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:

СТВОРЕННЯ СЕРВІСУ ДЛЯ РЕАЛІЗАЦІЇ АРХІТЕКТУРНОГО ПІДХОДУ "MULTITENANCY" З ОПТИМАЛЬНИМ РОЗПОДІЛЕННЯМ РЕСУРСІВ

Виконав студент 4-го курсу
Владислав ЯКИМЕНКО

____
(підпис)

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

____
(підпис)

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

Студент

____
(підпис)

Роботу розглянуто й допущено до захисту
на засіданні кафедри інтелектуальних програмних
систем
«25» травня 2022 р.,
протокол № 10

Завідувач кафедри
Олександр ПРОВОТАР

____
(підпис)

Київ – 2022

РЕФЕРАТ

JAVA, MULTITENANCY, POSTGRESQL, SAAS, SPRING, БАЗА ДАНИХ, МІКРОСЕРВІСИ, ХМАРНІ ТЕХНОЛОГІЇ

Об’єктом роботи є розроблення та дослідження програмного забезпечення що знаходиться безпосередньо в серці архітектурного підходу Multitenancy що в свою чергу широко використовується у SaaS моделі хмарового обслуговування клієнтів.

Метою роботи є розробка сервісу для обслуговування програмного забезпечення заточеного під згаданий вище архітектурний підхід з оптимізацією використання хмарових ресурсів.

Інструментами розробки є інтегроване середовище розробки програмного забезпечення для багатьох мов програмування, зокрема Java – IntelliJ IDEA, мова програмування Java версії 17, вільна об'єктно-реляційна система управління базами даних PostgreSQL версії 14, універсальний фреймворк з відкритим вихідним кодом для Java-платформи – Spring, а також система автоматичного збирання Java проектів Gradle.

Результати роботи: розроблено готовий до використання програмний сервіс, який займається створенням, менеджментом вже створених, сутностей орендарів під егідою багатоорендної архітектури, а також зберіганням та наданням інформації про бази даних, створені для кожного з орендарів, клієнтам даного сервісу.

В подальшому сервіс може розвиватися в бік додавання нового функціоналу, а також забезпечення найвищого рівня безпеки для зберігання різного роду секретної інформації, як-от паролі до створених баз даних орендаторів, тощо.

ЗМІСТ

РЕФЕРАТ **2**

СКОРОЧЕННЯ ТА УМОВНІ ПОЗНАЧЕННЯ **5**

ВСТУП **6**

РОЗДІЛ 1 ТЕОРЕТИЧНА ЧАСТИНА **8**

1.1 Хмарні обчислення 8

1.1.1 Як хмарні обчислення працюють 9

1.1.2 Основні типи хмарних обчислень 10

1.1.3 Типи розгортання хмари 11

1.2 Багатоорендність 12

1.2.1 Різниця між одноорендністю та багатоорендністю 13

1.2.2 Основні переваги багатоорендної архітектури 14

1.2.3 Основні складності пов’язані з даною архітектурою 15

1.2.4 Важливість багатоорендної архітектури 15

1.2.5 Варіанти реалізації багатоорендності в SaaS застосунках 16

РОЗДІЛ 2 ОПИС ВИКОРИСТАНИХ ТЕХНОЛОГІЙ **18**

2.1 Spring Framework 18

2.1.1 Spring Boot 19

2.1.2 Spring Data 22

РОЗДІЛ 3 ПРАКТИЧНА ЧАСТИНА **24**

3.1 Інструменти розробки 24

3.2 Створення унікального ідентифікатора орендаря 24

3.3 Вибір моделі реалізації багатоорендності 26

3.3.1 Спільне використання однієї бази даних і схеми через логічне розділення клієнтів 27

3.3.2 Окремі схеми бази даних для різних орендарів 27

3.3.3 Окремі бази даних для різних орендарів 28

3.4 Процес створення нового орендаря 28
3.4.1 Створення сутності орендаря 28
3.4.2 Створення баз даних відповідно до потреб орендаря 29
ВИСНОВКИ 30
ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ 31

СКОРОЧЕННЯ ТА УМОВНІ ПОЗНАЧЕННЯ

API - Application Programming Interface, програмний інтерфейс програми

AWS – Amazon Web Services, комерційна публічна хмара

SaaS – Software as a Service, програмне забезпечення як послуга

PaaS – Platform as a Service, платформа як послуга

IaaS – Infrastructure as a Service, інфраструктура як послуга

JVM – Java Virtual Machine, віртуальна машина Java

JDK - Java Development Kit, набір інструментів для розробки додатків мовою Java

JAR – Java-архів

JPA – Java Persistence API

DI – Dependency Injection, інверсія залежностей

POJO – Plain Old Java Object

ORM - Object-Relational Mapping, об'єктно-реляційне відображення

DAO - Data Access Object

SQL – Structured Query Language, мова структурованих запитів

JPQL - Java Persistence Query Language, мова запитів для JPA

LTS – Long-Term Support

ВСТУП

Оцінка сучасного стану об’єкта дослідження або розробки. Ріст популярності хмарних обчислень останні роки, без перебільшення, неймовірний – відповідно до дослідження компанії 451 Research[1]: 90% відсотків компаній використовують хмарні обчислення. За інфографікою даного дослідження можна побачити, що хмара почала набирати популярність ще у 2019 році. Більше того, експерти кажуть, що в 2019 році 60% робочих навантажень виконувалися на розміщених в хмарі сервісах. Для порівняння, у 2018 це число було 45%. На 2021 рік це число зросло аж до 94%.

Багато факторів мало вплив на таку динаміку, починаючи від великого спектру послуг, які надають хмарні провайдери, де кожен клієнт може знайти щось корисне для себе, і закінчуючи економністю та зручністю такого підходу для розміщення власних сервісів, відносно до традиційного, коли кожна компанія мала свої сервери і керувала ними самотужки.

Дуже важливою технологією, яка і дозволила хмарним обчисленням набути такої популярності є віртуалізація. Саме віртуалізація дозволяє ділити загальнодоступні ресурси між клієнтами, але при цьому залишати кожного з них ізольованими відносно один одного, що в свою чергу надає можливість якомога ефективніше використовувати апаратні ресурси, так як не залишається фізичних машин, які використовують тільки малу частину свого потенціалу, а розподіляють ці ресурси рівномірно між віртуальними машинами.

Актуальність роботи та підстави для її виконання. Через зростання популярності хмарних обчислень, набирає також попиту архітектурний шаблон “Multitenancy” так як його природа дуже добре вписується в цю парадигму використання апаратних ресурсів. Кожна компанія, яка використовує багатоорендність реалізує і підтримує даний підхід власноруч - саме тому було вирішено спроектувати та реалізувати сервіс, який допоміг би позбавити компанії

---

[1] Посилання на дослідження 451 Research (https://451research.com/images/Marketing/press_releases/Pre_Re-Invent_2018_press_release_final_11_22.pdf)

задачі кожного разу вигадувати колесо заново. Особливістю конкретно моєї реалізації буде динамічне розподілення баз даних на різних серверах-хостах відповідно до метрик, які з цих віддалених серверів можна отримати.

Окрім того, інструментом для реалізації даного сервісу було вибрано мову програмування Java в кон'юнкції із фреймворком Spring Boot, які займають лідуючі позиції в світі розробки застосунків корпоративного рівня.

Мета й завдання роботи. Метою роботи є проектування та створення сервісу, що є необхідним інструментом при використанні архітектурного шаблону "Multitenancy" в програмних застосунках рівня SaaS. Даний сервіс повинен вміти маніпулювати сутностями орендарів, а також створювати та керувати базами даних, які необхідні для орендарів. Для досягнення цієї мети було поставлено наступні цілі:

- Проаналізувати архітектурний шаблон багатоорендності, дослідити існуючі підходи його реалізації у застосунках рівня SaaS.
- Спроектувати даний сервіс, виходячи з результатів аналізу найбільш підходящого варіанту реалізації багатооренності.
- Реалізувати даний сервіс

Можливі сфери використання. Створений програмний сервіс може бути використаний для підтримки багатоорендності у застосунках рівня SaaS.

РОЗДІЛ 1 ТЕОРЕТИЧНА ЧАСТИНА

1.1 Хмарні обчислення

«Хмара», по своїй суті, є сервером, а, окрім того, програмним забезпеченням та базами даних, що працюють на ньому, для доступу до яких потрібно використати Інтернет. Ці сервери знаходяться в дата-центрах по всій земній кулі. Користуючись хмарою, компаніям і користувачам не потрібно буде самостійно керувати фізичними серверами або запускати програмні застосунки на своїх машинах.

Хмарні сервери надають клієнтам доступ до одних і тих же файлів та програм майже з будь-якого пристрою, бо збереження даних та обчислення відбувається на серверах у дата-центрах, а не на конкретному пристрої клієнта. І як результат, користувач має можливість увійти до особистого облікового запису Instagram на новому телефоні після того, як його старий зламався, і так само побачити свій акаунт на місці разом зі всіма власними фото, відео, а також історією розмов. Також це стосується і хмарних постачальників пошти, таких як Microsoft Outlook або Gmail, і, окрім того, постачальників хмарних сховищ, наприклад Google Drive, Dropbox.

Компанії, які переходять на хмарний тип розгортання своїх застосунків також суттєво економлять на ІТ та супутніх витратах: для прикладу, цим компаніям більше не буде потрібно підтримувати та оновлювати свої сервери, бо це турбота компанії, яка надає хмарні послуги. Це має найбільший вплив на невеликі компанії, що, ймовірно, не могли собі дозволити мати особисту інфраструктуру, тим не менш, мають можливість компенсувати власні потреби в інфраструктурі за рахунок зовнішніх хмарних провайдерів, що надають свої послуги за доступною ціною. Більше того, використання хмари полегшує діяльність фірм на міжнародному рівні, тому що працівники і клієнти мають можливість отримати доступ до одних і тих самих файлів і програм з майже будь-якої точки планети.

1.1.1 Як хмарні обчислення працюють

Хмарні обчислення є можливими завдяки технологіям віртуалізації. Віртуалізація – це засіб, який надає можливість створити віртуальний комп'ютер всередині справжнього фізичного, що поводиться так, ніби це фізична машина з власним обладнанням, але при цьому є повністю цифровою імітацією. Термін, який використовується для опису такого роду імітацій – віртуальна машина. При правильному використанні віртуальні машини, що знаходяться на одному фізичному комп'ютері є відокремленими логічно, тому навіть можуть не знати про існування один одного, а ресурси, що знаходяться на одній віртуальній машині не видимі для інших віртуальних машин, навіть якщо вони увімкнені.

Ця технологія робить використання фізичного обладнання, на якому віртуальні машини розміщені, більш ефективним. Коли одночасно запускається велика кількість віртуальних машин, то тоді один фізичний сервер стає безліччю віртуальних, а дата-центр, де знаходяться ці фізичні сервери стає цілою низкою дата-центрів, що можуть надавати послуги великій кількості організацій. Отже, хмарні провайдери мають можливість надавати свої послуги набагато більшому числу клієнтів одночасно, аніж вони мали змогу це робити якщо віртуалізації не існувало б, і до того ж за сильно нижчою ціною.

Навіть коли частина хмарних серверів ламається, інші сервери, загалом, повинні завжди бути доступними для підключень. Хмарні провайдери, як правило, мають резервні копії власних сервісів на декількох серверах та регіонах.

Клієнти мають змогу отримати доступ до служб, або використовуючи браузер, або через застосунок, підключившись до хмари за допомогою Інтернету, цебто встановивши з'єднання через взаємопов'язані мережі, незважаючи на те, який пристрій використовується.

1.1.2 Основні типи хмарних обчислень

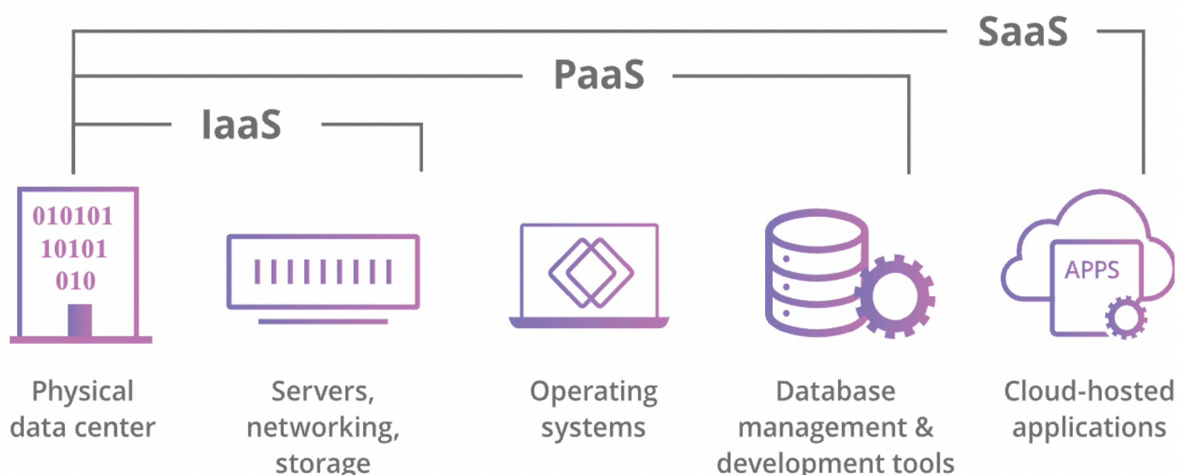

Рисунок 1.1 – Типи хмарних обчислень

Програмне забезпечення як послуга (SaaS): на відміну від того, щоб клієнти завантажували застосунок на власний пристрій, додатки рівня SaaS розташовуються на серверах хмарних провайдерів, і клієнти мають змогу використовувати їх через Інтернет. Гарним прикладом SaaS є оренда апартаментів: орендодавець доглядає за апартаментами, а той хто орендує використовує їх, наче він власник. Прикладом таких SaaS програм є Slack і Salesforce.

Платформа як послуга (PaaS): ця модель дозволяє компаніям не платити за розміщення програм; замість того, клієнти платять за необхідні їм інструменти для створення свого власного програмного забезпечення. Провайдери PaaS надають все, що необхідно для створення програмного забезпечення, включаючи інфраструктуру, засоби розробки, а також операційні системи, доступ до яких можна отримати через Інтернет. Аналогію для PaaS можна провести з орендою обладнання та інструментів, що є необхідними для будівництва апартаментів, щоб не орендувати самі апартаменти. Прикладами PaaS можна назвати AWS та Google Cloud.

Інфраструктура як послуга (IaaS): ця модель дозволяє компаніям орендувати сховища даних та сервери у провайдера хмари. Після цього вони

користуються хмарною інфраструктурою для розробки свого програмного забезпечення. IaaS — наче компанія, що тимчасово орендує ділянку землі, на якій буде розміщуватися все, що вони хотітимуть побудувати, але для цього потрібно буде надати матеріали та будівельне обладнання. Постачальниками IaaS називають OpenStack та DigitalOcean.

Раніше ці три моделі хмарних обчислень були основними, і, по суті, всі сервіси, що надавали хмарні провайдери можна було описати однією з названих вище категорій. Однак, останніми роками з'явилася четверта модель:

Функція як послуга (FaaS): FaaS, також відомий як безсерверна модель обчислення. Вона ділить програми, розміщені в хмарі, навіть на менші компоненти в порівнянні з іншими трьома, що починають використовуватися тільки тоді, коли є потрібними. Уявіть собі, якби можна було орендувати апартаменти по частинам: орендар платив би за їдальню тільки коли їв, за спальню, тільки коли спав, вітальню, під час перегляду телевізору, а тоді коли він не користувався б кімнатами, то і платити йому не потрібно було б.

FaaS все ще працює на реальних серверах, так само як і інші моделі, але називається «безсерверним», тому що компаніям, які створюють програми, не потрібно керувати жодними серверами.

1.1.3 Типи розгортання хмари

На відміну від розглянутих вище моделей хмарних обчислень, що відрізняються тим, як послуги продаються клієнта, різні типи розгортання є пов’язаними з місцезнаходженням серверів та тим хто керує ними.

Найпоширеніші типи розгортання хмар:

- **Приватна хмара:** це сервер, дата-центр або розподілена мережа, яка повністю належить одній організації.
- **Публічна хмара:** це служба якою керує зовнішній постачальник, яка може включати в себе сервери в одному, або кількох центрах обробки даних. На відміну від приватної хмари, публічні хмари використовуються кількома організаціями. Використовуючи віртуальні машини, окремі сервери можуть

використовуватися різними компаніями. Підхід, який називається «багатоорендність», називається так через те, що кілька орендарів орендують серверний простір на одному сервері.

- **Гібридна хмара:** розгортання гібридної хмари поєднує публічні та приватні хмари і навіть може включати локальні застарілі сервери. Організація може використовувати приватну хмару для одних послуг і публічну хмару для інших, або вони можуть використовувати публічну як резервну копію для своєї приватної.
- **Мультихмара:** мультихмарний тип розгортання передбачає використання кількох публічних хмар. Іншими словами, організація з мультихмарним розгортанням орендує віртуальні сервери та послуги у кількох зовнішніх постачальників — якщо продовжити аналогію, використану вище, це все одно, що орендувати кілька сусідніх земельних ділянок у різних орендодавців. Багатохмарне розгортання також може бути гібридною хмарою і навпаки.

1.2 Багатоорендність

Багатоорендність — це форма хмарної архітектури, де кілька клієнтів одного постачальника хмари використовують однакові обчислювальні ресурси. Кожен клієнт відомий як орендар. Ця форма спільного доступу стосується спільного використання програмних ресурсів, а також спільного хостингу на серверах.

Багатоорендність дозволяє кільком екземплярам даної програми працювати в спільному середовищі. Таким чином, один екземпляр програмного забезпечення працює на сервері і вміщує безліч орендарів. Орендарі інтегруються фізично, але вони розділені логічно.

Орендарі мають певну міру свободи для кастомізації спільного ресурсу, як-от контроль того, які користувачі можуть отримати доступ до ресурсів або як додаток виглядає та відчуває себе. Однак вони не можуть налаштувати код.

Ось типовий приклад. Зверніть увагу на ваш місцевий банк. Багато клієнтів вносять гроші в банк, але кожен має свій особистий рахунок, де інші клієнти не

можуть торкатися їхніх активів, хоча всі зберігають свої кошти в одному банку. Колеги-клієнти банку не спілкуються один з одним, не мають доступу до ресурсів один одного і навіть не знають про існування один одного — один банк, багато клієнтів.

Це модель загальнодоступної хмари — одна інфраструктура, багато серверів, причому кожен сервер має ексклюзивні права на використання та доступ до відповідних даних та інших хмарних ресурсів.

1.2.1 Різниця між одноорендністю та багатоорендністю

Архітектура з одним орендарем, як не дивно, є протилежністю архітектури з багатьма орендарями. Термін «один орендар» — це словосполучення, яке безпосередньо пояснює природу і сенс цієї архітектури. Єдина оренда означає, що програмне забезпечення та допоміжна інфраструктура обслуговують лише одного клієнта. Кожен клієнт має свою окрему, незалежну базу даних або програмний екземпляр, і ніхто не ділиться.

Модель з одним орендарем має свої переваги. Одна оренда пропонує, для початку, більш безпечне середовище, оскільки ресурси кожного орендаря повністю відокремлені від ресурсів та інформації інших орендарів. Крім того, клієнт має повну свободу налаштування та контролю функціональності. Нарешті, клієнти-орендарі користуються більшою надійністю та стабільністю, оскільки ресурси завжди доступні та в достатку.

Однак, у єдиної оренди є і недоліки. Відсутність спільного використання ресурсів також означає відсутність розподілу витрат, тому одна оренда, як правило, дорожча. Крім того, одиноким орендарям немає з ким поділитися регулярними обов'язками по технічному обслуговуванню та налаштуванню, тому потрібно більше роботи.

Таким чином, єдина оренда забезпечує більшу конфіденційність і доступність ресурсів, ніж багатоорендне користування. Однак, останнє споживає менше часу, ресурсів і грошей клієнта.

1.2.2 Основні переваги багатоорендної архітектури

Ми коротко торкалися деяких переваг багатоорендної архітектури, але давайте детальніше.

- Це рентабельно. Подібно до того, як ділити поїздку з іншими людьми для нас дешевше, дешевше ділитися і хмарними ресурсами. Клієнти в кінцевому підсумку платять тільки за те, що їм потрібно, і нічого більше. Праця та персонал, підключення нових орендарів, технічне обслуговування та розробка, а також оновлення — усе це виконується хмарним хостом, при цьому лише частина витрат передається і розподіляється між орендарями.
- Легке масштабування. Завдяки економічній ефективності, це буде також означати, що клієнти зможуть додавати або видаляти ресурси за потреби. Ця гнучкість ідеально підходить для організацій, які швидко, але непередбачувано розвиваються.
- Це досить безпечно і забезпечує достатній рівень конфіденційності. Хоча це правда, що одна оренда є більш безпечною, багатоорендне використання все ще добре виявляє загрози та зберігає ресурси орендарів окремо один від одного.
- Більш ефективне використання ресурсів. З боку хоста, багатоорендний підхід дозволяє краще використовувати інфраструктуру. Має сенс відкрити доступ до сервера багатьом клієнтам, а не обмежувати його тільки одним.
- Замовник не потребує обслуговування. Клієнти не тільки не повинні оплачувати витрати на моніторинг та адміністрування, але й не повинні брати участь у роботі. Хост обробляє такі завдання як оновлення програмного та фізичного забезпечення, технічне обслуговування та інші пов’язані завдання. Розглянемо цю модель як оренду квартири — орендодавець займається ремонтом, орендар — платить за оренду.

1.2.3 Основні складності пов'язані з даною архітектурою

Звичайно, жодна модель хмарного хостингу не є ідеальною або без ризиків. Ось деякі недоліки багатоорендної архітектури:

- Це можлива загроза безпеці. Якщо ви вирішите використовувати модель багатоорендного користування, існує ймовірність, хоч і незначна, що ваші дані можуть стати доступними для третіх сторін. Цей злом може статися випадково, через збій у роботі системи або як програмна помилка, яка розкриває дані, або навмисно, завдяки хакеру, який використовує слабкі місця в архітектурі. Цей ризик збільшується або зменшується відносно рівня заходів безпеки, які надає хост.
- Можливі проблеми з часом відповіді. Мешканці квартир ризикують, що галасливі сусіди зіпсують їх тишу. У середовищі хмарного хостингу клієнти, які надмірно споживають ресурси ЦП, є галасливим віртуальним сусідом. Таке споживання потужності ЦП уповільнює час обробки для всіх.
- Хост керує часом простою. Подібно до того, як багатоквартирні будинки не мають права голосу, коли начальник домоуправління вирішує перефарбувати будівлю або попрацювати з сантехнікою, хмарні клієнти не диктують заплановані простої.
- Це складніше. Що стосується хоста, то багатоорендність є більш складною, ніж одноорендий підхід.
- Програми можуть бути менш гнучкими. Спільні програми, як правило, менш гнучкі, ніж програми, надані в інших архітектурах для клієнта.

1.2.4 Важливість багатоорендної архітектури

Багатоорендність є важливою концепцією, оскільки вона допомагає нам отримати максимальну користь від загальнодоступного хмарного середовища. Хмарне середовище, у свою чергу, містить дані кожного, клієнтів, постачальників, партнерів та всесвітні ресурси. Таким чином, багатоорендність пропонує якісний і постійний доступ до хмари для всіх за доступною ціною. Це чудовий спосіб

вирівняти ігрові умови, щоб малий бізнес міг краще конкурувати з монолітною мегакорпорацією.

Коли хост хмарних технологій представляє нові функціональні можливості та інновації, то ці покращення дістаються більшій аудиторії. У багатьох випадках саме потреби кількох клієнтів багатоорендного застосунку спонукають хоста до розробки нової функціональності. Тим не менш, врешті-решт, кожен користувач, в багатоорендній архітектурі, в кінцевому підсумку отримає новий функціонал також, що не може не бути позитивним бонусом для клієнта, який навіть не робив запит на нові функціональні можливості. Одноорендні моделі не працюють таким чином, оскільки будь-які вдосконалення розробляються тільки для даного клієнта.

1.2.5 Варіанти реалізації багатоорендності в SaaS застосунках

Модель реалізації багатоорендності вплине на основні аспекти вашого SaaS застосунку. Ви повинні будувати сервіс на основі потреб вашого бізнес-домену, а не довільних припущень.

Ось варіанти реалізації багатоорендності, які варто розглянути:

- Окремі бази даних. Кожен орендар отримує нову базу даних. Однакові програми масштабуються або по вертикалі (додавання ресурсів на вузол) або горизонтально (додавання додаткових вузлів). Однак бази даних в тих самих групах ресурсів поділяються на еластичні пули. Постачальники можуть переміщувати бази даних орендарів між цими пулами, щоб оптимізувати управління ресурсами.
- Єдина база даних для кількох орендарів. База даних із багатьма клієнтами складається з кількох стовпців ідентифікаторів орендарів, тоді як сховище та обчислювальні ресурси спільно використовують усі користувачі. Це призводить до зниження витрат на одного орендаря. З іншого боку, робоче навантаження одного орендаря може вплинути на те, як послуга працює для інших.
- Розділені бази даних для орендарів. Цей підхід називається шардінг і дозволяє зберігати дані клієнта в кількох базах даних. У міру зростання робочого

навантаження ви зможете розділити щільно заповнені сегменти на кілька менш завантажених вузлів (або об’єднати їх так само легко). Крім того, розділені бази даних можна помістити в еластичні пули для подальшого покращення оперативного управління та масштабованості.

- Гібридні розділені бази даних для орендарів. Ви можете переміщати клієнта або цілі групи між виділеними та спільними базами даних. Цей підхід найкраще працює, коли у вас є кілька груп орендарів, які можна ідентифікувати, з різними потребами в ресурсах (рівні безкоштовної пробної версії та преміум-клієнти).

При розробці багатоорендних додатків слід враховувати кількість орендарів та їх ізоляцію.

РОЗДІЛ 2 ОПИС ВИКОРИСТАНИХ ТЕХНОЛОГІЙ

2.1 Spring Framework

Spring дозволяє легко створювати програми корпоративного рівня на мові програмування Java. Він надає все, що вам потрібно для використання Java в корпоративному середовищі, також він має підтримку Groovy та Kotlin як альтернатив Java на JVM. Цей фреймворк є гнучким для створення багатьох типів архітектур залежно від потреб застосунку що розробляється. Починаючи з версії 5.1, Spring вимагає рівень JDK не нижче 8 (Java SE 8+) і також має готову підтримку для 11 LTS версії JDK.

Spring підтримує широкий спектр сценаріїв застосування. Часто великі корпоративні програми існують досить тривалий час і мають працювати на JDK, та сервері для розгортання програм, цикл оновлення якого не контролюється розробником. Інші можуть працювати як один jar-архів із вмонтованим сервлет-контейнером, ймовірно, у хмарному середовищі. Ще інші можуть бути окремими програмами (наприклад, пакетні або інтеграційні робочі навантаження), яким не потрібен сервер.

Spring має відкритий вихідним код. Він має активну та велику спільноту, що забезпечує безперервний зворотній зв’язок на основі різноманітних випадків використання його в реальному світі. Це допомогло фреймворку успішно розвиватися протягом дуже тривалого часу.

Термін «Spring» має різні значення для різних контекстів. Його можна використовувати для посилання на сам проект Spring Framework, з якого все починалося. З часом інші проекти Spring були створені поверх Spring Framework. Найчастіше, коли говорять «Spring», мають на увазі всю сім’ю проектів.

Даний фреймворк розділений на модулі. Програмні застосунки можуть вибирати які модулі їм потрібні. В основі — модулі контейнера ядра, включаючи механізм ін’єкції залежностей, а також модель конфігурації. Більше того, Spring Framework забезпечує базову підтримку для різних архітектур додатків, включаючи обмін повідомленнями, транзакційність даних та модуль для зберігання

даних, а також веб. Spring включає в себе веб-фреймворк Spring MVC, який працює на основі сервлетів і, одночасно з тим, реактивний веб-фреймворк Spring WebFlux.

2.1.1 Spring Boot

Розробники Spring прийняли рішення поставляти фреймворк відразу з набором корисних утиліт, які автоматизують процедуру налаштування та прискорюють процес створення та розгортання додатків написаних з його допомогою у хмарному середовищі, під загальною назвою Spring Boot.

Spring Boot є корисним проектом, головною ціллю якого є спрощення розробки додатків з використанням Spring. Він надає найпростіший спосіб створити web-застосунок, потребуючи від програміста мінімум зусиль з його налаштування та написання шаблонного коду.

Spring Boot має широкий функціонал, але найбільш корисними його функціями є: управління залежностями, автоматична конфігурація та вмонтовані контейнери сервлетів.

2.1.1.1 Легкість управління залежностями

Для прискорення процесу управління залежностями, Spring Boot неявно пакує необхідні сторонні залежності для кожного типу програми на основі Spring і поставляє їх розробнику у вигляді так званих starter-пакетів: spring-boot-starter-data-jpa, spring-boot-starter-web і т.д.

Starter-пакети є набором зручних дескрипторів залежностей, які можна включити у свою програму. Це дасть можливість мати універсальне рішення для всіх, пов'язаних зі Spring фреймворком технологій, позбавляючи розробника зайвого часу пошуку прикладів коду та завантаження необхідних дескрипторів залежностей (приклад таких дескрипторів та стартових пакетів буде показано нижче).

Наприклад, якщо ви хочете мати можливість використовувати у своєму додатку Spring Data JPA для з’єднання з базою даних, то потрібно просто додати у свій проект залежність на spring-boot-starter-data-jpa і цього буде достатньо, щоб

почати його використовувати у своєму застосунку (вам не доведеться шукати сумісні драйвери баз даних та відповідну версію Hibernate ORM).

Або якщо ви бажаєте створити web-додаток з використанням Spring MVC, то просто додайте залежність spring-boot-starter-web, яка каскадом додасть в проект усі бібліотеки, які необхідні для розробки такого роду додатків, наприклад spring-webmvc, validation-api, Apache Tomcat, jackson-json.

Іншими словами, Spring Boot збирає всі залежності, які зазвичай використовуються у даного типу додатках та визначає їх в одному місці, що дозволяє розробникам використовувати цілий набір залежностей одним махом, визначивши залежність стартера в дескрипторі збирання проекту, замість того, щоб винаходити колесо щоразу, коли вони створюють нову програму.

Звідси, використовуючи build.gradle або pom.xml файли в Spring Boot додатках ви матимете набагато менше рядків залежностей і конфігурацій, ніж при використанні його в звичайних Spring-додатках.

2.1.1.2 Автоматична конфігурація

Другою чудовою функцією, яку має Spring Boot є автоматична конфігурація програми. Після додавання нового starter-пакета, Spring Boot буде намагатися автоматично налаштувати Spring-додаток на основі доданих вами, або автоматично завантажених jar-архівів, які з’явилися в структурі вашого java-застосунку після збирання проекту.

Уявимо, що ви додали стартер для web функціоналу, після цього Spring Boot автоматично спробує зконфігурувати біни необхідні для такого типу додатку, як-от DispatcherServlet, MessageSource, ResourceHandlers, тощо. Якщо використовується spring-boot-starter-jdbc, то Spring Boot автоматично зформує біни для роботи з базою даних, а саме: DataSource, TransactionManager, EntityManagerFactory та візьме до уваги інформацію, яка необхідна для підключення до бази даних із спеціального файлу конфігурації - application.yml, де окрім цих, описані також і інші налаштування для перевизначення тих, які Spring Boot використовує за замовчуванням.

Якщо ви додали даний стартер до дескриптору збірки проекту, але не збираєтеся використовувати бази даних, і відповідно не надали ніякої інформації, яка є необхідною для підключення в ручному режимі, то тоді Spring Boot займеться налаштуванням бази даних в оперативній пам'яті без додавання будь-якої додаткової конфігурації з вашого боку (за наявності H2 або HSQL бібліотек).

2.1.1.3 Вбудована підтримка сервера програм — контейнера сервлетів

Кожен web-додаток, який використовує Spring Boot включає в себе вмонтований web-сервер.

Розробникам більше не прийдеться турбуватися за налаштування та розгортання програми в контейнері сервлетів. Тепер програма може запускатися самтужки, як самостійний jar-архів з використанням вмонтованого сервера.

Якщо є потреба для використання окремого HTTP-серверу, то для цьої цілі достатньо вимкнути налаштування залежностей за замовчуванням. Spring Boot постачає різні starter-пакети для різних HTTP-серверів.

Створення самостійних web-застосунків з вмонтованими серверами є не тільки дуже зручно для розробки, але є і допустимим рішенням для програм корпоративного рівня існуючих у світі мікросервісів. Можливість швидко упаковати весь сервіс (наприклад, автентифікацію користувача) в автономному артефакті, що повністю розгортається, який також надає API — робить установку і розгортання програми значно простіше.

2.1.2 Spring Data

Місія Spring Data полягає в тому, щоб забезпечити знайому та послідовну модель програмування на основі Spring для доступу до даних, зберігаючи при цьому особливі риси основного сховища даних.

Це спрощує використання технологій доступу до даних, реляційних і нереляційних баз даних, фреймворків для "map-reduce" і хмарних сервісів даних. Це загальний проект, який містить багато підпроектів, специфічних для даної бази даних. Проекти розробляються разом із багатьма компаніями та розробниками, які стоять за цими технологіями.

Так як ми плануємо взяти за основу реляційну модель для баз даних, то нас цікавить саме модуль 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 - це специфікація Java, що описує систему управління збереженням POJO об'єктів у таблиці реляційних баз даних у зручному вигляді. Сама Java не містить реалізації JPA, однак є доволі велика кількість реалізацій даної специфікації від великої кількості компаній, як відкритих так і закритих. Це не єдиний спосіб збереження POJO об'єктів у базі даних (ORM систем), але один із найпопулярніших у Java світі.

2.1.2.1 Spring Data JPA

Реалізація рівня доступу до даних програми була громіздкою протягом досить довгого часу. Для виконання простих запитів, а також для розбиття на сторінки та аудиту потрібно написати занадто багато шаблонного коду. Spring Data JPA має на меті значно покращити реалізацію рівнів доступу до даних, зменшуючи зусилля до фактично необхідного обсягу. Як розробник, ви пишете інтерфейси свого репозиторію, включаючи спеціальні методи пошуку, і Spring автоматично забезпечить реалізацію.

Spring Data JPA містить концепцію репозиторіїв JPA, набір інтерфейсів, які визначають методи запитів. Repository та Entity Bean представляють рівень DAO у програмі. Більше не потрібно писати нативні запити. Іноді нам потрібно писати запити або частину запитів, але це запити JPQL, а не рідні запити до бази даних.

РОЗДІЛ 3 ПРАКТИЧНА ЧАСТИНА

3.1 Інструменти розробки

Для реалізації проекту була вибрана 17 LTS-версія мови програмування Java, що на даний момент є останньою довгостроковою версією цієї суворо типізованої об'єктно-орієнтованої мови програмування загального призначення. У якості системи автоматичного збирання java-проектів був вибраний Gradle. Окрім того, використовувався Spring Framework версії 5.1, точніше його надбудова Spring Boot версії 2.6.3 з наступним набором стартерів: spring-boot-starter-web, spring-boot-starter-data-jpa, spring-boot-starter-validation. Варто згадати і цікаві бібліотеки, які були застосовані для полегшення розробки: MapStruct - генератор коду, який значно спрощує реалізацію відображень між POJO об'єктами на основі підходу "конвенція понад конфігурацією"; Lombok – інший генератор, який позбавляє нас необхідності написання іншого роду шаблонного коду такого як-от геттери та сеттери для полів класів, або для автоматичної реалізації такого породжуючого дизайн-шаблону як Будівельник, тощо. Для контролю версій структури та міграцій бази даних використовувався Liquibase. Також була використана вільна об'єктно-реляційна система управління базами даних PostgreSQL версії 14 як для зберігання даних користувачів всередині сервісу так і як головний діалект SQL на серверах баз даних. Розробка проводилася в інтегрованому середовищі розробки програмного забезпечення для багатьох мов програмування, зокрема Java – IntelliJ IDEA.

3.2 Створення унікального ідентифікатора орендаря

Кожен орендар повинен мати свій унікальний ідентифікатор (далі – tenantId), який може бути використаний для широкого спектру функціональних можливостей: наприклад, залежно від підписки, яку оформив клієнт, він матиме право на користування різною кількостю SaaS продуктів. Кожен з продуктів буде мати свою власну базу даних, і tenantId буде використовуватися як частина назви такої бази даних. Також цей ідентифікатор добре підходить як частина DNS-адреси

до застосунку клієнта; це зручно, коли користувач має декілька акаунтів і за допомогою цієї адреси можна точно розуміти який акаунт зараз використовується.

API мого сервісу буде мати два різні ендпоїнти для створення сутності орендаря: один генерує tenantId самостійно, а інший надає можливість користувачу задати цей ідентифікатор вручну.

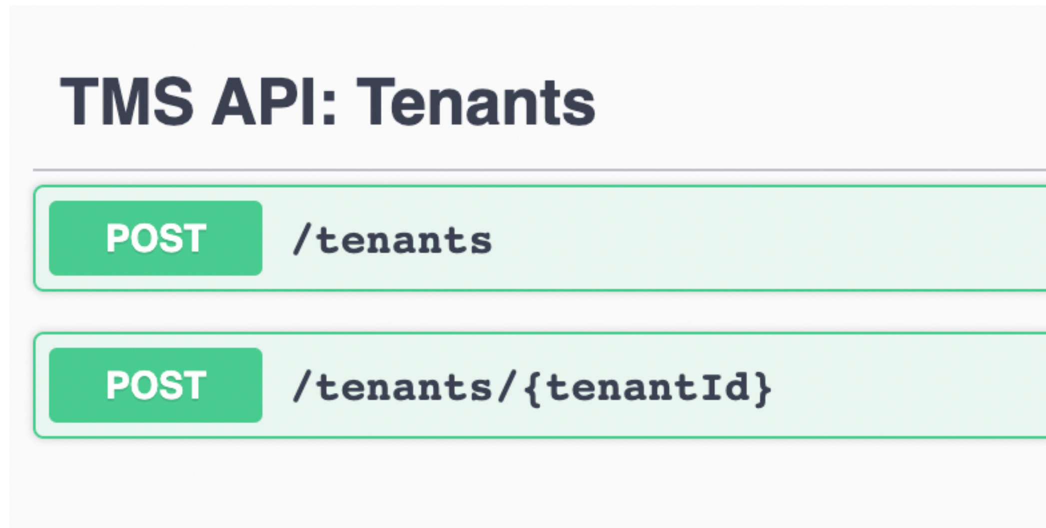

Рисунок 3.1 – Ендпоїнти для створення сутності орендаря

Спочатку розглянемо випадок, коли сервіс генерує ідентифікатор орендаря самостійно. Для цього використовується клас java.util.UUID: після виклику статичного методу randomUUID() видаляються всі дефіси і беруться тільки перші 10 символів, а інші значення відкидаються. UUID - це стандарт ідентифікації, який використовується у розробці програмного забезпечення, стандартизований Open Software Foundation (OSF), що є частиною DCE - середовищ розподілених обчислень. Головним призначенням UUID є дозволити розподіленим системам унікально ідентифікувати інформацію без єдиного центру координації.

У випадку, коли користувач використовує інший ендпоїнт – він зобов'язується надати десятизначну послідовність символів; вона може бути довільною.

Після того як tenantId, тим чи іншим способом, отриманий, сервіс додатково верифікує його унікальність за допомогою механізму обмежень в PostgreSQL, а

саме “UNIQUE Constraint”, який ми наклали на колонку що містить ідентифікатор орендаря в Liquibase скрипті при створені таблиці орендарів.

```yaml
databaseChangeLog:
  - changeSet:
      id: create-tenants-table
      author: Vladyslav Yakymenko
      changes:
        - createTable:
            tableName: tenants
            columns:
              - column:
                  name: id
                  type: SERIAL
                  autoIncrement: true
                  constraints:
                    primaryKey: true
              - column:
                  name: tenant_id
                  type: VARCHAR(10)
                  constraints:
                    unique: true
                    nullable: false
                    uniqueConstraintName: uq_tenants_tenantId
```

Рисунок 3.2 - Частина Liquibase скрипта для створення таблиці орендарів з підкресленим обмеженням для унікальності значень в колонці tenantId

3.3 Вибір моделі реалізації багатоорендності

Проблема вибору моделі реалізації цього архітектурного шаблону була ключовою під час проектування даного сервісу, тому на різниці найпоширеніших підходів для його реалізації я зупинюся детальніше. Отже, існують три основні варіанти реалізації багатоорендності у SaaS застосунках:

- Спільне використання таблиць бази даних за допомогою логічного розділення клієнтів
- Окремі схеми однієї бази даних для різних орендарів
- Окремі бази даних для різних орендарів

Розберемо кожен з цих варіантів відповідно до наших потреб.

3.3.1 Спільне використання однієї бази даних і схеми через логічне розділення клієнтів

При такому підході всі користувачі будуть спільно використовувати одні і ті самі таблиці, а розділення даних різних орендарів буде відбуватися не фізично, а логічно за допомогою поля, або полів, які містять унікальний ідентифікатор певного клієнта – tenantId, наприклад.

Такий підхід є найекономніший стосовно хмарних ресурсів, але це, напевно, його єдиний плюс. Разом з цією моделлю реалізації багатоорендності в комплекті йде ціла купа проблема, починаючи з відомої проблеми шумного сусіда, коли при надмірній активності одного орендаря починають страждати також і інші через високе завантаження сервера на якому знаходиться база даних і інші клієнти можуть мати проблеми з обробкою своїх власних запитів, і закінчуючи проблемами з конфіденційністю даних, адже користувач, отримавший доступ до бази даних, буде мати інформацію про абсолютно всіх користувачів в системі. До того ж такий підхід вимагає більшого об'єму пам'яті та ресурсів, що в свою чергу створює верхню границю для масштабованості сервісу і навіть більше того база даних стане занадто громіздкою ще до моменту досягнення цієї межі. Отже, дана модель не є найкращим варіантом, якщо у нас немає жорсткого обмеження по хмарним ресурсам і ми можемо дозволити собі мати більшу кількість баз даних.

3.3.2 Окремі схеми бази даних для різних орендарів

Як і в минулому варіанті інформація клієнтів все ще знаходиться в одній базі даних, але при цьому за орендарями закріплена своя власна схема. Цей підхід дещо надійніший за попередній через те, що клієнтам можна обмежувати доступ до певних схем. Але при цьому всьому все ще залишається проблема "шумного сусіда", бо якщо "шумний сусід" навіть буде обмежений своєю схемою – його надмірна кількість запитів буде збільшувати навантаження на всю базу даних в цілому.

3.3.3 Окремі бази даних для різних орендарів

Даний підхід є найнадійнішим серед цих трьох так як кожен орендар буде мати свою, повністю ізольовану, базу даних, паролі до якої буде мати тільки він. За потреби можливо взагалі розширити функціонал сервісу, щоб користувачі могли створювати бази даних на їх власному сервері-хості баз даних, де можна налаштувати певні стратегії шифрування, тощо. Цей підхід, очевидно, є найзатратнішим відносно вартості, але при цьому його надійність дозволяє використовувати його в таких бізнес-доменах як банкінг та медицина. Окрім цього, він є найгнучкішим для персоналізації під конкретні вимоги користувача, що в свою чергу дозволяє змінювати схему, структури таблиць, а також легко створювати резервні копії баз даних користувачів за потреби, без шкоди для інших орендарів.

Зважаючи на всі ці переваги, озвучені вище, мною було прийняте рішення використовувати саме його.

3.4 Процес створення нового орендаря

Цей процес можна розбити на два логічних етапи: 1) створення сутності орендаря з визначенням необхідної кількості та типів баз даних що потрібно створити для даного клієнта; 2) безпосередньо створення баз даних, ролей, які будуть мати права її використовувати та інші необхідні пререквізити.

3.4.1 Створення сутності орендаря

Створення сутності орендаря – це по факту процес заповнення необхідних таблиць всередині самого сервісу, без створення ресурсів орендаря на відповідних серверах-хостах баз даних. Першою фазою створення орендаря є визначення його унікального ідентифікатор – tenantId, як було описано раніше в цій секції. Після чого збирається та аналізується кількість та тип баз даних, які буде необхідно створити. Для цього використовується файл конфігурацій у форматі .yml (який може бути екстерналізований і розгортатися на сервері незалежно від сервісу для більшої гнучкості) що в кон’юнкції з анотацією @ConfigurationProperties дозволить

перевести цей файл в POJO сутність, з якої потім можна дістати потрібну нам інформацію у вигляді кількості та типів баз даних.

3.4.2 Створення баз даних відповідно до потреб орендаря

Після того як ми отримали повний список необхідних баз даних потрібно вирішити на якому з серверів їх розмістити. Для цього кроку ми будемо використовувати 2 величини: кількість баз даних і кількість запитів до цього сервера за останню годину. Відповідно сервер з найменшими величинами і буде нашим кандидатом. Для отримання кількості баз даних ми можемо виконати безпосередньо SQL запит до цього RDS'у:

```
SELECT count(*) FROM pg_database;
```

Рисунок 3.3 – SQL-запит для отримання кількості баз даних,
що зараз є на даному RDS

Для отримання метрик відносно кількості запитів на сервер за останню годину потрібно скористатися спеціальним API, що надає AWS – MetricQuery. З цього API ми можемо отримати поле "db.load.avg" – що буде зберігати в собі масштабоване уявлення кількості активних сеансів для ядра бази даних, а також "db.sampledload.avg" – що зберігає необроблену кількість активних сеансів для ядра бази даних.

Після визначення найбільш підходящого сервера строюється користувач бази даних з іменем у вигляді: "tenantId_databaseName_randomCharSequence" – він і буде мастер-користувачем для неї. Потім створюється база даних з іменем у такому ж вигляді як і користувача, у них буде відрізнятися тільки випадкова частина назви. Останнім кроком буде оновлення вже існуючих записів у внутрішній базі даних сервісу, що стосуються орендаря для якого ці бази даних і створювалися, щоб клієнти, у яких є на це права, могли отримати інформацію для підключення до власних баз даних.

ВИСНОВКИ

В даній роботі було проаналізовано архітектурний шаблон “Multitenancy”, його важливість у світі зі стрімким зростанням популярності хмарних обчислень, окрім цього, було досліджено три основні моделі реалізації багатоорендності в застосунках рівня SaaS. При виборі якогось одного з них потрібно, перш за все, опиратися на потреби вашого бізнес-домену, а не на довільні припущення.

В результаті роботи був створений програмний продукт, який може бути використаний при розробці застосунків рівня SaaS та дозволяє керувати сутностями орендарів, створювати бази даних необхідні для продуктів доступ до яких є у клієнта, надавати інформацію про ці бази даних авторизованим користувачам, в тому числі і продуктам-застосункам. А також автоматично розподіляти нові бази даних на різних серверах, відповідно до завантаження наявних серверів-хостів баз даних.

В подальшому до сервісу може бути доданий функціонал створення і інших ресурсів необхідних для коректної роботи продуктів наявних у орендаря, як-от топіки в меседж-брокерах для конкретного клієнта, а також доопрацювання сервісу в бік гарантій максимальної безпеки даних користувачів що його використовують.

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. – No100. – С. 139–148.
2. Multi-tenancy [Електронний ресурс] // Community Documentation – Режим доступу до ресурсу: https://docs.jboss.org/hibernate/orm/4.3/devguide/en-US/html/ch16.html.
3. Furda A. A practical approach for detecting multi-tenancy data interference / A. Furda, C. Fidge, A. Barros. // Elsevier. – 2018. – No163. – С. 160–173.
4. 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.
5. Beginning Spring / M.Caliskan, K. Sevindik, R. Johnson, J. Höller. – Birmingham: Wrox Press, 2015. – 480 с.
6. Spring Microservices in Action, Second Edition / John Carnell, Illary Huaylupo Sánchez. // Manning, - 2021, - 448
7. DI (Dependency Injection) [Електроний ресурс]. – Режим доступу: https://www.martinfowler.com/articles/injection.html
8. Effective Java / J. Bloch. // Addison-Wesley Professional. - 2017. – 416
9. Clean Code: A Handbook of Agile Software Craftsmanship / Robert C. Martin, Dean Wampler. // Pearson, 2008, - 464