати двома способами. Перший варіант схожий з вищеописаним рішенням для випадку окремої бази даних, у другому випадку з’єднання або пул з’єднань будуть спільними для всіх орендаторів, але перед кожним використанням буде необхідним вказати схему самостійно, наприклад за допомогою SQL команди SET SCHEMA.

Значним недоліком такої реалізації є відновлення даних у разі відмови. У випадку окремої бази даних для цього потрібно лише відновити всі записи бази даних з останнього файлу резервної копії. Якщо виконати такі самі дії у випадку спільного сховища, то крім даних для конкретного орендатора, буде оновлена вся інформація у базі даних, включаючи тих користувачів, які не потребують цього. Тому для того, щоб відновити дані для конкретного користувача, адміністратору необхідно завантажити всі дані з резервної копії на тимчасовий сервер і тільки після цього імпортувати необхідні таблиці на робочий сервер, що є досить складною і затратною по часу операцією.

1.2.1.3 Розділення даних за допомогою дискримінатора

Всі клієнти використовують одні і ті самі таблиці і розділені окремим дискримінатором (тобто полем або полями які містять ідентифікатор клієнта), який є унікальним значенням для кожного орендатора. Крім того, може виникнути потреба у створенні додаткових таблиць або полів, які використовуються лише частиною орендаторів. Цей підхід дозволяє зберігати дані, на які посилаються всі

клієнти, як наприклад типи користувачів в системі, лише один раз в одному місці, але при додаванні нових клієнтів розмір бази даних збільшується, що вимагає більшого об’єму пам’яті та ресурсів. Зазвичай у такого масштабування існує верхня границя, тим не менш, часто база даних стає занадто громіздкою для користування ще до досягнення цієї межі. Більше того, управління такою системою є складнішим у порівнянні з іншими варіантами, а зі збільшенням розмірів сховища операції виконуються повільніше.

При такому сценарії розв’язання задачі ізоляції даних різних клієнтів одне від одного стає проблемним місцем і вимагає від розробника гарантій того, що жоден запит не зачіпляє даних більше ніж одного орендатора.

Такий спосіб може використовуватися, коли необхідно зберігати велику кількість орендаторів з якомога меншою кількістю серверів, тобто коли клієнт готовий пожертвувати ізоляцією даних в обмін на меншу вартість обладнання.

Як і в попередньому випадку, єдиний пул з’єднань вказує на базу даних, але в цьому випадку кожен SQL запит має бути модифікований, оскільки до нього необхідно додати ідентифікатор клієнта.

1.2.2 Вибір підходу

При виборі підходу для реалізації мультиорендності потрібно зважати на декілька речей. По-перше, це кількість орендаторів, їх поведінка та структура бази даних для кожного з них. Якщо в системі передбачається велике число клієнтів, то скоріш за все розробник буде прагнути зменшення вартості апаратного забезпечення і надасть перевагу другому або третьому методу. Тим не менш, якщо кількість даних для кожного клієнта є значною, або якщо очікується велика кількість клієнтів, які працюють в паралельному режимі то використання підходу окремої бази даних для кожного клієнта певним чином оптимізує роботу програми. Крім того, цей спосіб є найпростішим в реалізації, і у випадку міграції від одноорендного сервісу до мультиорендного, може виявитися найменш затратним рішенням, оскільки воно майже не потребує внесення змін до програмного коду.

По-друге, як було зазначено вище, різні рівні доступу до сховища гарантують різний рівень ізоляції даних. Розділення даних за допомогою дискримінатора є найменш надійним у цьому випадку, тоді як використання окремої бази даних забезпечує високий рівень приватності.

В цілому, вибір стратегії збереження даних не впливає на функціональність сервісу і залежить лише від того, які критерії [4], як наприклад масштабованість, рівень ізоляції даних, адаптивність тощо, є важливими при вирішенні даної задачі.

1.2.3 Особливості багатоорендної архітектури

Мультиорендність дозволяє зменшити вартість сервісу, оскільки надає можливість декільком орендаторам отримати доступ до одного й того самого спільного ресурсу замість того, щоб дублювати його для кожного окремо. Однак це, в свою чергу, вимагає більш складної архітектури при побудові програм, їх конфігурації та підтримки, і, як результат, випливає у більшу початкову вартість. Крім того, надання спільного доступу забезпечує більшу гнучкість та масштабованість програм.

Для того, щоб отримати можливість управління клієнтами вводиться поняття головного орендатора (master tenant) [5], який зберігає інформацію про всіх клієнтів. Цей підхід подібний до наявності у системі суперадміністратора, який керує даними всіх користувачів.

1.2.4 Захист даних у багатоорендних сервісах

Витік даних відбувається у випадку, коли приватна інформація одного орендатора стає випадково або навмисно доступною іншому, неавторизованому клієнту. Існує чотири можливі сценарії [6], за яких це може відбутися.

- Запит орендатора А отримує доступ до сховища даних орендатора В, таким чином маючи змогу прочитати або змінити дані, що там зберігаються, що порушує принцип цілісності даних орендатора В.
- Дані орендатора А, які були отримані з бази даних надійшли як результат запиту до сервісу орендатора В, що дало орендатору В доступ до даних орендатора А і спричинило порушення принципу конфіденційності даних.

- Вміст бази даних орендатора А було скопійовано до бази даних орендатора В, що дало доступ орендатору В до приватних даних орендатора А і стало порушенням як принципу цілісності даних (у випадку орендатора В) так і їх конфіденційності (у випадку орендатора А).
- Вміст об’єкта, який містить ідентифікатор клієнта буде скопійовано до об’єкта, який не містить такого ідентифікатора і стане доступний всім орендаторам, які користуються сервісом.

Безпечний SaaS сервіс має належним чином гарантувати приватність даних, використовуючи декілька рівнів захисту, які доповнюють один одного і в цілому створюють захист за будь-яких умов від зовнішніх і внутрішніх загроз. Для цього варто використовувати три основні технології: фільтрацію даних, що працює між рівнем бізнес-логіки та рівнем доступу до даних, режими доступу користувачів та шифрування даних.

У додатках з багаторівневою архітектурою традиційно виділяють два методи, які забезпечують певний рівень безпеки доступу до даних: імперсонізація (impersonation) та використання надійного акаунту підсистеми (trusted subsystem account). У першому випадку у базі даних налаштовані користувачі з певним правами доступу, і кожного разу, коли користувач хоче виконати операцію, яка вимагає доступу до сховища, то для бази даних сервіс виглядає як користувач з певним набором прав. У разі вибору другої стратегії з точки зору бази даних існує лише один користувач – сама система, який має дозвіл виконувати абсолютно всі операції, а права доступу регулюються самим сервісом.

У випадку багатоорендності допускається змішаний підхід через різницю у поняттях «користувач» і «орендатор», так як один орендатор ототожнює у собі одного або декількох користувачів з різним набором прав. Тому в даній ситуації орендатор дозволяє користувачам звертатися до бази даних, використовуючи підхід надійного акаунту підсистеми, тобто за допомогою свого облікового запису, але при цьому контролює яка частина з отриманої інформації буде доступна користувачу.

Шифрування даних буває двох видів – симетричне та асиметричне (з використанням публічного ключа) [7]. Симетричне шифрування передбачає наявність одного секретного ключа, за допомогою якого відбувається як шифрування так і дешифрування даних. Ризик, який виникає при застосуванні такого підходу полягає у тому, що в цьому випадку абонент А, який хоче надіслати зашифрований текст абоненту В, має спочатку надіслати йому ключ, який може бути перехоплений третьою небажаною особою. При асиметричному шифруванні існують два ключи, один публічний, який відомий всім, і другий приватний, який має зберігатися у секреті. При шифруванні тексту абонент А використовує публічний ключ абонента В і надсилає шифротекст. Абонент В у свою чергу може розшифрувати це повідомлення, використовуючи свій приватний ключ, який відомий лише йому.

У SaaS застосунках алгоритми шифрування за допомогою публічного ключа можуть бути неприйнятними через свою більшу складність з точки зору обчислень, тому іноді може бути використаний гібридний метод, який поєднує у собі переваги двох вищеописаних алгоритмів. У цьому підході для кожного орендатора генерується три ключі, один секретний, один приватний та один публічний. Використовуючи секретний ключ за допомогою симетричного алгоритму шифруються всі конфіденційні дані клієнта. Наступним етапом є шифрування самого секретного ключа асиметричним алгоритмом. Такий додатковий рівень протекції даних передбачає захист від ситуації, коли випадково або навмисно дані одного орендатора потрапили до іншого, оскільки він не зможе ними скористатися не володіючи коректним приватним ключем.

Необхідність шифрування зростає при виборі підходу з використанням спільної бази даних або спільних таблиць для збереження приватної інформації орендаторів

1.2.5 Додавання полів у багатоорендних сервісах

У кожній багатоорендній системі є базовий набір заздалегідь визначених таблиць і полів, який доступний всім клієнтам. Однак, у орендатора може

виникнути необхідність збереження додаткової специфічної для нього інформації. Найпростішим підходом буде передбачити в кожній таблиці набір полів базових типів, але такий спосіб містить декілька дуже важливих недоліків. По-перше, велика кількість пам’яті буде виділена під колонки, але насправді не використовуватися. По-друге, таких полів може не вистачити або навпаки, виникне потреба у більшій кількості колонок певного конкретного типу ніж було передбачено системою. Останню проблему легко вирішити за допомогою метаінформації. Для цього треба визначити тип всіх небазових полів як строковий і додатково ввести таблицю, у якій буде зберігатися ідентифікатор орендатора, назва, яку клієнт обрав для першої додаткової колонки, її фактичний тип, назву другої колонки, її тип тощо. Негативним аспектом такого рішення буде необхідність приводити дані до строкового типу при збереженні до бази даних і виконанні оберненої процедури при читанні інформації з неї. Крім того, залишається проблема кількості додаткових полів, яка при такому підході, має бути фіксованою.

Для того, щоб подолати ці обмеження крім таблиці з метаінформацією вводиться поняття таблиці розширень, при цьому до початкових базових таблиць додається ще одне поле, яке містить ідентифікатор запису. Таблиця розширень зберігає первинний ключ, посилання на запис базової таблиці та значення, яке має зберігатися в додатковому полі цієї таблиці у строковому вигляді. В свою чергу таблиця для збереження метаінформації трохи відрізняється від варіанту, запропонованого вище. Вона, як і раніше, містить ідентифікатор орендатора, назву колонки та її тип, а також додається поле, яке містить зовнішній ключ на ідентифікатор рядочку в таблиці розширень.

Такий підхід дозволяє додавати стільки додаткових полів, скільки орендатору потрібно для реалізації своїх бізнес-потреб, хоча в свою чергу додає певну складність до структури SQL запитів. Це рішення є найбільш оптимальним у випадку спільної бази даних для всіх орендаторів і найбільш широковживаним

(наприклад хмарною платформою force.com[1)]). При використанні окремої бази даних вищезгаданий спосіб може бути замінений звичайним додаванням полів до вже існуючих таблиць, оскільки такі дії будуть стосуватися лише одного орендатора і не вплинуть на спосіб збереження інформації інших клієнтів. Однак це призведе до різної кількості колонок у таблицях для кожного орендатора, що може призвести до певних складнощів під час розробки програми.

Незалежно від стратегії масштабування потрібно пам’ятати про необхідність адаптації програмної логіки до можливої появи додаткових полів, які можуть спричинити зміни у бізнес-логіці додатку.

1.2.6 Масштабування багатоорендних сервісів

Масштабування сервісу дозволяє застосувати певні стратегії в залежності від потреб для того, щоб зменшити вартість та підвищити ефективність роботи програми. Існує дві підходи до масштабування: горизонтальне та вертикальне [8]. Горизонтальне масштабування передбачає додавання паралельних компонентів (наприклад серверів) з однаковою функціональністю, в той час як ціль вертикального масштабування полягає у покращенні продуктивності вже існуючих ресурсів, таких як пам’ять, сховище або мережа. Необхідно розрізняти поняття масштабування застосунку, збільшення рівня навантаження, з яким може працювати додаток, і масштабування даних, тобто підвищенні ефективності роботи з ними.

У випадку роботи з даними використовуються дві стратегії масштабування: копіювання (replication) і ділення (partitioning). Ділення передбачає розбиття таблиці вертикально (тобто новоутворені таблиці містять однакову кількість рядочків, але кожна містить лише частину стовпчиків) та горизонтально (кількість стовпців однакова, але кількість рядочків менша). Горизонтальне ділення достатньо легко застосувати у випадку багатоорендності, оскільки його можна виконати за допомогою розрізання по ідентифікатору орендатора. Однак при

[1)] https://developer.salesforce.com/wiki/multi_tenant_architecture

виникненні проблем з продуктивністю роботи сервісу, які спричиняються тим, що багато користувачів паралельно користуються додатком, варто звертати увагу на те, що різні клієнти по-різному впливають на рівень навантаження. Тому більш ефективним рішенням буде поділ при якому на кожний сервер припадає однаковий рівень навантаження ніж якщо на кожному сервері буде однакова кількість клієнтів. В свою чергу, якщо проблеми стосуються значної кількості даних, які можуть проявлятися у занадто довгому часі роботи при виконанні запитів, то ділення бази даних на приблизно рівні частини буде найоптимальнішим розв'язанням проблеми.

Рішення про те, який з двох методів поділу доцільно застосувати має бути прийняте зважаючи на метрику, отриману в ході роботи програми. Крім того, з часом може виникнути необхідність перерозподілу даних, оскільки орендатори можуть змінювати свою поведінку. Інколи може трапитись ситуація, коли варто відокремити лише одного орендатора в окрему базу даних. Тоді зі зростанням кількості даних може бути застосоване вертикальне масштабування сховища до того часу, поки це можливо. Після того єдиним виходом буде поділ даних, однак ця ситуація буде відрізнятися від попередньої, тому що поділ буде застосований до бази даних, яка належить лише одному орендатору. Тоді необхідно аналізувати, відокремлення яких саме даних найменше знизить ефективність. Найчастіше до таких відносяться дані, які дуже рідко змінюються (як наприклад індивідуальний податковий номер).

Правильний вибір стратегії масштабування допомагає задовольнити потреби клієнтів, при цьому мінімізуючи вартість виконання пов'язаних з цим операцій і підвищуючи ефективність функціонування програми.

1.2.7 Переваги та недоліки використання багатоорендної архітектури

Використання багатоорендності дозволяє знизити вартість, оскільки передбачається спільне використання одного і того самого набору даних одразу декількома орендаторами. Крім того, додавання нового клієнта відбувається без потреби налаштування додаткових сховищ даних з боку розробника, оскільки це

передбачено реалізацією мультиорендності і відбувається автоматично. Це в свою чергу забезпечує новому клієнту можливість почати користуватися сервісом одразу після реєстрації.

Ще одним важливим фактором є можливість певної адаптації сервісу під потреби користувача без необхідності внесення змін в існуючу кодову базу, яка залишається однаковою для всіх користувачів. Однак масштаби такої кастомізації є значно меншими в порівнянні з тим, що доступно у випадку використання одноорендного застосунку.

Іншим негативним наслідком використання мультиорендної архітектури є великий розмір і складна структура баз даних, які одночасно використовуються багатьма клієнтами, що може призвести до збільшення часу очікування перед тим як отримати доступ до даних. Крім того, відмова серверу, де зберігається спільна інформація, вплине на роботу всіх користувачів, які належать до різних орендаторів.

РОЗДІЛ 2 ОПИС ВИКОРИСТАНИХ ТЕХНОЛОГІЙ

2.1 Spring Framework

Spring – це фреймворк для розробки застосунків на мові Java. На відміну від інших фреймворків, як наприклад Apache Struts, який призначений виключно для розробки веб застосунків, фреймворк Spring може використовуватися для створення будь-яких додатків на мові Java, включаючи автономні, корпоративні застосунки, веб-застосунки тощо. Головний принцип використання Spring полягає у мінімальній взаємодії з боку розробника для інтеграції ядра фреймворку у програмний продукт.

Необхідність у появі такого роду фреймворку була викликана складністю написання слабкозв'язаних програм, що майже унеможливлювало тестування корпоративних застосунків, які використовували з'єднання до зовнішніх ресурсів, таких як бази даних. Перший публічний випуск Spring 0.9 у 2003 році був заснований на матеріалі книги Рода Джонсона "Expert One-on-One J2EE Design and Development", у якій автор запропонував більш простий підхід до написання програм, який базувалося на звичайних Java класах (POJO – Plain Old Java Objects) та впровадженні залежностей (Dependency Injection - DI).

2.1.1 Інверсія управління та ін'єкція залежностей

Інверсія управління – принцип в програмній інженерії, коли процес створення і управління об'єктами передається контейнеру або фреймворку. На відміну від традиційного підходу, у якому компоненти коду викликають функції бібліотек, використання шаблону інверсії управління покладає на контейнер задачу керування процесом виконання програми, що включає в себе створення та зв'язування залежностей між собою. Це стає можливим за допомогою абстракції, яка досягається використанням абстрактних класів та інтерфейсів, та додаткової вбудованої поведінки.

Такий підхід у порівнянні з традиційним має декілька значних переваг. По-перше, це дозволяє зменшити кількість написаного коду і сконцентруватися на розробці бізнес-логіки програми не витрачаючи зусиль на створення додаткових

сервісів та компонент. По-друге, використання абстракції дозволяє значно простіше змінювати реалізацію об’єктів на різних стадіях розробки, як наприклад під час тестування.

Основними формами інверсії контролю є пошук та ін’єкція залежностей [9]. При першому підході контейнер має надати методи для того, щоб компоненти могли з їх допомогою знайти та отримати необхідні їм залежності, в той час як для роботи ін’єкції залежностей компоненти мають забезпечити наявність конструкторів або функцій для встановлення значень (сеттери - setters),через які контейнер зможе самостійно впровадити потрібні залежності. На відміну від попереднього випадку, пошук залежностей здійснюється досить рідко, а їх ін’єкція в основному відбувається під час створення компонентів за допомогою вищезгаданих допоміжних методів.

Основою Spring фреймворка є контейнер інверсії управління, який представлений інтерфейсом ApplicationContext. Його задача полягає в налаштуванні та керуванні Java об’єктів за допомогою рефлексії. Об’єкти, створені таким чином, отримали назву «керований об’єкт» або «бін». Для коректного управління ними контейнер фреймворка використовує метадані у форматі XML або анотації. В залежності від типу застосунку існує декілька імплементацій інтерфейсу ApplicationContext таких як ClassPathXmlApplicationContext та FileSystemXmlApplicationContext для автономних додатків та WebApplicationContext для веб застосунків.

Впровадження залежностей у контексті Spring може бути здійснено трьома способами: через конструктор, метод установки або безпосередньо у поле. Однак останній спосіб не рекомендується застосовувати через наявність певних недоліків. Каркас фреймворку надає можливість ін’єктувати не лише прості значення, а й інші компоненти, колекції, властивості, що були визначені зовні і компоненти з інших фабрик.

2.1.2 Spring Boot

Spring Boot це розширення фреймворку Spring, яке значно спрощує розробку Spring додатків за допомогою набору попередньо налаштованих стартерів (starter). Більшість проектів Spring Boot потребує мінімальної конфігурації, що збільшує продуктивність і дозволяє уникнути написання шаблонного коду тим самим зменшуючи час розробки. Spring Boot Starter являє собою комбінацію взаємопов'язаних залежностей, інкапсульованих в єдине ціле, що гарантує сумісність версій.

Spring Boot автоматично налаштовує проект виходячи з того, які залежності були в нього включені. Це відбувається за допомогою анотації @EnableAutoConfiguration, яка активує механізм автоконфігурації. Разом з нею використовуються дві інші анотації: @ComponentScan, за допомогою якої відбувається сканування вказаних пакетів у пошуку компонент застосунку, та @Configuration, яка реєструє додаткові біни в контейнері ApplicationContext або дозволяє імпортувати інші конфігураційні файли. Найчастіше трьох вищезгаданих анотації з дефолтними значеннями використовується анотація @SpringBootApplication.

Spring Boot створений на основі Spring однак між двома фреймворками існують певні відмінності. Головною метою Spring було забезпечення гнучкості за допомогою принципу інверсії контролю та побудова слабко зв'язаного програмного коду, в той час як Spring Boot фокусується на зменшенні кількості коду та полегшенні його запуску. Однак це може створювати певні незручності як наприклад зменшення контролю над налаштуваннями та поява непотрібних залежностей, які включаються разом із усім стартером. Іншою відмінністю є наявність вбудованого серверу, що полегшує розгортання застосунку та підтримка декількох баз даних, що зберігаються у пам'яті.

2.1.3 Spring Data

Spring Data – це високорівневий проект, який націлений на забезпечення уніфікованого та простого способу доступу до сховищ даних і надає можливості

роботи як з реляційними так і з нереляційними базами даних, фреймворками, які підтримують розподілену модель обчислень map-reduce, та хмарними сервісами. Spring Data включає в себе низку підпроектів та модулів [10], реалізація яких є специфічною для кожної бази даних. Одним з таких модулів є Spring Data JPA, що являє собою розширену підтримку JPA та спрямований на вирішення типових проблем пов'язаних з використанням технологій доступу до даних у застосунках, що використовують Spring фреймворк.

JPA – офіційна специфікація для мови Java, яка описує стандарт взаємодії застосунку з реляційними базами даних, а саме як класи співставляються таблицям, інтерфейси для основних CRUD операцій, вигляд SQL запитів тощо. Найпоширенішими реалізаціями JPA є Hibernate, Eclipse Link та ObjectDb.

2.2 Hibernate

При написанні програм швидше за все виникне потреба роботи з даними, така як збереження, отримання, видалення чи зміна. У мові Java для цього існує низькорівневий прикладний програмний інтерфейс JDBC (Java Database Connectivity), який надає можливість надсилати запити прямо до реляційної бази даних. Однак, написання будь-якого, навіть найпростішого запиту за допомогою JDBC вимагає багато допоміжного коду.

Hibernate – це ORM (Object Relational Mapping) фреймворк, який є надбудовою над стандартним інтерфейсом JDBC і націлений на те, щоб значно спростити операції з даними, замість того даючи можливість програмісту сконцентрувати увагу на розробці бізнес-логіки застосунку. Hibernate вимагає мінімального знання SQL, але для того, щоб мати можливість використовувати фреймворк на повну потужність розробник повинен мати глибокі знання про функціонування реляційних баз даних.

2.2.1 Переваги та недоліки використання ORM фреймворків

По-перше, використання ORM фреймворків дозволяє зменшити кількість шаблонного коду для написання запитів, а також полегшує процес валідації та розбиття на сторінки (pagination), що дає змогу розробнику зосередитися на

розробці бізнес логіки та зменшити час написання та відлагодження програмного коду.

По-друге, синтаксис різних діалектів мови SQL, як наприклад Oracle, PostgreSQL або MySQL, може значно відрізнятися один від одного, і завдяки тому, що Hibernate дозволяє максимально зменшити використання «чистого» SQL синтаксису, з'являється можливість змінювати постачальників SQL з мінімальними змінами в коді.

Крім того, більшість ORM фреймворків автоматично генерують підготовлені запити (prepared statement), що запобігає ризику SQL ін'єкцій.

Але хоча використання Hibernate дає можливість значно швидко й ефективно розробляти додатки, які використовують дані, тим не менш існують і певні недоліки через те, що між об'єктно-орієнтованим представленням даних і реляційною моделлю існує значна різниця. По-перше, через відсутність у реляційному представлені таких концепцій як наслідування і агрегація постає певна проблема у коректному відображенні цих зв'язків між класами. І хоча є певні підходи до вирішення цієї проблеми, вони тим не менш мають свої обмеження.

По-друге, у представленні складних ієрархічних моделей часто виникає проблема зі швидкістю виконання запитів. По-третє, іноді брак досвіду роботи з фреймворком і відсутність розуміння того, які запити використовуються для отримання даних, може призвести до таких наслідків, як проблема n+1 вибору або проблема декартового добутку [11].

2.2.2 Стратегія іменування Hibernate

Частиною процесу встановлення зв'язку між об'єктно-орієнтованою та реляційною моделлю є процес співставлення імені об'єкта та його відповідної назви у базі даних. Фреймворк Hibernate використовує для цього два етапи найменування: внутрішній (Implicit Naming Strategy) та фізичний (Physical Naming Strategy).

Внутрішня стратегія призначена для того, щоб визначити логічне ім'я сутності або її атрибуту і може бути налаштована двома способами – автоматично

визначена, використовуючи клас, який імплементує інтерфейс org.hibernate.boot.model.naming.ImplicitNamingStrategy або програмно налаштована за допомогою анотацій (@Column, @Table тощо). Фреймворк постачає чотири реалізації інтерфейсу ImplicitNamingStrategy [12], але розробник може самостійно створити свою власну імплементацію. Автоматичне визначення імені буде здійснюватися кожного разу в тому випадку, якщо логічне ім’я не вказано явно. Для того, щоб вказати яку реалізацію використовувати, необхідно встановити властивість hibernate.implicit_naming_strategy.

Фізична стратегія найменування передбачає конвертацію логічного імені у відповідне ім’я у базі даних. Ідея полягає у визначенні власних правил найменування замість того, щоб безпосередньо вказувати ці імена у коді. За замовчанням будуть використовуватися логічні імена, визначені внутрішньою стратегією. Для того, щоб змінити цю поведінку необхідно реалізувати клас, який імплементує інтерфейс org.hibernate.boot.model.naming.PhysicalNamingStrategy і вказати значення властивості hibernate.physical_naming_strategy.

2.2.3 Hibernate та Spring

Встановлення з’єднання з базою даних, виконання запитів та керування транзакціями реалізується за допомогою менеджера сутностей. Багато додатків вимагають наявності декількох з’єднань в процесі роботи. Так наприклад для веб застосунків цілком природньо відкривати нове з’єднання, при цьому використовуючи новий менеджер сутностей, з кожним HTTP запитом. Головна роль фабричного класу, який реалізовується інтерфейсом EntityManagerFactory, полягає у створенні та керуванні менеджерами сутностей, шляхом ефективного розподілу ресурсів між ними (наприклад пул сокетів). Створення фабрики менеджерів є затратною операцією з точки зору часу і ресурсів, однак це одноразова дія, яка зазвичай відбувається перед початком роботи програми.

При виборі налаштувань для роботи Hibernate з фреймворком Spring необхідно визначитися зі способом створення та керування менеджером сутностей, який імплементує інтерфейс EntityManager, і з’єднання з базою даних. Для цього

існує два різних підходи: керувати менеджером вручну, або покласти цю функцію на Java EE контейнер. Для того, щоб пояснити суттєву різницю між обидвома підходами необхідно ввести поняття контексту збереження (Persistence Context), який завжди прив’язаний до менеджера сутностей.

Контекст збереження керує набором сутностей, які мають бути збережені до бази даних, наприклад переводячи їх з одного стану в інший [13]. Кожного разу, при отриманні сутності з бази даних, вона поміщається у контекст збереження (який у Hibernate представлений сесією). Коли наступний раз виникає потреба у цій сутності, то замість того, щоб відправляти запит до бази даних, повертається та, що зберігається в персистентному контексті. Таким чином контекст збереження відіграє роль кешу першого рівня. Всі посилання на об’єкти у контексті є слабкими посиланнями (weak references). Якщо сутність змінюється, то вона певним чином помічається і коли виникне потреба у синхронізації об’єктів, які розміщені у пам’яті та вмісту бази даних, тобто в момент коміта транзакції або явного виклику метода flush(), то до всіх помічених сутностей буде застосована операція оновлення даних. Всі інші сутності, які більше не використовуються програмою, будуть знищені при зборі сміття і автоматично видалені з персистентного контексту. Контекст збереження гарантує, що кожній сутності у базі даних відповідає не більше однієї сутності, яка зберігається в оперативній пам’яті, і відноситься до одного і того самого менеджеру сутностей.

Якщо функція управління менеджером сутностей покладається на контейнер [14], то в цьому випадку контейнер самостійно створює екземпляр EntityManager з фабричного класу EntityManagerFactory, а також розпочинає транзакцію, завершує її і обробляє помилки, що виникають під час транзакції.

Часто працюючи з контейнером Java EE виникає потреба у створенні менеджера транзакцій. Якщо транзакція охоплює декілька компонент одночасно, то вони зазвичай потребують доступу до одного й того ж контексту збереження. Тоді контейнер намагається розподілити менеджери сутностей таким чином, щоб компоненти, які працюють в межах однієї транзакції, отримали менеджерів з одним

і тим самим контекстом збереження. В цьому випадку персистентний контекст прив’язаний до транзакції, і після її завершення усі сутності, які були змінені під час транзакції будуть оновлені у базі даних. [13]

В той час, якщо необхідності у спільному персистентному контексті не виникає, то у кожного менеджера сутності буде свій власний ізольований контекст збереження. Ця ситуація відповідає управлінню життєвим циклом менеджера сутності вручну.

2.2.4 Багатоорендність в Hibernate

Для коректного налаштування мультиорендності в Hibernate в режимах окремою бази даних або окремої схеми розробнику необхідно реалізувати два контракти [15]: MultitenantConnectionProvider та CurrentTenantIdentifierResolver. Перший відповідає за те, яким чином буде вибиратися з’єднання в залежності від орендатора. Другий контракт описує що саме буде вважатися ідентифікатором клієнта.

Він використовується у ситуації, яка полягає у поєднанні багатоорендності разом із принципом поточної сесії, тобто у випадку, коли одна транзакція використовує один і той самий персистентний контекст і повертає одну і ту саму сесію. У разі відсутності сесії фреймворк буде змушений створити нову, але при створенні виникне потреба специфікувати ідентифікатор орендатора. У цей момент виникне потреба у наявності реалізації контракту для встановлення поточного клієнта.

2.3 Міграції баз даних

Міграція бази даних – це процес керування змінами схеми реляційної бази даних, що дає змогу повернутися до новішої чи старішої версії [**Error! Reference source not found.**]. Процесом можна керувати вручну або використовувати для цього засоби для міграції такі як Flyway (flywaydb.org) або Liquibase (liquibase.org) [16].

Зазвичай засоби для міграцій баз даних створюють додаткову таблицю з метаінформацією, у якій зберігається детальна інформація про версії. Задача

програміста полягає у написанні команд для зміни структури бази даних, які можуть бути згруповані у набір змін (changeset), і відслідковуються системою контролю. Після запуску скриптів зі змінами за допомогою метаінформації робиться висновок чи потрібно відпрацьовувати зміни, чи вони вже були застосовані раніше. Також перевіряється, чи не було модифікацій до раніше написаних і відпрацьованих скриптів. Це відбувається шляхом підрахування контрольної суми для кожного набору змін. Якщо контрольна сума не збігається з тою, яка була обрахована раніше, то система повідомляє про помилку.

Не існує єдиного стандарту для написання скриптів змін, наприклад система Liquibase наразі підтримує чотири формати, у яких можуть зберігатися зміни: SQL, JSON, XML та YAML. При використанні будь-якого з зазначених форматів крім SQL, Liquibase генерує специфічний код в залежності від тої бази даних, яка використовується. Система підтримує майже всі популярні СУБД[1)], а також існують готові розширення для додаткового функціоналу.

Використання інструментів для міграції даних значно полегшує відтворення певного стану бази даних, зокрема це може знадобитися в процесі розгортання застосунку на сервері. Іншою важливою перевагою є незалежність написання скриптів від СУБД, яка використовується, однак в деяких випадках при зміні постачальника бази даних все ж таки може знадобитися певна їх модифікація.

2.4 Аспектно-орієнтоване програмування

Аспектно-орієнтовано програмування (АОП) – парадигма програмування, яка дозволяє виокремити наскрізну функціональність від основної логіки програми. Прикладами такої функціональності може бути протоколювання, перевірка прав доступу, обробка помилок тощо. На відміну від ООП, де основною структурною одиницею був об’єкт, в АОП таку роль буде відігравати аспект.

[1)] Список СУБД, які підтримуються системою Liquibase, можна знайти на офіційному сайті (https://www.liquibase.org/get-started/databases)

2.4.1 Основні поняття аспектно-орієнтованого програмування

Аспект – це модуль функціональності, яка наскрізно проходить через декілька класів.

Точка з'єднання – конкретне місце під час виконання програми, де буде вставлена логіка засобами АОП. Прикладами може бути виклик методу або ініціалізація класу.

Порада – конкретний набір кроків, який буде виконаний за допомогою АОП під час роботи програми. Існує декілька типів порад: ті, що передують точці з'єднання, ті які виконуються після успішного завершення виконання програми у точці з'єднання, ті що виконуються в разі виникнення помилки, ті, які виконуються незалежно від того чи метод виконався успішно чи в процесі його виконання сталася помилка, і ті, що виконуються до і після місця з'єднання.

Зріз – це предикат, який об'єднує декілька точок з'єднання певною умовою, і дозволяє виконати пораду у всіх точках з'єднання, які відповідають зрізу. Прикладом зрізу може слугувати ім'я методу.

Ціль – об'єкт, чий потік виконання змінюється певними порадами.

Зв'язування – процес вставки аспектів в певне місце прикладного коду. В залежності від типу АОП це може відбутися на стадія компіляції, в момент загрузки або під час виконання програми. Зв'язування на етапі компіляції ще називається статичним і може бути реалізовано за допомогою AspectJ компілятора. Цей спосіб є дуже ефективним оскільки після внесення необхідних змін в байт код більше не потребується виконувати жодних додаткових дій під час роботи програми. Однак при внесенні будь-яких, навіть таких найпростіших змін як додавання нової точки з'єднання вимагає перекомпіляції всієї програми.

Схоже до попереднього випадку зв'язування у процесі виконання програми носить назву динамічного зв'язування і може бути досягнено шляхом створення проксі-об'єкта. Саме такий підхід і використовується в модулі Spring AOP.

2.4.2 AspectJ та модуль Spring AOP

Фреймворк Spring підтримує два варіанти використання АОП: за допомогою проксі об’єктів та використовуючи AspectJ. AspectJ – це розширення мови Java для підтримки аспектно-орієнтованого програмування створене компанією PARC. Воно складається з двох основних частин: специфікації мови, що включає граматику і семантику, та імплементації, тобто процес зв’язування. AspectJ підтримує всі три типи зв’язування, найпростішим з яких вважається той, що відбувається під час роботи програми Але незалежно від вибору, результат виконання програми буде ідентичний.

В свою чергу модуль Spring AOP не є повноцінною реалізацією АОП і направлений на вирішення найпоширеніших задач, які вимагають аспектно-орієнтованого рішення. Для цього використовуються динамічні J2SE або CGLIB (Code Generation Library) проксі-об’єкти. Проксі – це об’єкт, який слугує «обгорткою» для іншого об’єкта зберігаючи його інтерфейси і додаючи до нього новий функціонал. Зазвичай такі проксі делегують виконання логіки реальному об’єкту, при цьому виконуючи певні дії до або/і після виклику метода. Звідси випливає факт, що внутрішній об’єкт не має доступу до проксі, а отже якщо виклик методу буде здійснено всередині даного об’єкту, то додатковий функціонал, як наприклад АОП, не буде виконаний. При виборі стратегії створення проксі фреймворк надає перевагу динамічному підходу, який буде використовуватися завжди в тому випадку, якщо об’єкт імплементує хоча б один інтерфейс. В іншій ситуації буде створений CGLIB проксі.

Технологія Spring AOP має певні обмеження в порівнянні з використання AspectJ. Зокрема не можна створювати аспекти для фінальних класів, оскільки вони не можуть бути перевизначені, а також це саме стосується статичних і фінальних методів. Також, оскільки у випадку AspectJ створення проксі відбувається на етапі компіляції, виконання АОП коду за допомогою модуля Spring AOP може бути від 8 до 35 раз повільніша [18].

РОЗДІЛ 3 ПРАКТИЧНА ЧАСТИНА

3.1 Засоби розробки

Для реалізації проекту була вибрана версія Java 11, що є останньою довгостроково підтримуваною версією мови. У якості засобу для автоматизації роботи з програмними проектами була надана перевага Apache Maven. Для написання програми використовувалося середовище Eclipse для корпоративної та веб-розробки Java застосунків, оскільки, на відміну від іншої популярної IDE Intellij Idea, Eclipse є безкоштовним для використання і крім того, він підтримує значну кількість доступних для встановлення і налаштування плагінів, які значно полегшують процес розробки.

3.2 Визначення ідентифікатора орендатора

Перед початком імплементації варіантів для реалізації багатоорендності необхідно створити певний модуль, який буде містити функціонал, спільний для усіх трьох підходів.

Ідентифікатор поточного орендатора зберігається у строковому форматі в класі TenantContext. Так як на кожен запит виділяється новий потік, для коректного збереження використовується змінна типу ThreadLocal, яка гарантує що кожен потік буде мати свою власну копію ідентифікатора. Сам ідентифікатор передається у заголовку X-TENANT-ID, який перехоплюється за допомогою класу TenantInetrceptor. Якщо значення заголовку не визначено, то ідентифікатор встановлюється рівним null.

Крім того, для більш ефективних роботи програми та розподілу ресурсів потоки виділяються з пулу з’єднань і тому використовуються декілька разів для обробки різних запитів. Зважаючи на це, після завершення запиту та перед поверненням потоку до пулу з’єднань необхідно очистити значення змінної ThreadLocal для того, щоб помилкове значення не було використано при наступному зверненні до цього потоку.

3.3 Окрема база даних

Фреймворк Spring Boot автоматично сканує пакети та конфігурує джерела даних, та крім того налаштовує пули з'єднань такі як HikariCP, Apache Tomcat, Common DBCP або Oracle UCP [19], виходячи з того, які залежності наявні у проекті. Налаштування джерел даних відбувається за допомогою параметрів spring.datasource.*, які можна вказати наприклад у файлі властивостей application.properties або application.yaml. Якщо не вказати адресу з'єднання, за яку відповідає параметр spring.datasource.url, то Spring Boot спробує підключитися до однієї з вбудованих баз даних як наприклад H2, HSQL або Derby.

Незважаючи на те, що в більшості випадків автосконфігурованих джерел даних достатньо, іноді з'являється потреба в їх адаптації для вирішення певних проблем.

3.3.1 Налаштування джерел інформації та менеджерів сутностей

Для налаштування джерела інформації необхідно створити бін, що реалізує інтерфейс DataSource, який потім буде використовуватися фреймворком для керування підключенням до бази даних. Додатково у конфігураційному файлі можна вказати значення, які будуть приймати такі властивості як ім'я користувача у базі даних, пароль, розмір пула з'єднань тощо. Крім того, для цих цілей фреймворк Spring Boot також забезпечує наявність службового класу DataSourceBuilder, який можна ініціалізувати використовуючи об'єкт типу DataSourceProperties.

Усе вищеописане справедливо і для налаштування більше ніж одного джерела інформації лише з тією відмінністю, що один з бінів необхідно помітити анотацією @Primary для того, щоб фреймворк в подальшому міг використати його при автоконфігурації. Ще однією відмінністю є те, що в цьому випадку виникає необхідність вказувати значення параметрів для декількох джерел інформації з різними префіксами. Для того, щоб специфікувати, які з налаштувань використовувати для конкретного джерела інформації перед фабричним методом

для створення біна DataSourceProperties вказується анотація @ConfigurationProperties, яка специфікує коректний префікс.

Крім створення джерел інформації для коректної роботи застосунку модуль JPA дозволяє програмно налаштовувати певні властивості [20], щоб отримати більше контролю над створеними репозиторіями. Однією з таких властивостей є entity-manager-factory-ref, яка явно зв’язує фабрику менеджерів сутності і певне сховище даних. Зазвичай використовується в тому випадку, коли передбачається використання двох і більше фабрик. Якщо властивість не вказана, то фабрика буде автоматично створена при запуску програми контекстом фреймворка. Іншою важливою конфігурацією є transaction-manager-ref, яка явно зв’язує менеджер транзакцій і репозиторій. Як і в попередньому випадку використовується при наявності більше одного менеджеру або/і більше однієї сконфігурованої фабрики. Якщо значення властивості відсутнє в явному вигляді, то за умовчанням буде використовуватися менеджер транзакцій, створений контекстом Spring.

Модуль Spring JPA може бути активований як за допомогою XML конфігурації так і за допомогою анотацій. Для того, щоб налаштувати JPA сховища даних у XML файлі треба вказати елемент <repositories /> та значення його атрибутів. Інший спосіб полягає у використанні анотації @EnableJpaRepositories у конфігураційному класі Java. Але для коректної роботи модулю Spring Data JPA необхідно знати розташування репозиторіїв, які відносяться до певного джерела інформації. За замовчанням фреймворк буде шукати їх декларацію в головному пакеті програми та всіх його підпакетах, тому якщо місцезнаходження репозиторія відрізняється від зазначеного вище, то його треба вказати в явному вигляді за допомогою параметру анотації basePackages (або XML атрибута base-package елемента <repositories />).

При використанні двох і більше джерел інформації необхідно явно вказати які репозиторії і сутності відносяться до певного джерела. Для цього їх необхідно помістити в різні пакети, при чому жоден з пакетів, у якому містяться репозиторії одного джерела, не повинен бути підпакетом, де зберігаються репозиторії іншого

джерела, і, як було зазначено вище, вказати назви пакетів як атрибут basePackages. Для сутностей має бути виконана схожа послідовність дій з тією різницею, що пакети сутностей реєструються за допомогою методу setPackagesToScan(), який наявний у фабрики сутностей.

Для того, щоб прив’язати джерело інформації до певної фабрики використовується метод setDataSource(). Також при створенні фабрики можна вказати додаткові властивості, як наприклад стратегію найменування або діалект.

У даному проекті при виборі способу керування сутностями була надана перевага більш гнучкому методу за допомогою контейнера, тому фабрика сутностей є екземпляром класу LocalContainerEntityManagerFactoryBean з пакету org.springframework.orm.jpa (для програмного управління існує клас LocalEntityManagerFactoryBean). Розташування репозиторіїв і класів сутностей вказано у конфігураційному файлі application.yaml за допомогою властивостей multitenancy.common.repository.packages та multitenancy.common.entityManager.packages відповідно. Джерело даних створюється за допомогою властивостей з префіксом multitenancy.common.datasource.

Стратегія найменування сутностей та інші додаткові властивості залишаються без змін, тобто використовуються значення за замовчанням. Так як ім’я для біна фабрики сутностей явно не вказано, то в якості нього буде використана назва фабричного методу для його створення, тобто commonEntityManagerFactory, що й буде передано як значення параметру entityManagerFactoryRef анотації @EnableJpaRepositories. Аналогічна процедура відбувається для вказання менеджера транзакцій. Повну реалізацію фабрики сутностей та менеджера транзакцій наведено на рисунку А.1.

Налаштовані таким чином джерело даних і менеджер сутностей призначені для встановлення зв’язку і виконання операцій з базою даних common, у якій зберігається спільна інформація, яка доступна всім орендаторам системи і навіть незареєстрованим клієнтам.

Налаштування джерел інформації для орендаторів є більш складною операцією, оскільки потрібно передбачити можливість динамічного додавання нових клієнтів. Як було зазначено у 2.2.4, необхідно реалізувати два контракти: CurrentTenantIdentifierResolver та MultitenantConnectionProvider. У першому випадку потрібно створити клас (рисунок А.2), який імплементує інтерфейс org.hibernate.context.spi.CurrentTenantIdentificationResolver і реалізувати два методи: resolveCurrentTenantIdentifier() та validateExistingCurrentSessions(). Перший метод повинен повертати ідентифікатор поточного орендатора, який можна отримати за допомогою статичної функції об’єкта TenantContext. Другий метод повертає логічне значення True або False і вказує чи потрібно перевіряти, що ідентифікатори всіх поточних сесій співпадають.

Реалізація другого контракту дозволяє отримати потрібне джерело інформації в залежності від вибраного орендатора. Для вирішення цієї задачі джерела інформації орендаторів зберігаються в кеші tenantDataSources, який представлений класом LoadingCache. Даний об’єкт за необхідності створює новий об’єкт класу DataSource, базуючись на інформації про орендатора отриманої з головної бази даних, а також коректним чином закриває з’єднання і прибирає з кешу ті об’єкти, які певний час не використовувалися. Задача створення джерела інформації покладається на допоміжну приватну функцію createAndConfigureDataSource(), а у якості його реалізації був вибраний пул з’єднань HikariCP. Таке рішення було прийнято після вивчення швидкодії роботи, яка забезпечується високо ефективною реалізацією [21]. За замовчанням інтервал часу, через який невикористані джерела інформації буде знищено складає 10 хвилин або вказується у файлі конфігурацій за допомогою властивості multitenancy.datasource-cache-expireAfterAccess.

Головну роль відіграють два методи: selectDataSource() та selectAnyDataSource(). Перша функція звертається до кешу і повертає джерело даних, яке відповідає поточному ідентифікатору. У випадку, коли не вдалося встановити орендатора буде викликаний другий метод, який поверне джерело

інформації, що вказує на спільне сховище даних. Зазвичай цей метод викликається на початку при запуску програми, коли інформація про орендаторів ще може бути відсутня, для того щоб валідувати схему або визначити діалект.

Після створення двох вищеописаних класів з’являється можливість створити і налаштувати фабрику менеджерів сутностей для орендаторів. Реалізація TenantPersistenceConfig, на який покладається виконання цієї функції схожий до створених вище подібних класів, однак на відміну від них при створенні цієї фабрики у якості додаткових властивостей вказуються тип багатоорендності, в даному випадку DATABASE та біни, які реалізують два вищеописані контракти.

Дані про орендаторів можуть бути отримані за допомогою GET запиту, який посилається по зазначеному у файлі властивостей URL і їх формат має відповідати структурі класу Tenant.

3.3.2 Додавання нових орендаторів

Додавання нових орендаторів має відбуватися динамічно, тобто без необхідності перезапуску додатку. Для цього бібліотека передбачає наявність API, за допомогою якого можна передати інформацію про нового орендатора. Це викличе процедуру створення джерела даних та виконання команди для створення нової бази даних, користувача, а також надання йому відповідних прав доступу. Після цього для новоствореної бази даних будуть запущені скрипти, якщо відповідні налаштування були вказані в конфігураційному файлі.

Процес створення бази даних є значно повільнішим за виконання звичайних команд, що видно з Рисунок 3.1.

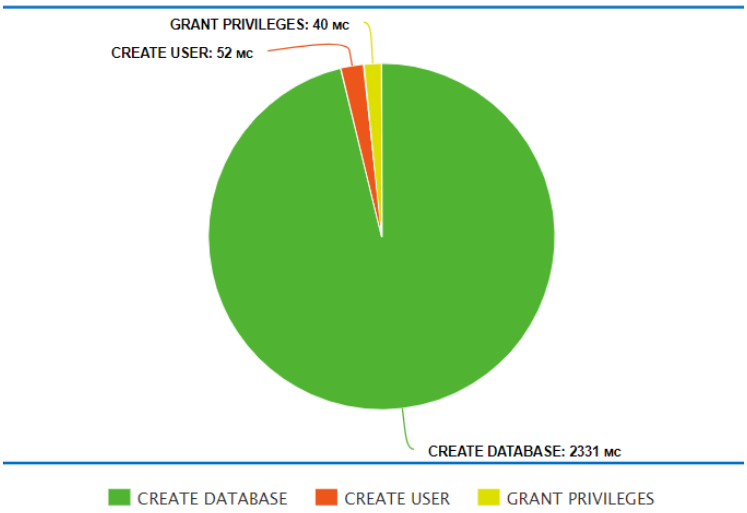

Рисунок 3.1 – Час виконання команд для створення і налаштування бази даних

3.3.3 Використання Liquibase

За замовчанням Spring Boot автоконфігурує систему Liquibase при запуску програми. Для цього потрібно лише додати необхідну залежність у файл зборки pom.xml. Для запуску Liquibase скриптів фреймворк використовує головне джерело даних DataSource, або у випадку декількох наявних, помічене анотацією @Primary. Якщо виникає потреба у використанні іншого джерела даних, то над таким класом має бути наявна анотація @LiquibaseDataSource.

При запуску програми система Liquibase шукає головний файл змін у папці db/changelog під назвою db.changelog-master.yaml. Інші, відмінні від стандартних шлях і назву файлу можна вказати у файлі властивостей.

Головний файл змін може не лише безпосередньо зберігати зміни і групи змін, а й вказувати на інші файли, у яких зберігаються скрипти. Це використовується для більш зручного структурування і управління файлами змін.

Оскільки документація системи Liquibase сконцентрована на написанні скриптів у форматі XML, була надана перевага саме такому способу. Для кращого управління скриптами існують рекомендації для розділення всіх змін на окремі файли.

Система Liquibase автоматично створює дві допоміжні таблиці. Перша, під назвою databasechangelog використовується для відслідковування того, які із скриптів вже були запущені. Ця таблиця не містить первинного ключа для того, щоб уникнути обмежень на його довжину, що є специфічним для кожної СУБД. Замість цього використовується той факт, що ідентифікатор скрипта, його автор та назва файлу разом мають формувати унікальну комбінацію. Для того, щоб уникнути конфліктів у випадку паралельного запуску декількох скриптів, Liquibase використовує другу таблицю під назвою databasechangeloglock. Однією з колонок є значення блокування, яке встановлюється рівним одиниці, коли для певної бази даних запущені скрипти міграцій і нулю у всіх інших випадках.

При запуску програми виникає необхідність створення баз даних, а також відпрацювання скриптів Liquibase для орендаторів, чия інформація була отримана за допомогою GET запиту. Оскільки така процедура займає значний проміжок часу, було прийнято рішення розпаралелити цю задачу таким чином, що кожен потік буде працювати з окремим орендатором. Порівняння швидкості виконання синхронних і асинхронних дій наведена на Рисунок 3.2.

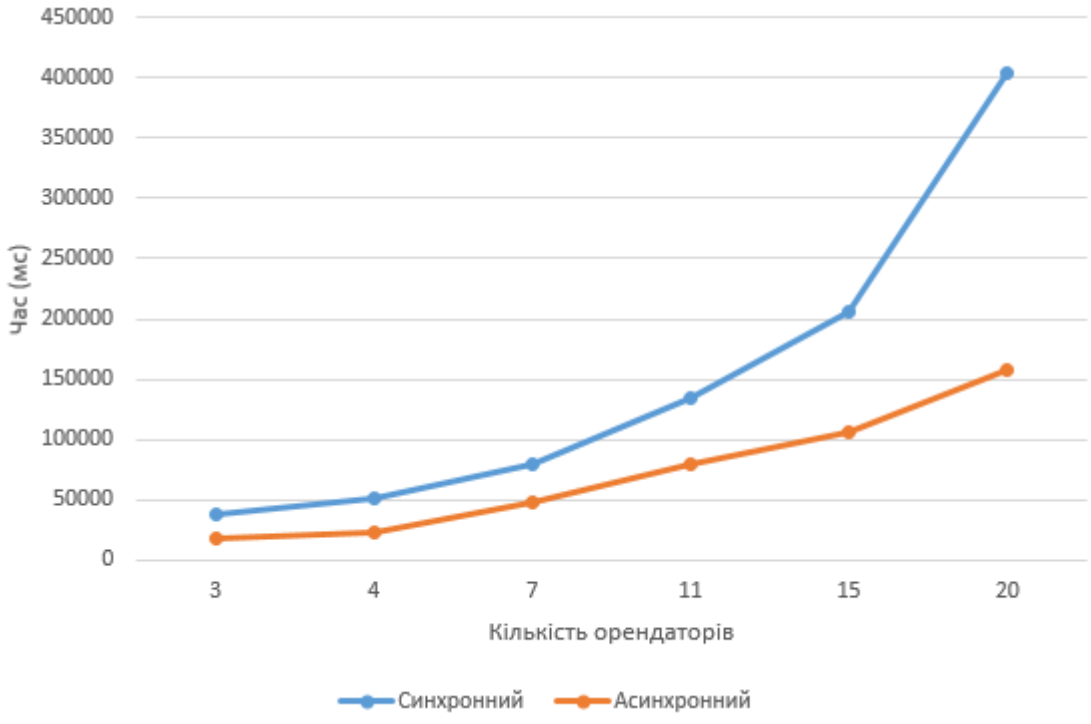

Рисунок 3.2 – Швидкість синхронного і асинхронного запуску програми

3.4 Окрема схема

Випадок окремої схеми для кожного орендатора у реалізації дуже схожий з попереднім підходом. Налаштування фабрики менеджерів сутності для спільної інформації нічим не відрізняється, а сама інформація тепер зберігається у схемі public. Конфігурація фабрики для орендаторів також залишається майже незмінною за винятком стратегії багатоорендності фреймворка Hibernate, яка в цьому випадку приймає значення SCHEMA.

Реалізація класу MultitenantConnectionProvider спрощується, тому що тепер немає необхідності створювати нове джерело інформації. Замість цього при кожному зверненні існуючому джерелу встановлюється коректне значення схеми.

Додавання нових орендаторів зводиться до виклику команди CREATE SCHEMA, яка виконується значно швидше команди для створення бази даних. Однак тепер розпаралелювання міграцій Liquibase при запуску програми майже не дасть виграшу у часі, тому що відповідно до 3.3.3 для уникнення конфліктів система Liquibase блокує доступ до бази даних при виконанні скриптів. Тому, незважаючи на те, що фактично відпрацювання скрипів у різних схемах не може спричинити конфліктів, всі операції будуть виконуватися послідовно.

Ідентично до випадку окремої бази даних інформація про існуючих у системі орендаторів може бути отримана за допомогою GET запиту до API, яке специфікується у конфігураційному файлі, але тепер формат даних вимагає лише наявності ідентифікатора орендатора і назву схеми.

3.5 Дискримінатор

На відміну від двох попередніх реалізацій використання дискримінатора для розділення даних орендаторів значно відрізняється. Це в першу чергу зумовлено тим, що Hibernate не передбачає готове рішення цієї задачі, і крім того, якщо у випадку використання окремої бази даних або окремої схеми потрібно було змінювати джерела даних або властивості джерела даних, то зараз виникає потреба зміни SQL запиту. Цю ситуацію можна умовно розділити на дві підзадачі: внесення коректного значення ідентифікатора у таблицю у випадку збереження інформації,

специфічної для кожного орендатора, та додавання умови WHERE під час отримання даних зі сховища.

Для цього було вирішено скористатися механізмом EntityListener, який дозволяє втручатися у життєвий цикл JPA об’єктів. Крім цього, був створений інтерфейс TenantAware, у якому присутній єдиний метод setTenantId(). Таким чином, під час видалення, зміни або збереження даних до сутності буде додане поле ідентифікатора орендатора. Для того, щоб при кожному запиті додавалася умова WHERE, був використаний механізм фільтрації Hibernate який представлений анотацією @Filter.

Щоб поєднати EntityListener і фільтрацію був створений абстрактний клас AbstractBaseEntity, який імплементує створений раніше інтерфейс TenantAware та містить єдине поле tenantId. Усі сутності, які зберігають інформацію, специфічну для кожного користувача повинні наслідувати даний клас. Його реалізація наведена на рисунку А.3.

Об’єкт Session створюється динамічно в процесі виконання програми, до нього неможливо неможливо застосувати фільтр перед початком роботи. Для цього необхідно скористатися можливостями, які надає аспектно-орієнтовано програмування. Оскільки об’єкт сесії не є біном, що управляється контейнером Spring, тому замість вбудованої у фреймворк підтримки АОП потрібно скористатися AspectJ.

Процес налаштування зв’язування на етапі загрузки (Load Time Weaving - LTW) можна умовно розділити на чотири етапи: конфігурацію аспектів з використанням XML файлу, налаштування фреймворку Spring для реалізації підтримки АОП, включення необхідних залежностей та передача коректних параметрів при кожному запуску застосунку.

Такі анотації як @Aspect, @Around та інші з пакету org.aspectj.package створені для розпізнавання компілятором AspectJ, але на відміну від модуля Spring AOP, AspectJ не може автоматично сканувати пакети в пошуках необхідних анотацій. Для цього у папку META-INF необхідно покласти файл aop.xml з

потрібною конфігурацією. Тег <include>, який є внутрішнім по відношенню до тегу <weaver> вказує пакети, які мають бути просканованні в пошуку точок з'єднання. Чим менша буде область пошуку, тим швидше буде відбуватися процес зв'язування. Тег <aspect> специфікує місцезнаходження класу, поміченого анотацією @Aspect.

Активація AspectJ у фреймворку Spring відбувається за допомогою анотації @EnableLoadTimeWeaving або відповідної конфігурації у XML файлі. За замовчанням, для початку роботи AspectJ потребує наявність вищезгаданого файлу aop.xml. Для того, щоб програма видавала помилку, якщо такого файлу не було знайдено, для анотації @EnableLoadTimeWeaving необхідно вказати параметр aspectjWeaving зі значенням ENABLED (у випадку XML конфігурації існує відповідний атрибут).

Для коректної роботи необхідно специфікувати аргументи, які будуть передаватися JVM при запуску програми. Це можна зробити або за допомогою файлу POM (Рисунок 3.3.а) або за допомогою аргументів командного рядка (Рисунок 3.3.б).

```
<agents>
    <agent>${project.build.directory}/spring-instrument-${spring-framework.version}.jar</agent>
    <agent>${project.build.directory}/aspectjweaver-${aspectj.version}.jar</agent>
</agents>
```

а)

```
java -javaagent:spring-instrument.jar -javaagent:aspectjweaver.jar -jar app.jar
```

б)

Рисунок 3.3 – Аргументи для запуску програми: а) – у POM файлі; б) – за допомогою командного рядка

ВИСНОВКИ

В даній роботі було досліджено три основні способи реалізації багатоорендності, а також проаналізовано їх переваги та недоліки. Надання переваги одному з них в порівнянні з іншими залежить від вимог даної системи до рівня безпеки, швидкодії, об’єму пам’яті тощо.

Створений в результаті роботи програмний продукт може використовуватися при розробці SaaS застосунків за допомогою фреймворків Spring та Hibernate і дозволить програмістам сконцентрувати увагу на розробці бізнес-логіки застосунку. Бібліотека легко інтегрується у вже існуючий сервіс, оскільки для початку роботи необхідно додати відповідний jar файл у проект і вказати мінімальну конфігурацію у файлі властивостей, таку як вибрану стратегію та розташування скриптів міграцій.

В подальшому бібліотека може бути покращена шляхом додавання підтримки різних ORM фреймворків, відмінних від Hibernate, та можливості використання іншого засобу для міграції баз даних. Крім того, подальше дослідження ефективності роботи даного програмного продукту може призвести до покращення швидкодії.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1. Defining Multi-Tenancy: A Systematic Mapping Study on the Academic and the Industrial Perspective / J.Kabbedijk, S. Jansen, A. Zaidman, A. Bezemer. // Journal of Systems and Software. – 2015. – №100. – С. 139–148.
2. Mell P. The NIST Definition of Cloud Computing / P. Mell, T. Grance // Computer Security / P. Mell, T. Grance. – Gaithersburg: National Institute of Standards and Technology, 2011.
3. Hurwitz J. Cloud Services for Dummies / J. Hurwitz, M. Kaufman, D. Halper. – Hoboken, New Jersey: John Wiley & Sons, Inc., 2012. – 70 с.
4. Multi-tenant SaaS database tenancy patterns [Електронний ресурс] // Microsoft Documentation. – 2019. – Режим доступу до ресурсу: https://docs.microsoft.com/en-us/azure/azure-sql/database/saas-tenancy-app-design-patterns
5. Fahad Khan M. An Approach Towards Customized Multi-Tenancy / M. Fahad Khan, M. Ahsan Ullah, Aziz-ur-Rehman. // Modern Education and Computer Science. – 2012. – №9. – С. 39–44.
6. Furda A. A practical approach for detecting multi-tenancy data interference / A. Furda, C. Fidge, A. Barros. // Elsevier. – 2018. – №163. – С. 160–173.
7. Stallings W. Cryptography and Network Security Principles and Practices / William Stallings. – Hoboken, New Jersey: Prentice Hall, 2005. – 592 с.
8. Database Scalability, Elasticity, and Autonomy in the Cloud : 16th International Conference, DASFAA 2011, 22-25 квіт. 2011, Гонконг, Китай / Springer – Л., 2011.
9. Beginning Spring / M.Caliskan, K. Sevindik, R. Johnson, J. Höller. – Birmingham: Wrox Press, 2015. – 480 с.
10. Pro Spring 5: An In-Depth Guide to the Spring Framework and Its Tools / I. Cosmina, C. Schaefer, R. Harrop, C. Ho. – California: Apress, 2017. – 865 с.

11. Bauer C. Java Persistence with Hibernate / C. Bauer, G. King, G. Gregory. – New York: Manning Publications, 2016. – 608 с.
12. Hibernate ORM 5.4.31.Final User Guide [Електронний ресурс] / [V. Mihalcea, S. Ebersole, A. Boriero та ін.] – Режим доступу до ресурсу: https://docs.jboss.org/hibernate/orm/5.4/userguide/html_single/Hibernate_User_Guide.html#naming.
13. Fisher P. Spring Persistence with Hibernate / P. Fisher, B. Murphy. – New York: Apress, 2016. – 177 с.
14. Guide to the Hibernate EntityManager [Електронний ресурс]. – 2020. – Режим доступу до ресурсу: https://www.baeldung.com/hibernate-entitymanager.
15. Multi-tenancy [Електронний ресурс] // Community Documentation – Режим доступу до ресурсу: https://docs.jboss.org/hibernate/orm/4.3/devguide/en-US/html/ch16.html.
16. Verona J. Learning DevOps: Continuously Deliver Better Software / J. Verona, M. Duffy, P. Swartout. – Birmingham: Packt, 2016. – 713 с.
17. Sadalage P. Evolutionary Database Design [Електронний ресурс] / P. Sadalage, M. Fowler. – 2016. – Режим доступу до ресурсу: https://martinfowler.com/articles/evodb.html.
18. Vasseur A. AOP Benchmark [Електронний ресурс] / Alexandre Vasseur. – 2004. – Режим доступу до ресурсу: https://web.archive.org/web/20150520175004if_/https:/docs.codehaus.org/display/AW/AOP+Benchmark.
19. Working with SQL Databases [Електронний ресурс] // Spring Boot Reference Documentation – Режим доступу до ресурсу: https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#boot-features-configure-datasource.
20. JPA Repositories [Електронний ресурс] // Part I. Reference Documentation – Режим доступу до ресурсу: https://docs.spring.io/spring-data/data-jpa/docs/1.3.4.RELEASE/reference/html/jpa.repositories.html.

21. HikariCP Benchmark [Електронний ресурс] – Режим доступу до ресурсу: https://github.com/brettwooldridge/HikariCP-benchmark.

ДОДАТОК А

```
@Configuration
@ConditionalOnProperty(prefix = "multitenancy.common.datasource", name = "url")
@ConditionalOnExpression("'${multitenancy.strategy}'.equals('database')")
@EnableJpaRepositories(
        basePackages = { "${multitenancy.common.repository.packages}" },
        entityManagerFactoryRef = "commonEntityManagerFactory",
        transactionManagerRef = "commonTransactionManager"
)
@EnableConfigurationProperties({DataSourceProperties.class, JpaProperties.class})
public class CommonPersistenceConfig {
    private final ConfigurableListableBeanFactory beanFactory;
    private final JpaProperties jpaProperties;
    private final String entityPackages;

    @Autowired
    public CommonPersistenceConfig(ConfigurableListableBeanFactory beanFactory,
                                   JpaProperties jpaProperties,
                                   @Value("${multitenancy.common.entityManager.packages}")
                                           String entityPackages) {
        this.beanFactory = beanFactory;
        this.jpaProperties = jpaProperties;
        this.entityPackages = entityPackages;
    }

    @Bean
    public LocalContainerEntityManagerFactoryBean commonEntityManagerFactory(
            @Qualifier("commonDataSource") DataSource dataSource) {
        LocalContainerEntityManagerFactoryBean em = new LocalContainerEntityManagerFactoryBean();

        em.setPersistenceUnitName("common-persistence-unit");
        em.setPackagesToScan(entityPackages);
        em.setDataSource(dataSource);

        JpaVendorAdapter vendorAdapter = new HibernateJpaVendorAdapter();
        em.setJpaVendorAdapter(vendorAdapter);


        Map<String, Object> properties = new HashMap<>(this.jpaProperties.getProperties());

        em.setJpaPropertyMap(properties);

        return em;
    }

    @Bean
    public JpaTransactionManager commonTransactionManager(
            @Qualifier("commonEntityManagerFactory") EntityManagerFactory emf) {
        JpaTransactionManager transactionManager = new JpaTransactionManager();
        transactionManager.setEntityManagerFactory(emf);
        return transactionManager;
    }
}

```

Рисунок А.1 – Реалізація класу CommonPersistenceConfig

```java
@Component("currentTenantIdentifierResolver")
@Conditional(value = SchemaOrDatabaseCondition.class)
public class CurrentTenantIdentifierResolverImpl implements CurrentTenantIdentifierResolver {

    @Override
    public String resolveCurrentTenantIdentifier() {
        String tenantId = TenantContext.getTenantId();
        if (StringUtils.isNotBlank(tenantId)) {
            log.info("CurrentTenantIdentifierResolver: " + tenantId);
            return tenantId;
        } else {
            log.info("No tenant was specified");
            return "";
        }
    }

    @Override
    public boolean validateExistingCurrentSessions() {
        return true;
    }
}
```

Рисунок А.2 – Реалізація контракту CurrentTenantIdentifierResolver

```java
@MappedSuperclass
@FilterDef(name = "tenantFilter", parameters = {@ParamDef(name = "tenantId", type = "string")})
@Filter(name = "tenantFilter", condition = "tenant_id = :tenantId")
@EntityListeners(com.knu.ynortman.multitenancy.discriminator.util.TenantListener.class)
@Slf4j
public abstract class AbstractBaseEntity implements TenantAware, Serializable {
    private static final long serialVersionUID = 1L;

    @Column(name = "tenant_id")
    private String tenantId;

    public AbstractBaseEntity(String tenantId) {
        log.info("ABSTRACT BASE ENTITY SET ID");
        this.tenantId = tenantId;
    }

}
```

Рисунок А.3 - Реалізація класу AbstractBaseEntity