Міністерство освіти і науки України
Тернопільський національний технічний університет імені Івана Пулюя

Факультет комп'ютерно-інформаційних систем і програмної інженерії
(повна назва факультету)
Кафедра комп'ютерних наук
(повна назва кафедри)

КВАЛІФІКАЦІЙНА РОБОТА

на здобуття освітнього ступеня

бакалавр
(назва освітнього ступеня)

на тему: Створення веб-сервісу для тестування хмарних додатків

Виконав: студент IV курсу, групи СНс-41
спеціальності 122 Комп'ютерні науки
(шифр і назва спеціальності)

(підпис) Миськів С. Р.
(прізвище та ініціали)

Керівник (підпис) Матійчук Л.П.
(прізвище та ініціали)

Нормоконтроль (підпис) Марценко С.В.
(прізвище та ініціали)

Завідувач кафедри (підпис) Боднарчук І.О.
(прізвище та ініціали)

Рецензент (підпис) Жаровський Р.О.
(прізвище та ініціали)

Тернопіль
2023

Міністерство освіти і науки України
Тернопільський національний технічний університет імені Івана Пулюя

Факультет комп'ютерно-інформаційних систем і програмної інженерії
(повна назва факультету)

Кафедра комп'ютерних наук
(повна назва кафедри)

ЗАТВЕРДЖУЮ
Завідувач кафедри
_____ Боднарчук І.О.
(підпис) (прізвище та ініціали)
«__» _____ 2023р.

ЗАВДАННЯ
НА КВАЛІФІКАЦІЙНУ РОБОТУ

на здобуття освітнього ступеня Бакалавр
(назва освітнього ступеня)

за спеціальністю 122 Комп'ютерні науки
(шифр і назва спеціальності)

Студенту Миськів Сергій Русланович
(прізвище, ім'я, по батькові)

1. Тема роботи Створення веб-сервісу для тестування хмарних додатків

Керівник роботи Матійчук Любомир Павлович, к.е.н., доцент кафедри КН
(прізвище, ім'я, по батькові, науковий ступінь, вчене звання)

Затверджені наказом ректора від «*07*» лютого 2023 року № 4/7-133.

2. Термін подання студентом завершеної роботи 12 червня 2023р.

3. Вихідні дані до роботи Наукові публікації, електронні ресурси, підручники , посібники з тематики дослідження

4. Зміст роботи (перелік питань, які потрібно розробити)
Вступ. Розділ 1. Аналіз предметної області підтримка прцесів для тестування хмарних додатків та проектування системи для тестування хмарних додатків. 1.1 Опис предметної області 1.2 Аналіз існуючих аналогів, що реалізують функції предметної області 1.3 Розроблення архітектури програмної системи та проектування структури бази даних 1.4 Висновки до першого розділу. Розділ 2. Програмна реалізація сервісу, тестування та дослідна експлуатація хмарних додатків. 2.1 Використовувані технології та стандарти та програмні інтерфейси сервісу. 2.2 Архітектура окремих методів тестування та процес створення компонентів сервісу. 2.3 Тестування та розгортання сервісу для тестування хмарних додатків. 2.4 Висновки до другого розділу. Розділ 3. Безпека життєдіяльності, основи охорони праці. Висновки. Перелік джерел. Додатки.

5. Перелік графічного матеріалу (з точним зазначенням обов'язкових креслень, слайдів)
1. Тема. 2. Схема cloud технологій. 3. Мета роботи. 4. Схема cloud тестування. 5-7. Огляд аналогів. 8. Діаграма варіантів використання. 9. Архітектура сервісу для тестування мікросервісних хмарних додатків. 10. Діаграми послідовності тестування. 11. Схема бази даних з вихідною інформацією. 12. Архітектура Ruby on Rails. 13. Процес створення компонентів сервісу тестування. 14. Процес створення компонентів сервісу тестування. 15. Процедура розгортання додатку на платформі Heroku. 16-17. Особливості реалізації web-сервісу та інструкція користувача. 18. Висновки. 19. Дякую за увагу.

6. Консультанти розділів роботи

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		завдання видав	завдання прийняв
Безпека життєдіяльності, основи охорони праці	Гурик О. Я., доцент кафедри МТ		

7. Дата видачі завдання 23 січня 2023 р.

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів роботи	Термін виконання етапів роботи	Примітка
1.	Ознайомлення з завданням до кваліфікаційної роботи	08.05.2023	*Виконано*
2.	Підбір джерел про створення веб-сервісу для тестування хмарних додатків	09.05.2023-11.05.2023	*Виконано*
3.	Опрацювання джерел по темі кваліфікаційної роботи	12.05.2023-15.05.2023	*Виконано*
4.	Виконання дослідження щодо створення веб-сервісу для тестування хмарних додатків	16.05.2023-22.05.2023	*Виконано*
5.	Оформлення розділу «Аналіз предметної області підтримка прцесів для тестування хмарних додатків та проектування системи для тестування хмарних додатків.»	23.05.2023-25.05.2023	*Виконано*
6.	Оформлення розділу «Програмна реалізація сервісу, тестування та дослідна експлуатація хмарних додатків.»	26.05.2023-28.05.2023	*Виконано*
7.	Виконання завдання до підрозділу «Безпека життєдіяльності»	29.05.2023-30.05.2023	*Виконано*
8.	Виконання завдання до підрозділу «Основи охорони праці»	31.05.2023-01.06.2023	*Виконано*
9.	Оформлення кваліфікаційної роботи	02.06.2023-06.06.2023	*Виконано*
10.	Нормоконтроль	07.06.2023-08.06.2023	*Виконано*
11.	Перевірка на плагіат	09.06.2023	*Виконано*
12.	Попередній захист кваліфікаційної роботи	12.06.2023	*Виконано*
13.	Захист кваліфікаційної роботи	14.06.2023	

Студент _____ Миськів С. Р.
(підпис) (прізвище та ініціали)

Керівник роботи _____ Матійчук Л.П.
(підпис) (прізвище та ініціали)

АНОТАЦІЯ

Створення веб-сервісу для тестування хмарних додатків // Кваліфікаційна робота освітнього рівня «Бакалавр» // Миськів Сергій Русланович // Тернопільський національний технічний університет імені Івана Пулюя, факультет комп'ютерно-інформаційних систем і програмної інженерії, кафедра комп'ютерних наук, група СНс-41 // Тернопіль, 2023 // С. –64, рис. – 38, табл. – 9, бібліогр. – 30.

Ключові слова: мобільний додаток, архітектура програмної системи, веб-сервіс, MySQL, база даних, хмарні сховища.

У першому розділі здійснено опис предметної області, напрями діяльності. Проведено аналіз відомих програмних систем. Здійснено аналіз вимог до програмної системи. Розроблено архітектуру web-сервісу, що дозволить краще зрозуміти функції основних його частин. Створено та описано структурну схему, основними компонентами якої є: рівень клієнта, рівень бізнес-логіки та рівень даних. Описано функціональну структуру системи та її основних елементів – модулів обробки даних. Визначено основні елементи бази даних та встановлено зв'язки між ними. Спроектовано структуру бази даних.

У другому розділі здійснено опис предметної області, напрями діяльності. Визначено склад функцій, що входять до бізнес-процесу на основі яких розроблено схему управління бізнес-процесом. Проведено аналіз відомих програмних систем. Здійснено аналіз вимог до програмної системи. Розроблено архітектуру web-сервісу, що дозволить краще зрозуміти функції основних його частин. Створено та описано структурну схему, основними компонентами якої є: рівень клієнта, рівень бізнес-логіки та рівень даних. Описано функціональну структуру системи та її основних елементів – модулів обробки даних. Визначено основні елементи бази даних та встановлено зв'язки між ними. Спроектовано структуру бази даних.

ANNOTATION

Web Service Development for Cloud Apps Testing // Qualification work of the educational level "Bachelor" // Serhiy Myskiv // Ivan Pulyuy Ternopil National Technical University, Faculty of Computer Information Systems and Software Engineering, Department of Computer Sciences, group SNs-41 // Ternopil, 2023 // C.– 64, fig. –38 , tab. – 9, bibliography –30.

Keywords: mobile application, software system architecture, web service, MySQL, database, cloud storage.

In the first section, a description of the subject area, directions of activity was carried out. The analysis of known software systems was carried out. An analysis of the requirements for the software system was carried out. The architecture of the web service has been developed, which will allow a better understanding of the functions of its main parts. A structural diagram has been created and described, the main components of which are: client level, business logic level, and data level. The functional structure of the system and its main elements - data processing modules - are described. The structure of the database is designed.

In the second chapter, a description of the subject area, directions of activity was carried out. The composition of the functions included in the business process was determined, on the basis of which the business process management scheme was developed. The analysis of known software systems was carried out. An analysis of the requirements for the software system was carried out. The architecture of the web service has been developed, which will allow a better understanding of the functions of its main parts. A structural diagram has been created and described, the main components of which are: client level, business logic level, and data level. The functional structure of the system and its main elements - data processing modules - are described. The main elements of the database are defined and the connections between them are established. The structure of the database is designed.

ЗМІСТ

ВСТУП

Актуальність теми. В рамках мікросервісного підходу, серверна частина програми розбивається на окремі, ізольовані компоненти - мікросервіси, що забезпечують прозорий веб-доступ до своїх функціональних можливостей і реалізують певну роль в бізнес-логіці програми. Мікросервіси зберігають своє положення в окремих незалежних базах даних і взаємодіють з іншими мікросервісамі за допомогою відкритого веб-протоколу (наприклад, відповідно до стилю REST [5]). Для функціонування такого хмарного додатка необхідно забезпечити спільну роботу (оркестрації) множини мікросервісів. Мікросервісний підхід має ряд істотних переваг в порівнянніз монолітним підходом: сервіси мають чітку межу відповідальності,легко масштабуються можуть бути написані на різних мовах програмування і розроблені різними командами [10].

У зв’язку з високим ступенем масштабованості і слабо зв’язаності хмарних додатків, в даний час мікросервісний підхід до проектування архітектури ПЗкористується великою популярністю. Прикладами використання даного підходу є рішення, пропоновані такими компаніями як Amazon,eBay, Netfix і ін. [13].

Автоматизоване тестування грає дуже важливу роль в сучасному процесі розробки програмного забезпечення. Воно дозволяє завжди підтримувати працездатність продукту, вносити зміни і проводити рефакторинг.

Мікросервісний підхід вносить свої складності в питання тестування, так як потрібно перевіряти коректність взаємодії мікросервісів в різних умовах роботи. Для повного тестування мікросервісів в необхідно розгортати складне тестуючи середовище на спеціальних серверах, що вимагає великих людських і матеріальних витрат.

Актуальність даної теми обумовлена тим, що багатьом компаніям складно і дорого розгортати на своїх серверах систему для тестування своїх мікросервісних додатків, в прагненні знизити витрати вони звертають уваги на

засоби хмарного тестування. Хмарні додатки можуть надати зручний сервіс для якісної перевірки додатків користувачів за невелику плату.

Мета і задачі дослідження. Метою кваліфікаційної роботи є реалізація сервісу для тестування мікросервісних хмарних додатків. Для досягнення цієї мети потрібно вирішити наступні завдання:

1) вивчення та аналіз відомих рішень і методологій в області хмарного тестування;

2) визначення вимог до хмарного сервісу для тестуванням мікросервісних додатків;

3) розробка архітектури хмарного сервісу для тестування мікросервісних додатків, і на основі неї опис деталей реалізації окремих модулів розроблюваного рішення;

4) реалізація хмарного сервісу для тестування мікросервісних додатків;

5) тестування розробленого додатка.

Практичне значення одержаних результатів. Здійснено опис процедур тестування та їхніх результатів, описані тест-вимоги до програмного забезпечення, а також виявлені дефекти. Розкрито питання встановлення та налаштування програмного забезпечення на хмарній платформі, а також вказані вимоги, дотримання яких необхідно для користування сервісом,описана інструкція користувача для роботи із системою.

РОЗДІЛ 1. АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ ПІДТРИМКА ПРОЦЕСІВ ДЛЯ ТЕСТУВАННЯ ХМАРНИХ ДОДАТКІВ ТА ПРОЕКТУВАННЯ СИСТЕМИ ДЛЯ ТЕСТУВАННЯ ХМАРНИХ ДОДАТКІВ

1.1. Опис предметної області

Хмарні обчислення (англ. cloud computing) — це модель забезпечення повсюдного та зручного мережевого доступу на вимогу до загального пулу конфігуруємих обчислювальних ресурсів (наприклад, мереж передачі даних, серверів, пристроїв зберігання даних, додатків і сервісів - як разом, так і окремо) , які можуть бути оперативно надані і звільнені з мінімальними експлуатаційними витратами і/або зверненнями до провайдера. На рисунку 1.1 представлено загальну схему організації cloud технології.

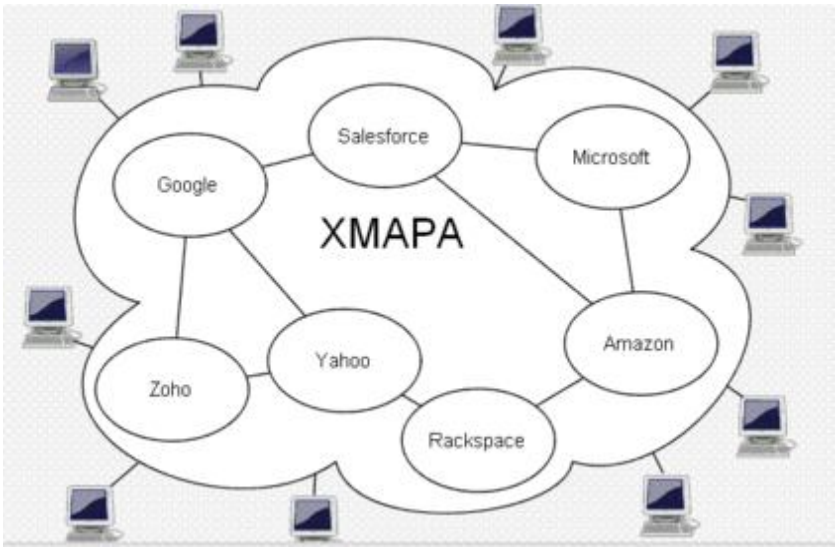

Рисунок 1.1 – Схема cloud технологій

Основні переваги:

- не потрібні великі обчислювальні потужності ПК - по суті будь-який смартфон, планшет і т.д., при відкритті вікна браузера отримує величезний потенціал.
- відмовостійкість;

- певний рівень безпеки;
- висока швидкість обробки даних;
- економія на покупці софта - всі необхідні програми вже є в сервісі, де будуть працювати додатки;
- Ваш власний вінчестер не наповнюється - всі дані зберігаються в мережі..

Є й ряд недоліків:

- хмарна послуга надається завжди якоюсь компанією, відповідно, збереження даних користувача залежить від цієї компанії;
- поява хмарних монополістів;
- необхідність завжди бути в мережі для роботи;
- небезпека хакерських атак на сервер (при зберіганні даних на комп'ютері ви в будь-який час можете відключитися від мережі і очистити систему за допомогою антивірусу);
- можлива подальша монетизація ресурсу - цілком можливо, що компанії надалі вирішить брати за послуги плату з користувачів.

З точки зору постачальника, завдяки об’єднанню ресурсів і непостійному характеру споживання з боку споживачів, хмарні обчислення дозволяють економити на масштабах, використовуючи менші апаратні ресурси, ніж потрібні були б при виділених апаратних потужностях для кожного споживача, а за рахунок автоматизації процедур модифікації виділення ресурсів істотно знижуються витрати на абонентське обслуговування.

З точки зору споживача, ці характеристики дозволяють отримати послуги з високим рівнем доступності (англ. High availability) і низькими ризиками непрацездатності, забезпечити швидке масштабування обчислювальної системи завдяки еластичності без необхідності створення, обслуговування і модернізації власної апаратної інфраструктури.

Зручність і універсальність доступу забезпечується широкою доступністю послуг і підтримкою різного класу термінальних пристроїв (персональних комп'ютерів, мобільних телефонів, інтернет-планшетів).

Національним інститутом стандартів і технологій США (ANSI) зафіксовані такі обов'язкові характеристики хмарних обчислень:

- Самообслуговування по вимозі (self service on demand), споживач самостійно визначає і змінює обчислювальні потреби, такі як серверний час, швидкість доступу та обробки даних, обсяг збережених даних без взаємодії з представником постачальника послуг;
- Універсальний доступ по мережі, послуги доступні споживачам по мережі передачі даних незалежно від використовуваного термінального пристрою;
- Об'єднання ресурсів (resource pooling), постачальник послуг об'єднує ресурси для обслуговування великої кількості споживачів в єдиний пул для динамічного перерозподілу потужностей між споживачами в умовах постійної зміни попиту на потужності; при цьому споживачі контролюють тільки основні параметри послуги (наприклад, обсяг даних, швидкість доступу), але фактичний розподіл ресурсів, що надаються споживачеві, здійснює постачальник (в деяких випадках споживачі все-таки можуть управляти деякими фізичними параметрами перерозподілу, наприклад, вказувати бажаний центр обробки даних з міркувань географічної близькості);
- Еластичність, послуги можуть бути надані, розширені, звужені в будь-який момент часу, без додаткових витрат на взаємодію з постачальником, як правило, в автоматичному режимі;
- Облік споживання, постачальник послуг автоматично обчислює спожиті ресурси на певному рівні абстракції (наприклад, обсяг збережених даних, пропускна спроможність, кількість користувачів, кількість транзакцій), і на основі цих даних оцінює обсяг наданих споживачам послуг.

Приватна хмара (private cloud) – інфраструктура, призначена для використання однією організацією, що включає кілька споживачів (наприклад, підрозділів однієї організації), можливо також клієнтами і підрядниками даної організації. Приватна хмара може перебувати у власності, управлінні та експлуатації як самої організації, так і третьої сторони (або будь-якої їх

комбінації), і воно може фізично існувати як всередині, так і поза юрисдикцією власника.

Публічна хмара (public cloud) — інфраструктура, призначена для вільного використання широкою публікою. Публічна хмара може перебувати у власності, управлінні та експлуатації комерційних, наукових та урядових організацій (або будь-якої їх комбінації). Публічна хмара фізично існує в юрисдикції власника — постачальника послуг.

Гібридна хмара (hybrid cloud) — це комбінація з двох або більше різних хмарних інфраструктур (приватних, публічних або суспільних), що залишаються унікальними об'єктами, але пов'язаних між собою стандартизованими або приватними технологіями передачі даних і додатків (наприклад, короткочасне використання ресурсів публічних хмар для балансування навантаження між хмарами).

Суспільна хмара (англ. community cloud) — вид інфраструктури, призначений для використання конкретною спільнотою споживачів з організацій, що мають спільні завдання (наприклад, місії, вимоги безпеки, політики, та відповідності різним вимогам). Громадська хмара може перебувати в кооперативній (спільній) власності, управлінні та експлуатації однієї або більше з організацій співтовариства або третьої сторони (або будь-якої їх комбінації), і вона може фізично існувати як всередині, так і поза юрисдикцією власника.

Хмарне тестування (рисунок 1.2) використовує хмарну інфраструктуру для тестування програмного забезпечення[1]. Організації, що переслідують тестування в цілому і навантаження, тестування продуктивності і моніторинг виробництва послуг, зокрема, викликають кілька проблем, таких як обмежений бюджет тестування, дотримання строків, високі витрати на випробування, велика кількість тестів, і мало або немає повторного використання тестів і географічну класифікацію користувачів які додають проблеми.

Крім того забезпечення надання послуг високої якості і уникаючи простоїв вимагає перевірки у своїй обробці даних, поза дата-центром, або

обидва випадки. Хмарне тестування, це вирішення всіх цих проблем. Ефективне необмежене зберігання, швидка наявність інфраструктури з масштабованістю, гнучкістю та доступністю розподіленого середовища тестування дозволяє скоротити час тестування великих додатків і привести до економічно ефективних рішень.

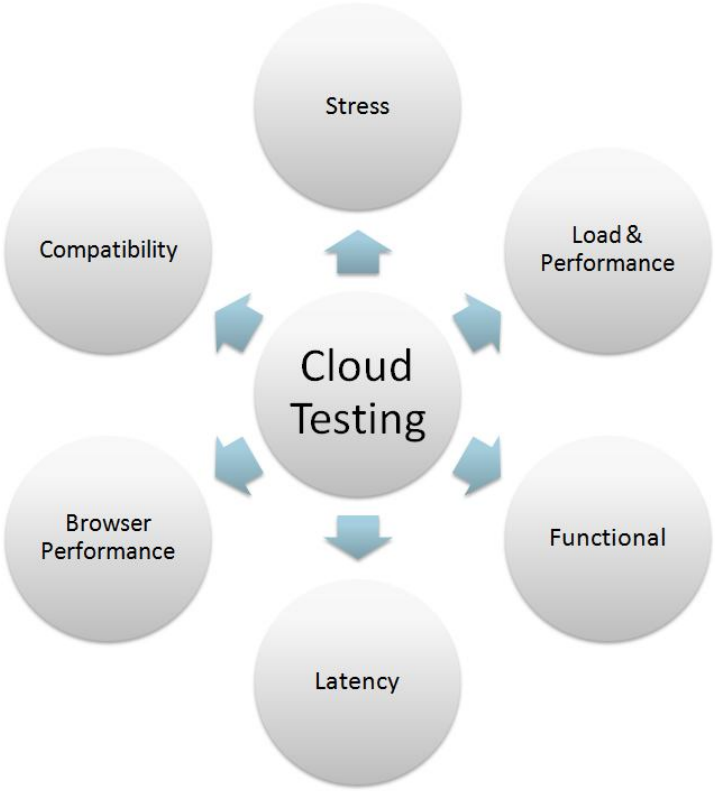

Рисунок 1.2 – Схема cloud тестування

Потреба хмарного тестування. Традиційні підходи, щоб перевірити програмне забезпечення бере на себе високу вартість для імітації активності користувача у різних географічних розташуваннях[2]. Тестування міжмережевих екранів і балансування навантаження включає в себе витрати на обладнання, програмне забезпечення і його зміст[3]. У разі застосування, де темпи зростання числа користувачів є непередбачувані або є відмінності в середовищі розгортання в залежності від вимог клієнта, хмарне тестування є більш ефективним.

1.2. Аналіз існуючих аналогів, що реалізують функції предметної області

Застосування хмарних технологій для тестування додатків розвивається, починаючи з кінця 2000-х років. Термін «Testing as a Service» (TaaS)був вперше запропонований компанією Tieto в 2009 році [15]. TaaS - це модель використання зовнішніх ресурсів (аутсорсингу), при якій завдання тестування додатків вирішуються за допомогою залучення сторонньої компанії [10]. Хмарне тестування - це різновид TaaS, при якій тестування додатків реалізується на основі хмарних обчислювальних ресурсів [14]. Застосування хмарного тестування замість класичних методів тестування додатків володіє такими перевагами як висока масштабованість тестового середовища та зниження витрат на тестування.

Дискусія щодо визначення поняття та сфери застосування хмарного тестування представлена в статті [6]. Зокрема, в статті порушені такі теми, як:

1) додатки, які підходять для хмарного тестування;

2) критерії якості тестування в хмарі;

3) розподіл потужностей для тестують додатків в багатокористувацькому середовищі;

4) онлайнові тестуючі рішення для корпоративних додатків.

Хмарне тестування має свої ризики і витрати і підходить не для всіх користувачів і типів програмного забезпечення. У статті [6] наводяться наступні критерії, яким повинні задовольняти додатки,щоб вони підходили для хмарного тестування:

1) тестові випадки повинні бути незалежні один від одного;

2) додаток має розгортатися в одній зі стандартних ОС;

3) програмний інтерфейс програми повинен підходити для автоматизованого тестування.

Ґрунтуючись на цьому, найбільш підходящими випадками для тестування є наступні:

1) юніт-тестування і регресивне тестування;

2) велика кількість автоматизованих тестів;

3) тестування навантаження.

Авторами статті [2] виділяються три випадки використання хмарного тестування:

1) TaaS для розробників - вид тестування, коли розробники пишуть автоматизовані тести і використовують їх для безперервної інтеграції;

2) TaaS для кінцевих користувачів - вид тестування, яке необхідно звичайним користувачам, сюди входять емулятори телефонів для перевірки працездатності додатків, сервіси аналізу комп'ютерна предмет відповідності вимогам до системи додатки і т.д .;

3) TaaS для сертифікаційних центрів - вид тестування, який використовується для сертифікації додатків, наприклад, для публікації вAppStore. Цей вид тестування аналізує додаток на наявність дефектів і для кожного знайденого дефекту надає факти. На основі кількості дефектів виставляється рейтинг тестування. Найчастіше всього таке тестування зводиться до тестування продуктивності і надійності.

У статті [8] автори описують такі переваги хмарного тестування:

- швидкість розгортання;

- економія коштів на утримання машин для тестування;

- висока продуктивність хмарного тестування;

- можливість перевірки специфічних характеристик веб-додатків таких, як масштабованість та надлишковість.

Всі ці переваги дозволяють знизити витрати на тестування на10-20% і підвищити його якість.

Апаратне забезпечення хмарних систем включає велику кількість процесорів, жорстких дисків, пам'яті та іншого обладнання. Різне обладнання призводить до відмов, тому виникає питання тестування відновлення додатків після збоїв. Варіант вирішення цієї проблеми запропоновано в статті [7], у вигляді платформ FATE (FailureTesting Service) (рисунок 1.3) і DESTINI

(Declarative Testing Specifications) (рисунок 1.4). FATE з певною періодичністю імітує відмову обладнання, тим самим тестуючисистему на автоматичне відновлення. DESTINI створений для зручного опису процедури а тематичного відновлення після збоїв.

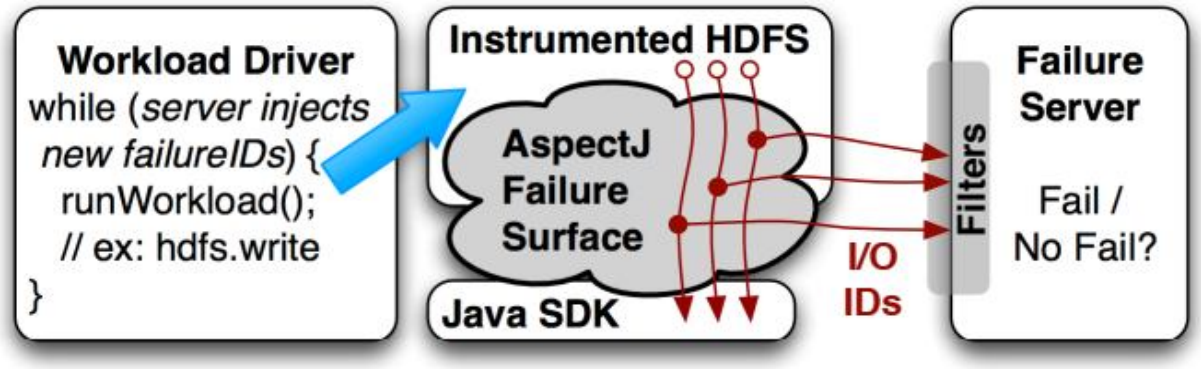

Рисунок 1.3 – Архітектура FATE (FailureTesting Service)

У статтях [7, 8] була запропонована методологія тестування мікросервісних додатків. Вона базується на трьох характеристиках якості мікросервісних програм: стабільність системи, час відгуку і безпеку. Виходячи з цього, пропонується наступна послідовність тестування.

1. Відповідно до особливості мови проведення unit тестів, для окремих компонентів мікросервісних додатків для перевірки коректності роботи окремих модулів.

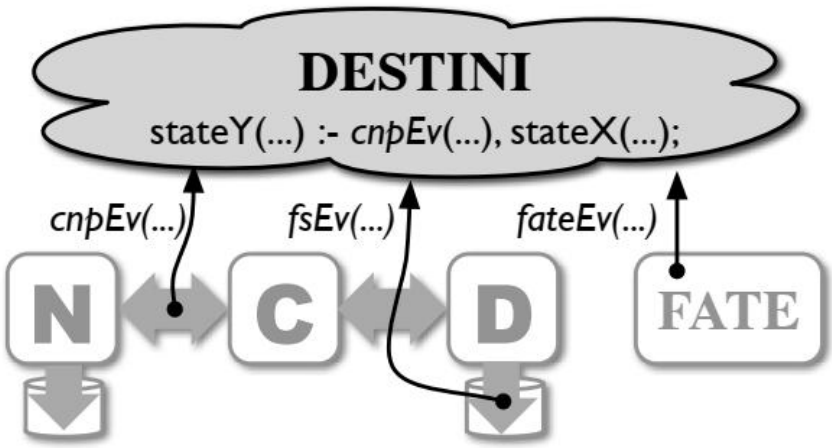

Рисунок 1.4 – Архітектура DESTINI (Declarative Testing Specifications)

2. Після завершення першого етапу тестування мікросервіс упаковується в контейнер і відбувається тестування інтерфейсу сервісу, тобто перевірка правильності виконання REST запитів до сервісу.

3. Якщо само тестування відбулося успішно, то мікросервісне розгортаються в загальному тестовому середовищі, де проводиться функціональне інтеграційне тестування, навантажувальне інтеграційне тестування і тестування безпеки.

4. Якщо всі тести пройшли вдало, то додаток розгортається на робочих серверах. Для тестування безперервної інтеграції випадковим чином, згідно зі сценарієм, виникають відмови устаткування, з якими система повинна справлятися самостійно.

Одним із запропонованих засобів для тестування розподілених і паралельних систем є D-Cloud (рисунок 1.5), запропонований в статті [1]. Ця система дозволяє створювати різні конфігурації віртуальних машин, зрізним об’ємом пам’яті, кількістю процесорів та інше. Так само для тестування відмов, вона емулює збої в роботі дисків, пам’яті, мережі та іншого, як частина тестового сценарію.

Іншим підходом до тестування хмарних додатків є Cloud [3] - хмарний сервіс для тестування, що використовує для своєї роботи ресурси Amazon EC2, що дозволяє добре розпаралелювати етапи тестування. У статті [3] показана ефективність даної методики для регресивних тестів.

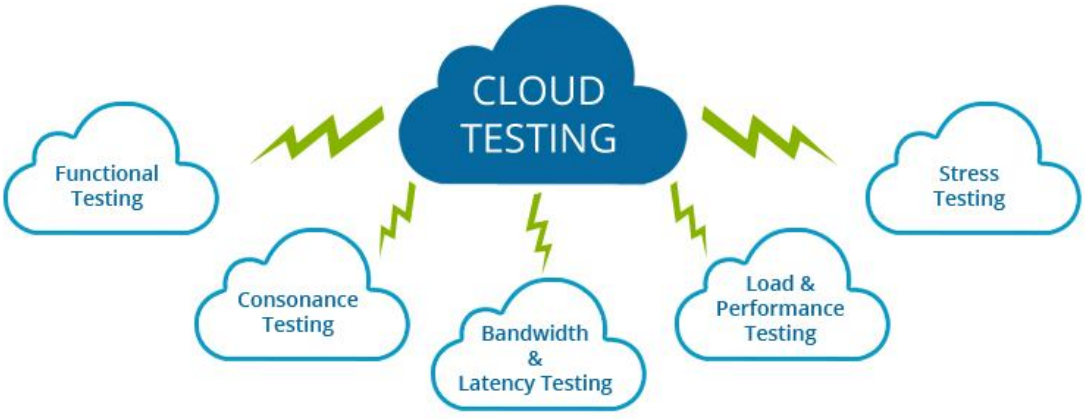

Рисунок 1.5 – Інструмент D-Cloud

Окремо необхідно розглянути існуючі комерційні пропозиції для організації хмарного тестування. Компанією IBM представлена платформа «Integrated Development and Test Environment forCloud» (рисунок 1.6), яка дозволяє інтегрувати засоби тестування в хмарну інфраструктуру [6], що включає наступні сервіси:

1) IBM Testing Services for Cloud - сервіс, що надає віртуалізацію додатків і засоби тестування продуктивності;

2) IBM Rational Load Testing - сервіс, що дозволяє виробляти інтеграційне тестування продуктивності додатку;

3) IBM Defect Analysis Starter (DAS) - сервіс, що дозволяє оцінити слабкі місця в додатку, в плані продуктивності і безпеки;

4) IBM Defect Analysis Starter for Environment - сервіс, який розширює DAS і дозволяє проводити оцінку середовища хмарних додатків.

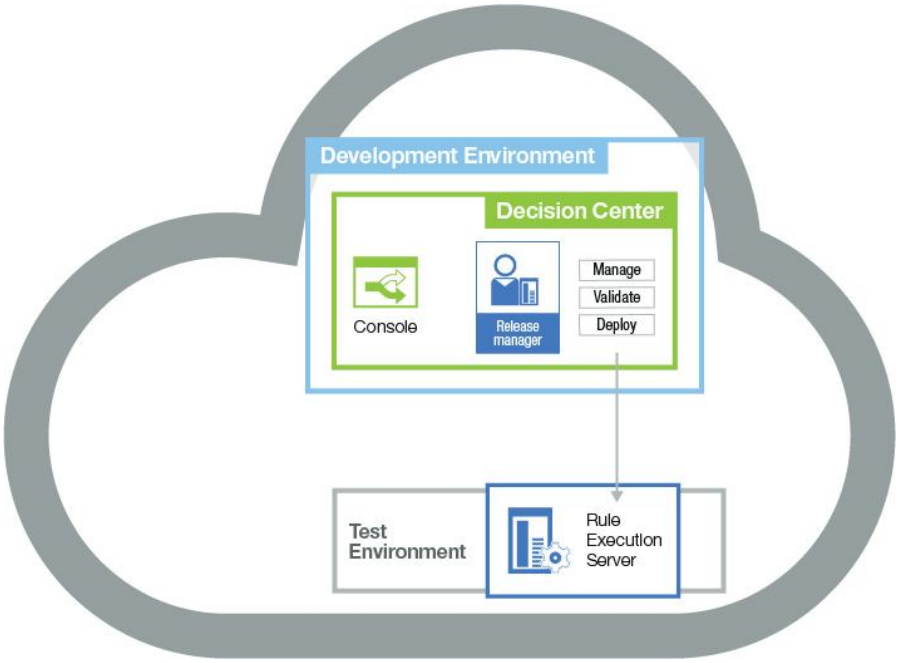

Рисунок 1.6 – Integrated Development and Test Environment forCloud

Також, існує ряд рішень в області хмарного тестування відкомпаній Hewlett-Packard [8], SOASTA [1], які надають можливості регресивного, навантажувального тестування та тестування безпеки хмарних додатків.

Окремо можна відзначити систему Load Impact (рисунок 1.7)[9] - хмарний сервіс, який надає можливості навантажувального тестування веб-сайтів. Він дозволяє симулювати до 10 тисяч користувачів одночасно. Page Analyzer tool дозволяє емулювати різні види браузерів і оцінювати різні метрики такі, як швидкість завантаження, час відгуку, час в черги в них.

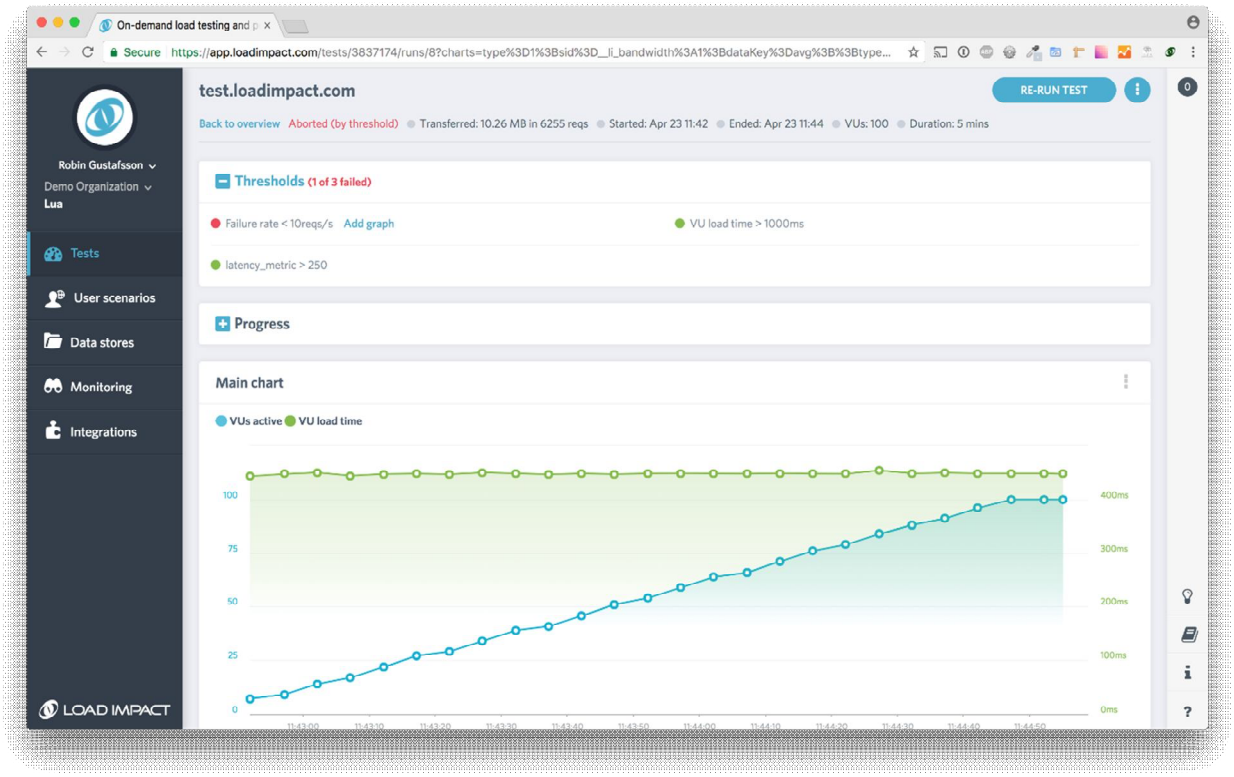

Рисунок 1.7 – Load Impact

Цікавим рішенням для тестування роботи додатків є New Relic (рисунок 1.8) [12]. New Relic - це інструмент для розробників, який здійснює моніторинг робочих програм та надає докладні дані про їх продуктивності і надійності. Таким чином, він дозволяє швидше виявляти і діагностувати проблеми, що виникають з продуктивністю, а також надає в ваше розпорядження дані, необхідні для вирішення цих проблем.

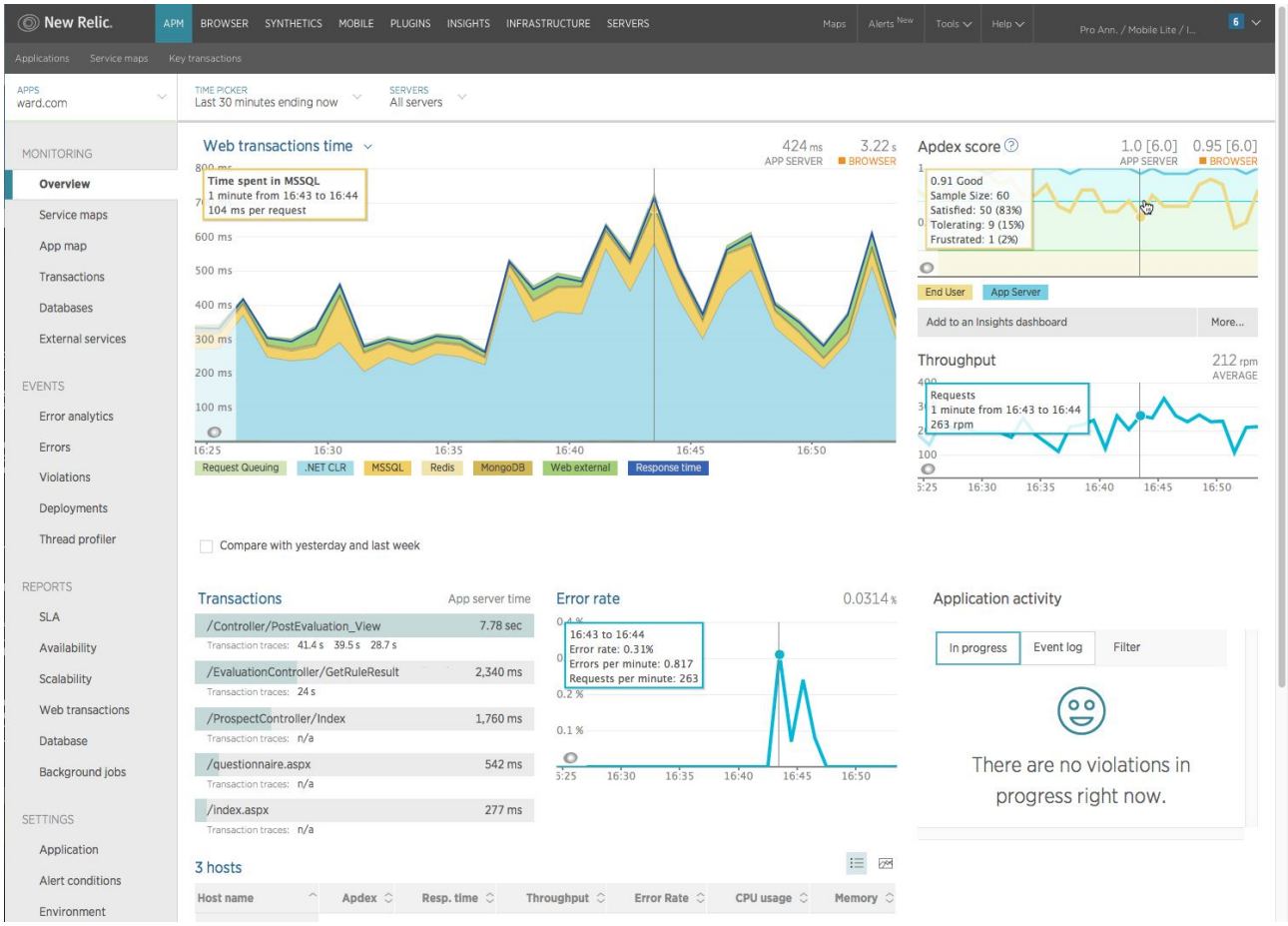

Рисунок 1.8 – Інструмент New Relic

Інструмент New Relic відстежує час завантаження і пропускну здатність веб-транзакцій як через сервер, так і через браузери користувачів. Він дозволяє відстежити використання бази даних, аналізує повільні запити і веб-запити, забезпечує моніторинг безперебійної роботи і відображає відповідні попередження, відстежує виключення в додатках, а також підтримує багато інших функцій [2].

Хмарний сервіс для тестування мікросервісних додатків дозволить автоматизувати і вивести в хмару наступні види тестування:

- компонентне тестування класів, що входять в мікросервіс на основі тестування вихідного коду;

- тестування REST інтерфейсу сервісу, на основі передбачуваних запитів.

Надалі передбачається розвиток системи і додавання нових методів тестування.

Кожен окремий метод тестування буде реалізований у вигляді незалежного сервісу тестування. Таким чином, система буде підтримувати інтеграцію існуючих хмарних методів тестування, а також розширення функціоналу за рахунок можливості інтеграції нових методів тестування.

Система представляється у вигляді веб-додатку, що дозволяє тестувати мікросервісні додатки користувачів в автоматичному режимі. Вона дозволяє налаштовувати, створювати методи тестування і пропонувати їх користувачам, відповідно до зазначених ними атрибутами додатка.

Визначимо основних акторів, що взаємодіють з системою. Тестувальник - це користувач системи, який використовує сервіс тестування для тестування будь-якого хмарного мікросервісного додатку.

Розробник методів тестування - це користувач системи, який створює і додає в сервіс тестування нові методи тестування. Він може створити свій метод тестування у вигляді окремого сервісу,приймає параметри тестованої програми (адреси кінцевих точок, репозиторії вихідного коду та ін.), а на вихід віддає результат пройдених тестів.

Мікросервісний додаток - це тестована система. Діаграма варіантів використання Web-сервісу для тестування хмарних додатків приведена на рисунку 1.9. З сервісом взаємодіють 3 актора: тестувальник, розробник методів тестування і мікросервісний додаток.

Розробник методів тестування створює методи тестування і пов’язує їх з атрибутами додатків.

Тестувальник може:

- створити проект тестування - тестувальник створює проект, який буде містити всі цілі тестування;

- створити мету тестування - тестувальник задає мету тестування, де вказує назву, опис мети, і атрибути додатка; метоюможе бути тестування як окремих компонентів мікросервісног ододатку, так і деяких аспектів функціональності (тестування навантаження на жорсткі диски, навантаження на мережу і т.д.);

- вибрати і налаштувати методи тестування - тестувальник вибирає із запропонованих методів тестування ті, які йому потрібні, і вказує необхідні вхідні дані для них.

- виконати мету тестування - тестувальник запускає мету тестування, або призначає її виконання на будь-який час, це може бути, як виконання послідовності тестів, так і режим постійного моніторингу та тестування ефектів у програмному забезпеченні, в залежності відзазначених методів тестування в цілі тестування;

- отримувати звіт про тестування - тестувальник отримує від системи звіт про результати роботи мети тестування.

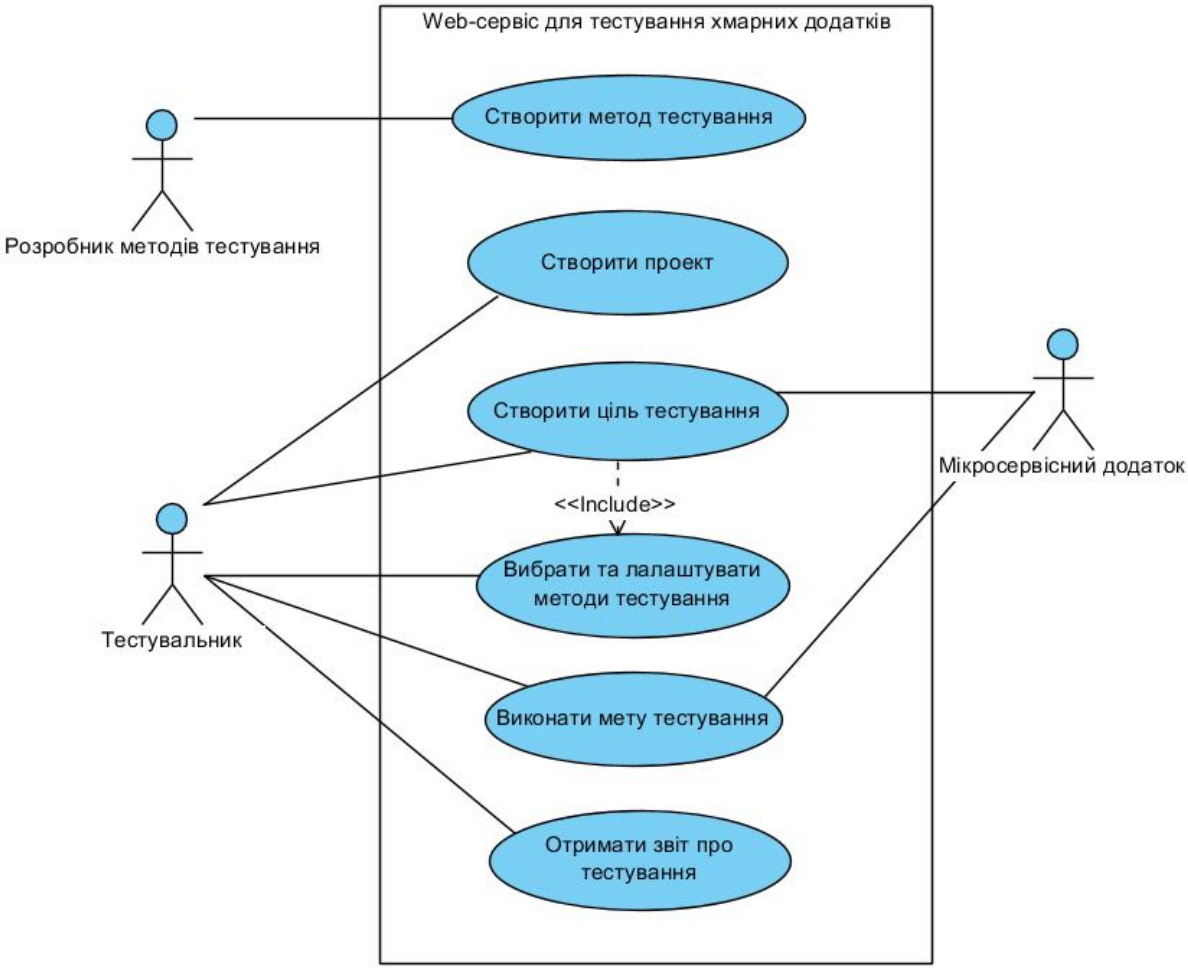

Рисунок 1.9 – Діаграма варіантів використання

Щоб мати більш повне уявлення про роботу сервісу, розглянемо

докладніше основні сценарії використання.

1.3. Розроблення архітектури програмної системи та проектування структури бази даних

Архітектура пропонованого рішення зображена на рисунку 1.10. Додаток пропонується як хмарний веб-сервіс з мікросервісной архітектурою. Він складається з основного хмарного сервісу для тестування мікросервісних додатків, який містить 3 компонента, і підключаючих методів тестування, що реалізуються у вигляді окремих мікросервісів. Розглянемо архітектуру детальніше.

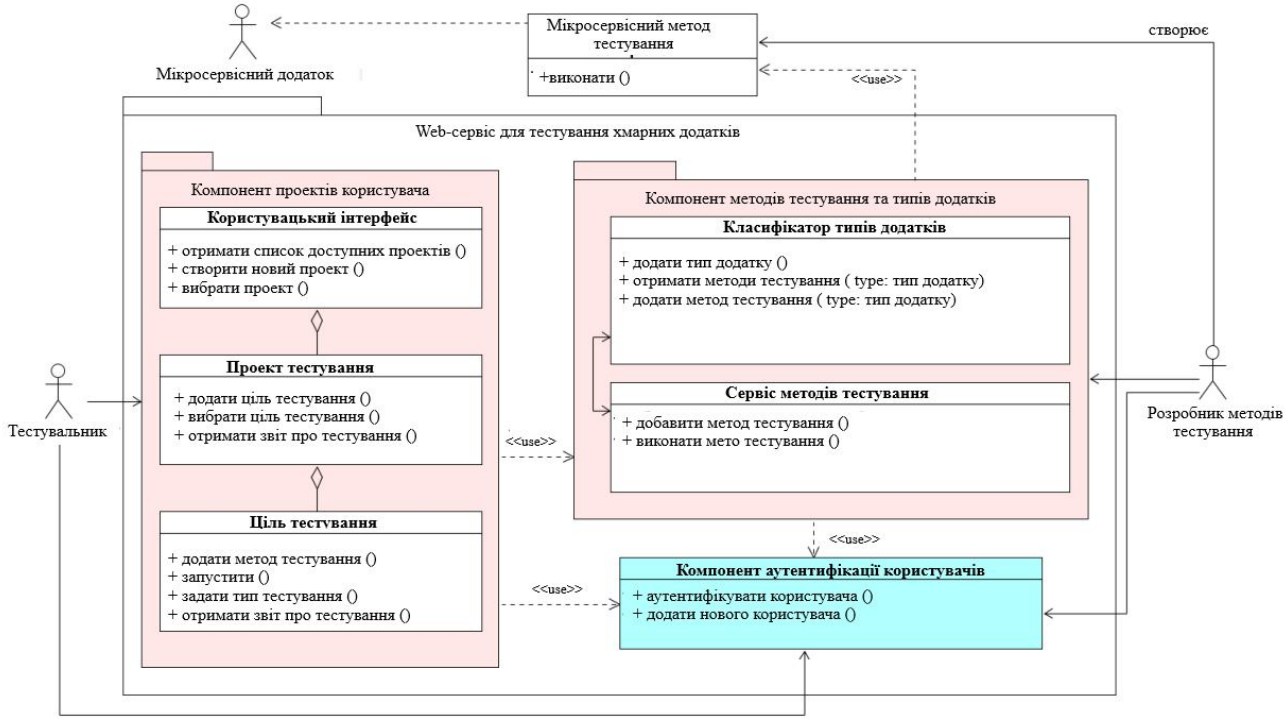

Рисунок 1.10 – Архітектура сервісу для тестування мікросервісних хмарних додатків

Компонент аутентифікації користувачів - компонент, який відповідає за ідентифікацію користувачів. Він дозволяє аутентифікувати користувача за допомогою логіна і пароля, відкритого SSH-ключа, а також за допомогою

екаунтів з соціальних мереж.

Компонент проектів користувача - компонент, який реалізує взаємодію з користувачем. Він агрегує проекти користувачів і надає зовнішні інтерфейси як у вигляді веб-інтерфейсу, так і у вигляді програмного інтерфейсу на основі REST-запитів. Сервіс зберігає дані по проектах, мету і конфігурацію методів тестування всіх користувачів. Для ідентифікації тестувальників сервіс взаємодіє з мікросервісом аутентифікації.

Компонент методів тестування і атрибутів додатків - компонент, відповідальний за зберігання даних по атрибутах додатів і зареєстрованих в системі методах тестування. Включає в себе класифікатор атрибутів додатків і сервіс методів тестування. При запитах по атрибутам додатки видаю відповідні методи тестуванняі необхідні параметри для них.

Мікросервіс методу тестування - окремий мікросервіс, що приймає на вхід конфігурацію методу тестування і повертає результат. Методи тестування інтегруються в систему за допомогою реєстрації в компоненті методів тестування і атрибутів додатків.

Визначимо процес створення проекту тестування з налаштуванням вньому цілей тестування, деталізація процесу представлена на рисунку 1.11:

1) тестувальник аутентифікується в системі за допомогою компонента аутентифікації користувачів, вводячи свій логін і пароль;

2) тестувальник створює новий проект тестування, вказуючи його назву;

3) після цього тестувальник створює мета тестування вказуючи її ім’я та опис;

4) існуючі в системі атрибути додатка запитую у сервісу методів тестування;

5) тестувальник вказує атрибути додатка, які підходять для його мети тестування;

6) на основі вибраних атрибутів компонент проектів користувача звертається до сервісу методів тестування і атрибутів додатку і запитує відповідні методи тестування для даних атрибутів додатку;

7) користувач вибирає і специфікує всі методи тестування, вказуючи необхідні параметри для них, такі як шлях до ресурсів,файли вихідного коду, обмеження за часом роботи і т.д.

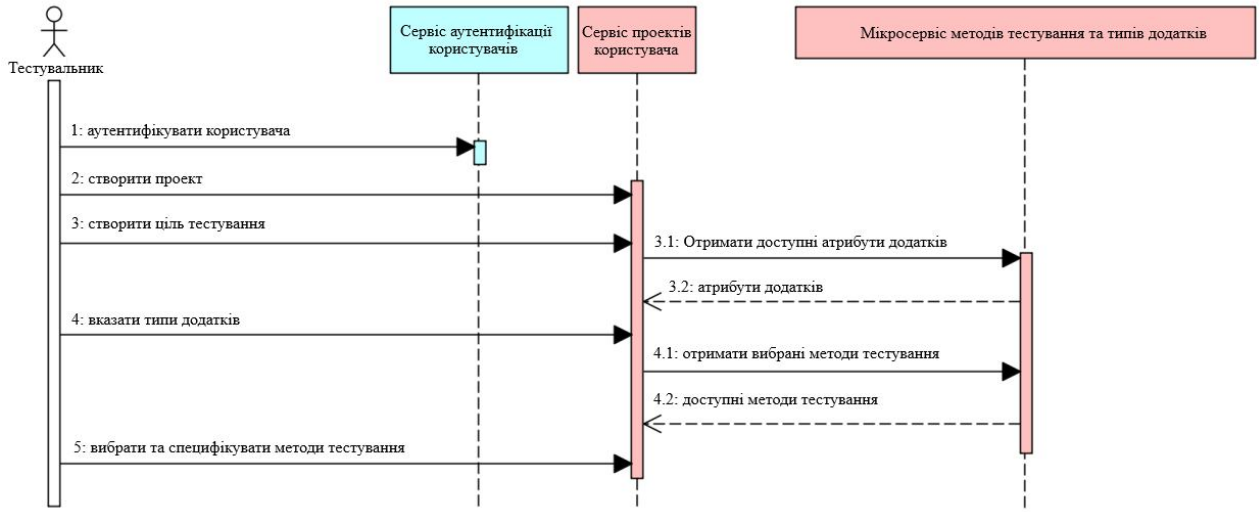

Рисунок 1.11 – Діаграма послідовності створення проекту тестування з налаштуванням цілей тестування

Також, опишемо стандартний алгоритм дії тестувальника з тестування додатка (рисунок 1.12).

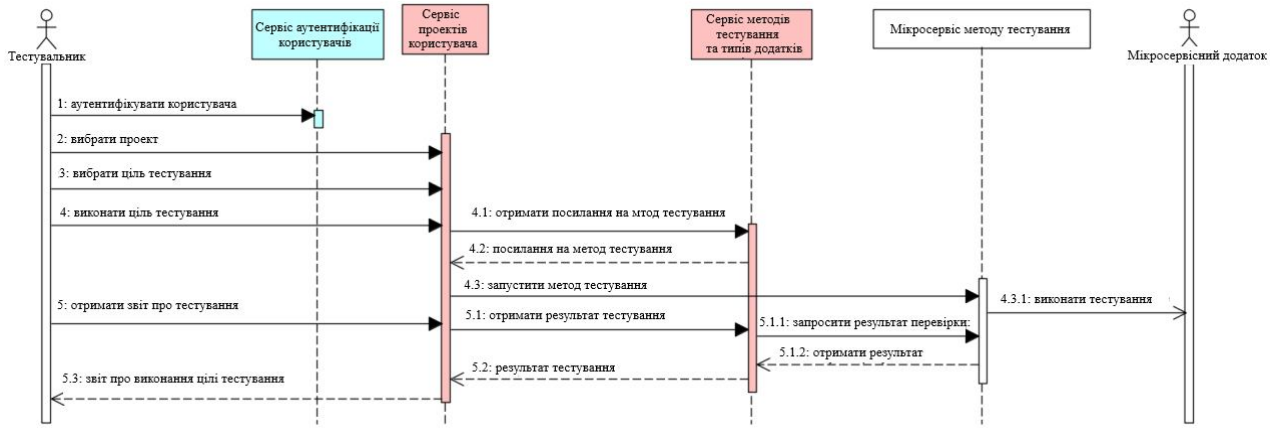

Рисунок 1.12 – Діаграма послідовності сценарію тестування мікросервісного додатку

1) тестувальник аутентифікується в системі за допомогою мікросервіса аутентифікації користувачів, вводячи свій логін і пароль;

2) далі він запитує необхідний проект від сервісу проектів користувача і вибирає потрібну мету тестування;

3) після чого тестувальник передає команду на виконання мети тестування компонента проектів користувача;

4) цей компонент звертається до компоненту методів тестування та запрошує посилання на методи тестування, зазначені в меті тестування;

5) отримавши дані, компонент проектів користувача запускає мікросервіси методів тестування в порядку, зазначеному в цілі тестування,передаючи їм необхідні дані;

6) сервіс методу тестування, витягує отримані дані і або розгортає у себе в середовищі тестований додаток, або підключається до вже запущеного, або виконує будь-яку іншу дію (наприклад, переглядає файли, для статичного аналізу вихідного коду)в залежності від призначення методу тестування. Після цього він виробляє тестування і повертає звіт про результати своєї роботи сервісу проектів користувача;

7) той же в свою чергу формує загальний звіт від всіх методів іпередає його тестувальникам.

Представлена схема бази даних сервісу для тестування,згенерована бібліотекою ERD для Ruby. Вона складається з наступних відношень:

- Testing Projects - відношення, що зберігає проекти користувачів, детальний опис представлено у таблиці 1.1;

Таблиця 1.1

СтруктуравідношенняTestingProjects

Найменування поля	Тип поля	Семантичний опис
id	INT NOT NULL	Первинний ключ
description	VARCHAR(255)	Опис проекту
title	INT NOT NULL	Назва
user_id	INT NOT NULL	Ідентифікатор користувача

- TestingTarget - відношення, що зберігає мету тестуваннядетальний опис представлено у таблиці 1.2;

Таблиця 1.2

Структура відношення TestingTarget

Найменування поля	Тип поля	Семантичний опис
id	INT NOT NULL	Первинний ключ
description	VARCHAR(255)	Опис мети
title	INT NOT NULL	Назва
testting_project_id	INT NOT NULL	Ідентифікатор проекту

- TestingMethod - відношення, в якому містяться посилання на зовнішні методи тестування та їх інтерфейс, детальний опис представлено у таблиці 1.3;

Таблиця 1.3

Структура відношення TestingMethod

Найменування поля	Тип поля	Семантичний опис
id	INT NOT NULL	Первинний ключ
fail_reason	TEXT	Причини
name	VARCHAR(255)	Назва
retried_at	DATETIME	Дата тестування
retry_count	INT	Результат
status	INT	Статус
url	VARCHAR(255)	Посилання на методи

- Users - відношення, яке містить користувачів, детальний опис представлено у таблиці 1.4;

Таблиця 1.4

Структура відношення Users

Найменування поля	Тип поля	Семантичний опис
id	INT NOT NULL	Первинний ключ
last_sign_in_at	DATETIME	Дата останнього входу
last_sign_in_ip	VARCHAR(255)	IP адреса останнього входу
remember_created_at	DATETIME	Дата
reset_password_sent_at	DATETIME	Даа зміни пароля
reset_password_token	VARCHAR(255)	Зміна пароля
sign_in_count	INT	Число входів
encrypted_password	VARCHAR(255)	Пароль
email	VARCHAR(255)	Пошта
curent_sign_in_at	DATETIME	Дата входу
curent_sign_in_ip	VARCHAR(255)	IP адреса входу

- MethodOptions –відношення, яке містить окремі параметри методів тестування, по суті реалізовує пару key-value, детальний опис представлено у таблиці 1.5 ;

Таблиця 1.5

Структура відношення MethodOptions

Найменування поля	Тип поля	Семантичний опис
id	INT NOT NULL	Первинний ключ
Name	VARCHAR(255)	Назва
parameter_type	VARCHAR(255)	Тип параметрів
possible_value	TEXT	Дійсні значення
testing method_id	INT	Ідентифікатор тесту

- ApplicationType - відношення, яке містить класифікатор атрибутів додатку, детальний опис представлено у таблиці 1.6;

Таблиця 1.6

Структура відношення ApplicationType

Найменування поля	Тип поля	Семантичний опис
id	INT NOT NULL	Первинний ключ
Name	VARCHAR(255)	Назва

- TargetMethod - відношення, яке містить параметри, які будутьпередаватися в метод тестування, при його запуску, детальний опис представлено у таблиці 1.7;

Таблиця 1.7

Структура відношення TargetMethod

Найменування поля	Тип поля	Семантичний опис
id	INT NOT NULL	Первинний ключ
testing_method_id	INT	Ідентифікатор методу тестування
testing_target_id	INT	Ідентифікатор цілі тестування

- TargetMethodOptions - відношення, що зберігає окремі заданізначення для кожного параметра в TargetMethods, детальний опис представлено у таблиці 1.8.

Таблиця 1.8

Структура відношення TargetMethodOption

Найменування поля	Тип поля	Семантичний опис
id	INT NOT NULL	Первинний ключ
method_option_id	INT	Ідентифікатор методу тестування
testing_project_id	INT	Ідентифікатор проекту
value	VARCHAR(255)	Результат

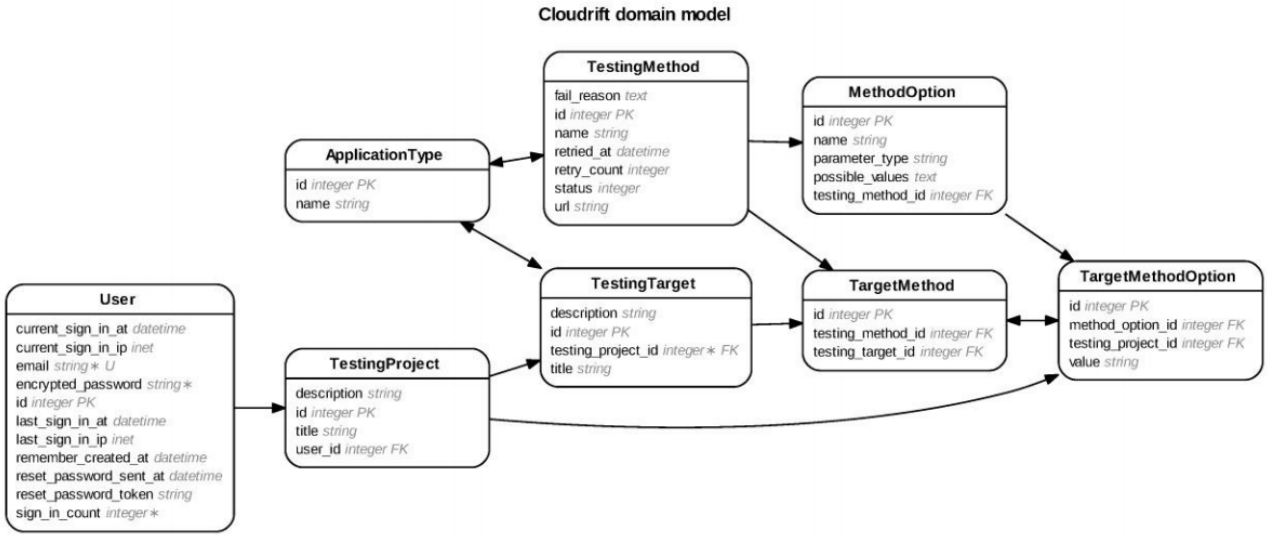

Рисунок 1.13 – Схема бази даних з вихідною інформацією

1.4 Висновки до першого розділу

У цьому розділі здійснено опис предметної області, напрями діяльності. Визначено склад функцій, що входять до бізнес-процесу на основі яких розроблено схему управління бізнес-процесом. Проведено аналіз відомих програмних систем. Здійснено аналіз вимог до програмної системи.

розроблено архітектуру web-сервісу, що дозволить краще зрозуміти функції основних його частин. Створено та описано структурну схему, основними компонентами якої є: рівень клієнта, рівень бізнес-логіки та рівень даних. Описано функціональну структуру системи та її основних елементів – модулів обробки даних. Визначено основні елементи бази даних та встановлено зв’язки між ними. Спроектовано структуру бази даних.

РОЗДІЛ 2 . ПРОГРАМНА РЕАЛІЗАЦІЯ СЕРВІСУ, ТЕСТУВАННЯ ТА ДОСЛІДНА ЕКСПЛУАТАЦІЯ ХМАРНИХ ДОДАТКІВ

2.1. Використовувані технології та стандарти та програмні інтерфейси серві

Сервіс використовує поєднання декількох технологій. До них відносяться мова Ruby, фреймворк Rails, база даних Postgres. Перевагу платформі Rails було віддано через те, що в ній є потужні інструменти для роботи з REST запитами. База Postgres була обрана також через легку інтегрованість в Rails.

Ruby on Rails - програмний каркас, написаний на мові програмування Ruby. Загальна архітектура представлена на рисунку 2.1. Ruby on Rails надає архітектурний зразок Model-View-Controller (модель-представлення-контролер) для веб-додатків, а також забезпечує їх інтеграцію з веб-сервером і сервером бази даних. Ruby on Rails є відкритим програмним забезпеченням і розповсюджується під ліцензією MIT.

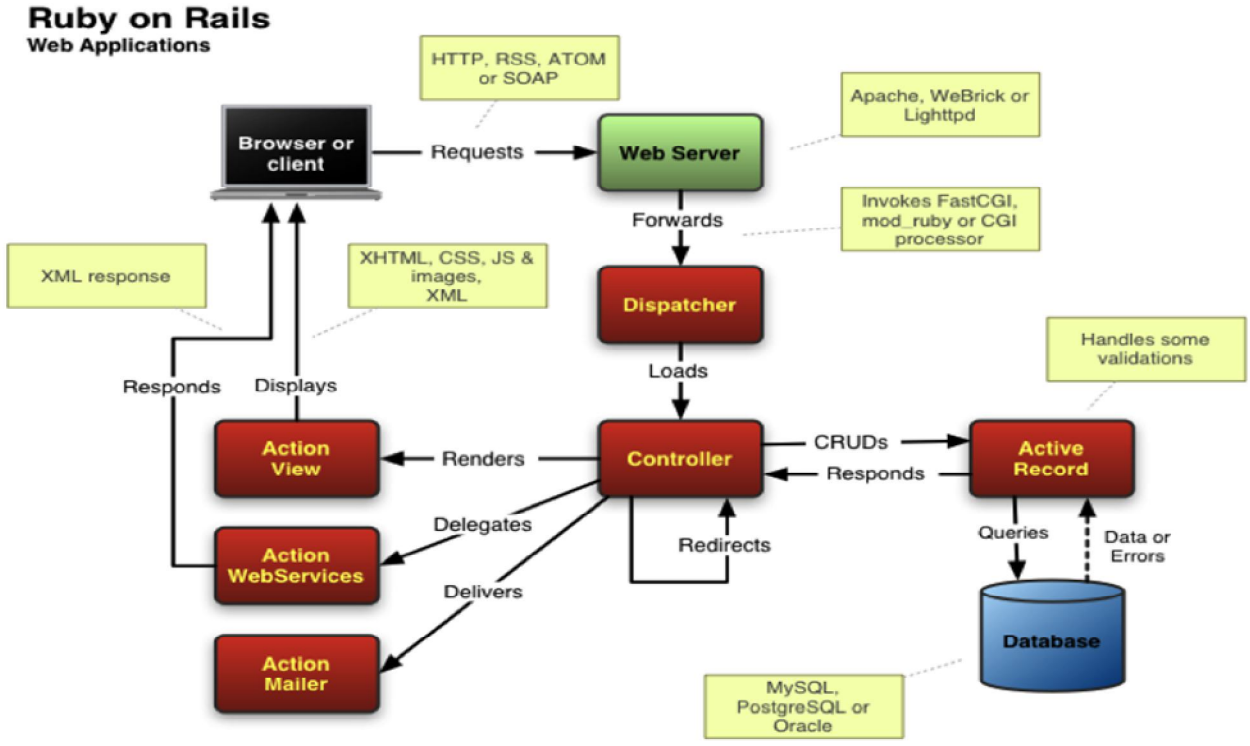

Рисунок 2.1 – Архітектура Ruby on Rails

Ruby on Rails визначає наступні принципи розробки додатків. Ruby on Rails надає механізми повторного використання, що дозволяють мінімізувати дублювання коду в додатках (принцип Don’t Repeat Yourself).

За замовчуванням використовується узгодженості конфігурації, типові для більшості додатків (принцип Convention over configuration). Явна специфікація конфігурації потрібна тільки в нестандартних випадках.

Основними компонентами додатків Ruby on Rails є модель (model), представлення (view) і контролер (controller). Модель надає іншим компонентам об’єктно-орієнтоване уявлення даних (таких як каталог продуктів або список замовлень). Об’єкти моделі можуть здійснювати завантаження і збереження даних в реляційній базі даних, а також реалізують бізнес-логіку.

Для зберігання об’єктів моделі в реляційній СУБД за замовчуванням в Rails 3 використана бібліотека ActiveRecord. Конкуруючий аналог - DataMapper.

Представлення створює користувальницький інтерфейс для відображення отриманих від контролера даних. Представлення також передає запити користувача на маніпуляцію даними в контролер (як правило, подання не змінює безпосередньо модель).

HTML і CSS. Для відображення інтерфейсу веб-додатку в браузері основними інструментами розробника є HTML і CSS.HTML - «мова розмітки документів, що використовує за основу гіпертекст» [11]. Браузери інтерпретують HTML і відображають отриманий в результаті форматований текст. Документ на мові html є набором елементів, початок і кінець яких обрамляються тегами (іменованими мітками) – відкриваючим та закриваючим відповідно. Кожен елемент може мати атрибути, що визначають властивість елемента, які вказуються в відкриваючому тегу.

HTML простий в розумінні, зручний і найбільш популярний, тому був обраний в якості інструменту для реалізації програми.

CSS - мова опису зовнішнього вигляду документа, написаного за допомогою мови розмітки. Розділяє опис структури документа від опису його

зовнішнього вигляду. «Кожне правило CSS з таблиці стилів має дві основні частини - селектор і блок оголошень. Селектор, розташований в лівій частині правила, визначає, на які частини документа поширюється правило. Блок оголошень розташовується в правій частині правила. Він міститься в фігурні дужки, і, в свою чергу, складається з одного або більше оголошень, розділених знаком «;». Кожне оголошення є поєднанням властивості CSS і значення, розділених знаком ":". Селектори можуть групуватися в одному рядку через кому. В такому випадку властивість застосовується до кожного з них »[6].

CSS значно зменшує обсяг атрибутів в HTML-коді. Є популярним універсальним засобом для відображення інтерфейсу. Поєднання HTML і CSS в сучасній веб-розробки практично нерозривно, тому після вибору HTML в якості одного з інструментів вибір CSS став очевидним.

Bootstrap. Bootstrap - «набір інструментів для створення веб-додатків» [2]. В цілому, можна сказати, що bootstrap - це фреймворк для зовнішнього представлення. Він містить в собі шаблони і компоненти для різних інструментів розробки веб-сторінок. За основу взята 12-ти колонна адаптивна розмітка, де кожна колонка має свій розмір щодо розміру пристрою і один одного.

Вибір даного інструменту, в першу чергу, обумовлений тим, що його в своїх рішеннях використовує тема Inspinia, а також рядом деяких вагомих переваг: він має повну і зрозумілу документацію; велика спільнота, регулярно доповнює Bootstrap новими рішеннями; безліч вже готових рішень; випущений під ліцензією MIT і є безкоштовним.

REST. «REST - архітектурний стиль взаємодії компонентів розподіленого додатка в мережі. REST є узгоджений набір обмежень, що враховуються при проектуванні розподіленої гіпермедіа-системи »[13]. Веб-додатки, побудовані з урахуванням REST, називаються RESTful.

Основні обмеження REST:

• Єдиний інтерфейс - визначає інтерфейс між сервером і клієнтом. Заснований на ресурсах, тобто сервер відправляє дані в форматі XML, HTML або JSON, що представляють записи в базі даних, а не її саму.

• Відсутність станів - необхідний стан клієнта для обробки запиту міститься в самому запиті. Серверу не потрібно знати про стан клієнта. Після обробки запиту стан повертається клієнтові через заголовки.

• Кешування відповіді - клієнт вміє кешувати відповіді від сервера, причому в них самих міститься інформації про необхідність кешування. Таким чином, усувається необхідність в зайвій взаємодії з сервером, внаслідок чого поліпшується продуктивність програми.

• Клієнт-сервер - сервер не пов'язаний з інтерфейсом або станом додатку, а клієнт не знає про зберігання даних. Таким чином клієнтська і серверні частини можуть бути замінні і розроблятися незалежно.

• Багаторівнева система - клієнт може взаємодіяти з «проміжними» серверами, таким чином забезпечується балансування навантаження і можливість додаткового кешування.

• Код на вимогу - розширення клієнта за рахунок передачі з сервера логіки, яку він може виконати. Є необов'язковим обмеженням.

Для інтеграції системи тестування мікросервісів з системами безперервної інтеграції та іншими зовнішніми сервісами в ній передбачений як для користувача графічний інтерфейс, так і JSONREST API. За допомогою графічного інтерфейсу відбувається конфігурація проекту, його цілей, методів; REST API дозволяє запускати цілі тестування, передаючи туди необхідні параметри, а також запитувати статус тестування, лог виконання і помилок. Всі дії,ініційовані за допомогою API, можливо виконувати також і за допомогою графічного інтерфейса.

На рисунку3.2 представлений інтерфейс конфігурації нової метитестування. Крім очевидних полів, від користувача вимагається надати інформацію про те, до яких атрибутів відноситься додаток. Після цього система

може запропонувати ряд методів тестування, характерних для об’єднання зазначених атрибутів додатків.

Рисунок 2.2 – Конфігурація нової цілі тестування

Після створення мети можна задати ряд параметрів, що не змінюються від запуску до запуску, а також настроїти послідовність і логіку роботи методів тестування (чи повинна помилка в одному з методів приводити до аварійного завершення всієї мети). Сконфігуровану мету можна запустити за допомогою POST-запиту, приклад якого можна побачити на рисунку 2.3.

```
POST /api/v1/targets/my_netty_target
{
  source: "git@github.com:skayred/netty-upload-example.git",
  methods: "all",
  framework: "maven",
  endpoint: "https://netty-upload.herokuapp.com"
}
```

Рисунок 2.3 – JSON-запиту на запуск мети тестування

Поля JSON-запиту містять інформацію, необхідну методам тестування, які входять в ціль. В даному запиті надається розташування вихідного коду програми, список методів (в даному випадку запускаються всі методи тестування), фреймворк (потрібно методом юніт-тестування), а також розташування розгорнутого веб-сервісу для запуску навантажувальних тестів, тестів безпеки, а також, можливо, функціональних тестів, що використовують зовнішній інтерфейс сервісу та реалізованих користувачем.

2.2. Архітектура окремих методів тестування та процес створення компонентів сервісу

Кожен сервіс має стандартний Rest API і надаєтакі ресурси:

- GET / interfaces - ресурс, який повертає параметри, необхідні сервісом для тестування, кожен параметр складається з імені та типу,приклад інтерфейсу зображений на рисунку 2.4;

- POST / methods - ресурс, який приймає на вхід необхідні параметри, задані інтерфейсом, запускає сценарій тестування та повертає ідентифікатор по якому можна отримати результат перевірки.

- GET / methods / [id сценарію тестування] - ресурс, який повертає звіт про виконання сценарію тестування у вигляді JSON.

Система складається з основного модуля і окремих модулів методів тестування. Кожен модуль це окремий мікросервісний додаток,взаємодіє з іншими модулями за допомогою REST API.

Всі сервіси реалізовані на базі платформи Ruby on Rails. Перевага даної мови і платформи:

- вбудована підтримка патерна MVC;

- розвинене співтовариство і велика кількість існуючих компонентів;

- можливість генерувати необхідні моделі, контролери та відображення в напівавтоматичному режимі за допомогою вбудованих в платформу засобів.

```
params: [
  {
    name: 'method'
    type: "enum",
    values: ["GET", "POST", "PUT", "PATCH", "DELETE", "HEAD", "OPTIONS"]
  },
  {
    name: 'url'
    type: "string"
  },
  {
    name: 'request'
    type: "hash"
  },
  {
    name: 'response'
    type: "hash"
  }
]
```

Рисунок 2.4 – Приклад інтерфейсу окремого методу тестування

Структура проекту формується на основі структури директорій. Основні компоненти представлені в таблиці 3.1.

Таблиця 3.1

Основні компоненти проекту RoR

app/	Основний код додатка (app), включає моделі, представлення, контролери та хелпери
config/	Конфігурація додатка
db/	Файли баз даих
vendor/	Код сторонніх розробників, такий як плагіни і геми
test/	Тест додатків
vendor/	Код сторонніх розробників, такий як плагіни і геми
gemfile	Бібліотеки необхідні даному додатку

При створенні програми використовувалася розподілена система контролю версій Git. Основними перевагами даної системи є:

- висока продуктивність;

- простота управління вихідним кодом.

Як інструмент для створення веб-інтерфейсу був обраний Sematic UI. Його основними перевагами є велика кількість існуючих компонентів, зручний сайт з керівництвом по використанню і, внаслідок цього, велике прискорення розробки графічних веб-інтерфейсів.

Для реалізації аутентифікації користувачів використаний сторонній ruby-devise (приклад аутентифікації користувача представлено на рисунку 2.5.

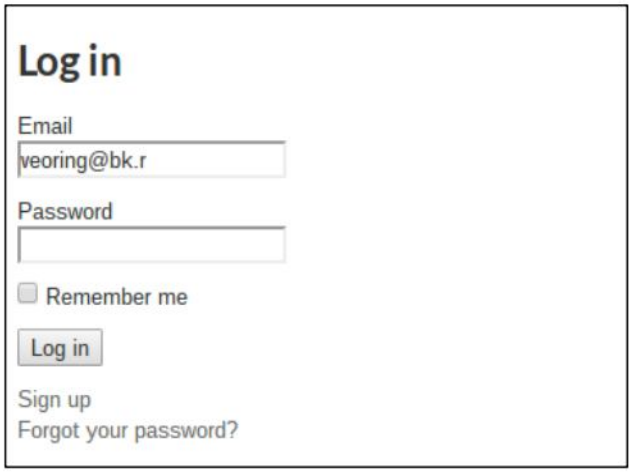

Рисунок 2.5 – Приклад форми аутентифікації, створюваної за замовчуванням Devise

Виділяють наступні переваги:

- є завершеним MVC-рішенням, заснованим на Rails;

- легкий у використанні;

- заснований на модульній: використовує тільки те, що вам дійсно необхідно.

Пакет створює таблицю User в базі даних; додає моделі, контролери та подання для аутентифікації та реєстрації користувачів.Для роботи з ним була додано посилання на вхід / вихід користувачав головне меню і для кожного проекту тестування встановлено зовнішнійключ на користувача.

Створення компонентів сервісу тестування хмарних додатківтаких, як проекти тестування, методи тестування, цілі тестуванняі інших, відбувалося схожим чином. Опишемо послідовність дій на прикладі створення проекту тестування.

По-перше, необхідно виконати в папці з проектом програми наступну команду: generate scaffold TestingProject name: string, description: text, user: references. Ця команда генерує повний набір з моделі, міграції бази даних для цієї моделі, контролер для впливу на неї, представлення для перегляду і поводження з даними і тестовий набір для всього цього (рисунок 2.6).

```
      invoke  active_record
          create    db/migrate/20130717151933_create_testing_projects.rb
          create    app/models/testing_project.rb
          invoke    test_unit
          create      test/models/testing_project_test.rb
          create      test/fixtures/ testing_project.yml
          invoke  resource_route
           route    resources : testing_projects
          invoke  scaffold_controller
          create    app/controllers/testing_projects_controller.rb
          invoke    erb
          create      app/views/testing_projects
          create      app/views/testing_projects/index.html.erb
          create      app/views/testing_projects/edit.html.erb
          create      app/views/testing_projects/show.html.erb
          create      app/views/testing_projects/new.html.erb
          create      app/views/testing_projects/_form.html.erb
          invoke    test_unit
          create      test/controllers/testing_projects_controller_test.rb
          invoke    helper
          create      app/helpers/testing_projects_helper.rb
```

Рисунок 2.6 – Згенеровані файли при виконанні команди scaffold

По-друге, необхідно в міграції бази даних вказати foreign keудля User (рисунок 2.7) і застосувати міграцію виконавши команду rake db: migrate.

```
      class CreateTestingProjects < ActiveRecord::Migration
        def change
          create_table :testing_projects do |t|
            t.string :title
            t.string :description
              add_reference  :testing_projects,  :user,  index:true,  for-
eign_key: true
            t.timestamps null: false
          end
        end
      end
```

Рисунок 2.7 – Міграція для моделі TestingProject

По-третє, в моделі вказати, що TestingProject належить User і містить множину TestingTarget і додати функцію запиту до БД (scope) для отримання всіх проектів, що належать користувачеві (рисунок 2.8).

```
      class TestingProject < ActiveRecord::Base
        belongs_to :user
        has_many :testing_targets, dependent: :destroy
        has_many :target_method_options, dependent: :destroy

        scope :for_user, lambda { |user|
          where(user: user)
        }
      end
```

Рисунок 2.3 – Модель для Testing Project

Далі необхідно налаштувати контролер. За замовчуванням scaffold генерує такі методи:

1) index - для відображення всіх тестових проектів;

2) new - для сторінки створення нового проекту;

3) create - для безпосереднього створення нового проекту тестування;

4) update - для сторінки редагування проекту;

5) show - для сторінки відображення обраного проекту користувача;

6) destroy - для видалення обраної сторінки користувача.

```
      class TestingProjectsController < ApplicationController

        def index
          @testing_projects = TestingProject.for_user(current_user)
        end

        def show
          @testing_targets = @testing_project.testing_targets
          params[:testing_project_id] = @testing_project.id
        end

        # GET /testing_projects/new
        def new
          @testing_project = TestingProject.new

        end

        def edit
        end

        def create
          @testing_project = TestingProject.new(testing_project_params)
          current_user
          @testing_project.user = current_user
          respond_to do |format|
            if @testing_project.save
              format.html { redirect_to @testing_project, notice: 'Testing
project was successfully created.' }
            else
              format.html { render :new }
            end
          end
        end


        def update
          respond_to do |format|
            if @testing_project.update(testing_project_params)
              format.html { redirect_to @testing_project, notice: 'Testing
project was successfully updated.' }
            end
          end
        end

        def destroy
          @testing_project.destroy
          end
        end
      end
```

Рисунок 2.9 – Контролер для Testing Project

У index потрібно замінити отримання всіх проектів тестування на ті проекти, які належать користувачеві поточної сесії. В show слід додати отримання всіх цілей тестування, що належать даному проекту. Для show, edit, delete слід ввести перевірку на те, що користувач має доступ до запитуваного проекту. Для цього введемо перевірку перед даними запитами. Код контролера представлений на рисунку 2.9.

Далі необхідно налаштувати представлення. Кожна сторінка є HTML-файлом з вставками на ruby. Для оформлення використовувався Sematic UI, і можна використовувати його CSS-класи. У потрібні місця потрібно підставити елементи, взяті з контролера. У нашому випадку для index це буде назва проекту, його опис і існуючі цілі тестування (рисунок 2.10).

```
      <h1 class="ui center aligned header">Projects</h1>
      <div class="ui two item menu">
        <%= link_to 'New Testing project', new_testing_project_path, class:
"item" %>
        <%= link_to 'Existing Testing project', testing_projects_path, class:
"item active" %>
      </div>


      <div class="ui relaxed divided list">
        <% @testing_projects.each do |testing_project| %>
            <div class="item">
              <i class="file text outline icon"></i>
              <div class="content">
                <%= link_to testing_project.title, testing_project, class:
'header' %>
                <div   class="description"><%=   testing_project.description
%></div>


                <%= link_to testing_project, method: :delete, data: {confirm:
'Are you sure?'} do %>
                    <i class="trash right icon"></i>
                <% end %>


                <%= link_to edit_testing_project_path(testing_project) do %>
                    <i class="edit icon"></i>
                <% end %>
              </div>


            </div>
        <% end %>
      </div>
```

Рисунок 2.10 – Представлення для відображення всіх проектів тестування

Останнім етапом потрібно виконати додавання маршрутів до методів контролера, це можна легко зробити, використавши метод resources, який автоматично створить маршрути для стандартних REST запитів, create, index, і т.д. Посилання на нестандартні методи контролера, слід додавати всередину resources, приклад зображений на рисунку 2.11.

```
Rails.application.routes.draw do
  root 'testing_projects#index'

  resources :testing_projects, shallow: true do
    member do
      get :application
    end
    resources :testing_targets
  end
end
```

Рисунок 2.11– Приклад опису маршрутів для TestingProject

Сервіс використовує множину рішень, описаних раніше. До них відносяться мова Ruby, фреймворк Rails, база даних Postgres. Перевагу платформі Rails було віддано через те, що в ній є потужні інструменти для роботи з REST запитами. База Postgres була обрана також через легку інтегрованість в Rails.

Сервіс використовує REST інтерфейс, описаний раніше. Наведемо його реалізацію. Метод interface просто повертає JSON, створений заздалегідь (рисунок 2.12), він являє собою параметри, необхідні методу тестування(посилання на репозиторій, адреса розгорнутих додатків).

```
class InterfacesController < ApplicationController
  INTERFACE = {
    params: [
      {
        name: 'method',
        parameter_type: "enum",
        possible_values: ["GET", "POST", "PUT", "PATCH", "DELETE",
"HEAD", "OPTIONS"]
      },
      {
        name: 'url',
        parameter_type: "string"
      },
      {
        name: 'request',
        parameter_type: "hash"
      },
      {
        name: 'response',
        parameter_type: "hash"
      }
    ]
  }

  def show
    render json: INTERFACE
  end
end
```

Рисунок 2.12 – Контролер інтерфейсу методу тестування

Метод create приймає параметри для тестування, створює запис в базі даних про тестування, запускає асинхронне завдання з тестування і повертає користувачеві ідентифікатор, який вказує на запис в БД. (рисунок 2.13).

```
def create
  testing_case = TestingCase.new(result:'')
  testing_case.save
  answer = {id: testing_case.id}
  Thread.new do
    make_testing params, testing_case.id
  end
  render json: answer
end
```

Рисунок 2.13 – Програмний код створення нового потоку

Метод show запитує у БД результат тестування, якщо поле result пусте, то сервіс повертає відповідь про те, що тестування не закінчилося, в іншому випадку повертається відповідь, що результат записаний в базі даних. Вимагає передачі ідентифікатора тестування.

Метод make_testing виконує саме тестування (рисунок 2.14). Переданий JSON з параметрами тестування конвертується в хеш-таблицю,з неї витягуються дані про адресу тестованого сервісу, приклад запитіві приклад відповідей, які повинні приходити у відповідь, далі виконуються реальні запити до тестованого сервісу і порівнюються відповіді очікувані та дійсні. У реалізації цього методу використовується бібліотека Airborne, яка спрощує запити до зовнішнього сервісу і роботу з ними.

```
def make_testing(params,id)
    method = params[:method]
    url = params[:url]
    request = params[:request] || "{}"
    response = params[:response] || "{}"
    response = JSON.parse response
    airborne = AirTest.new
    reponse_real = JSON.parse(airborne.get(url))
    if  reponse_real == response
      result = 'success finish'
    else
      result = 'dismatch response expected: ' + response.to_s + ' re-
turned: '+ reponse_real.to_s
    end
    testing_case = TestingCase.find(id)
    testing_case.result = result
    testing_case.save
  end
```

Рисунок 2.14 – Програмний код тестування інтерфейс

2.3. Тестування та розгортання сервісу для тестування хмарних додатків.

Коли реалізований функціонал додатку, необхідно перевірити коректність його роботи. Зазвичай це завдання лягає на плечі людей, що займаються тестуванням. Але розробник бажає перевірити працездатність свого коду на етапі, коли ще не так складно щось змінити, а ще краще - автоматизувати цей процес. До того ж, не варто забувати про людський фактор, через який навіть професійний тестувальник може не помітити який-небудь недолік програми. Так з'явилася ідея тестування на рівні коду, для чого стали розроблятися інструменти, стандарти і принципи.

Тестування коду розділяється на кілька рівнів, в залежності від складності тестувальної структури: юніт-тестування, інтеграційне тестування, навантажувальне тестування і т.д.

Оскільки розробка всіх модулів здійснювалася з використання Ruby on Rails, то і тестування здійснюється за допомогою вбудованого вRails платформи тестування Minitest (рисунок 2.15). Вона забезпечує наступні види тестування:

- модульне тестування - тестування, яке покриває моделі і допоміжні методи. Найчастіше перевіряється коректність запису даних в БД, наприклад, перевірка на те, що адреса електронної пошти не може бути порожня, повинен містити знак @ і інші;

- функціональне тестування - тестування, що перевіряє роботу призначеного для користувача інтерфейсу за допомогою окремих HTTP-запитів. Середовище Rails дозволяє легко ініціювати окремі команди HTTP, GET і POST, формуючи основу тестів. Прикладами даних тестів, можеслужити коректність виведення списку елементів з БД, перевірка успішності запису в БД після запиту create та інші;

- інтеграційне тестування, тестування аналогічне функціональному в рамках Minitest, але дозволяє каскадне виконання HTTP запитів,

використовується для перевірки різних частин програмиі виконання великих тестових сценаріїв.

В рамках системи написано 60 тестів:

- 15 тестів моделей;

- 20 функціональних тестів;

- 25 інтеграційних тестів.

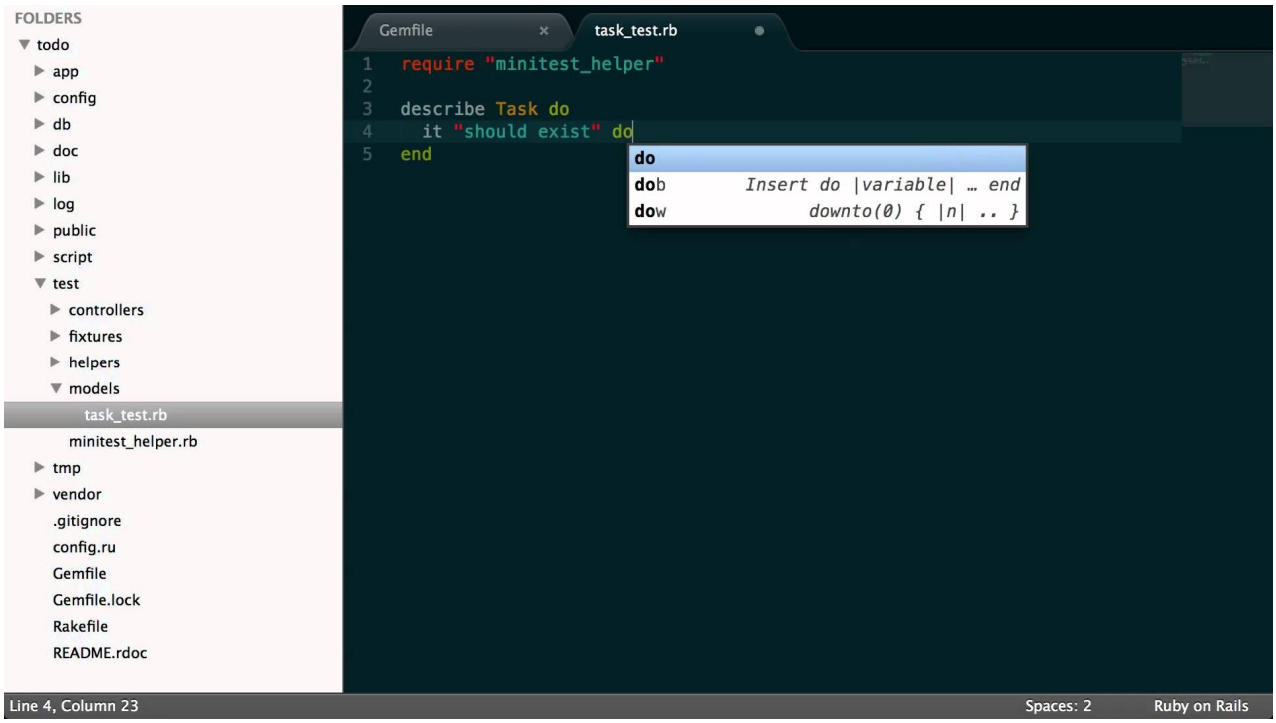

Рисунок 2.15 – Тестування за домогою Rails Minitest

Модульне тестування. Розглянемо приклад реалізації тестів на прикладі моделі користувача User.

Тестування моделей включає в себе наступні перевірки:

- перевірка коректності збереження користувача, з правильно введеними параметрами;

- перевірка на те, що ім’я користувача та e-mail не порожні;

- перевірка на те, що ім’я користувача не довше 51 символу;

- перевірка на те, що email не може перевищувати 244 символи.

- перевірка на те, що email відповідає зразку, тобто існуєзнак @ в запису, існує доменне ім’я і використовуються англійські літери;

- перевірка на те, що пароль складається з цифр і букв верхнього та нижнього регістру.

Код тестування моделей наведено на рисунку 2.16.

```
      class UserTest < ActiveSupport::TestCase
        def setup
          @user = User.new(name: "Example User", email: "user@example.com",
                           password: "foobar", password_confirmation: "foo-
bar")
        end

        test "should be valid" do
          assert @user.valid?
        end

        test "name should be present" do
          @user.name = "     "
          assert_not @user.valid?
        end

        test "email should be present" do
          @user.email = "     "
          assert_not @user.valid?
        end

        test "name should not be too long" do
          @user.name = "a" * 51
          assert_not @user.valid?
        end

        test "email should not be too long" do
          @user.email = "a" * 244 + "@example.com"
          assert_not @user.valid?
        end

        test "email validation should accept valid addresses" do
          valid_addresses = %w[user@example.com USER@foo.COM A_US-
ER@foo.bar.org
                               first.last@foo.jp alice+bob@baz.cn]
          valid_addresses.each do |valid_address|
            @user.email = valid_address
            assert @user.valid?, "#{valid_address.inspect} should be valid"
          end
        end

      end
```

Рисунок 2.16 – Приклад тестування моделі User

Функціональне тестування проводилося при тестуванні контролерів і включало в себе наступні перевірки:

- перевірка на те що існує адреса створення нового користувача;

- перевірка на те, що якщо користувач не авторизований, виконається перенаправлення на авторизацію при спробі редагування користувача;

- перевірка на те, що не можна редагувати іншого користувача;

- перевірка на необхідність авторизації в системі.

Приклад коду функціональних тестів зображений на рисунку 2.17.

```
      class UsersControllerTest < ActionController::TestCase
        def setup
          @user = users(:michael)
          @other_user = users(:archer)
        end

        test "should get new" do
          get :new
          assert_response :success
        end

        test "should redirect edit when not logged in" do
          get :edit, id: @user

          assert_not flash.empty?
          assert_redirected_to login_url
        end

        test "should redirect update when not logged in" do
          patch :update, id: @user, user:
      { name: @user.name, email: @user.email }

          assert_not flash.empty?
          assert_redirected_to login_url
        end

        test "should redirect edit when logged in as wrong user" do
          log_in_as(@other_user)
          get :edit, id: @user
          assert flash.empty?
          assert_redirected_to root_url
        end

        test "should redirect update when logged in as wrong user" do
          log_in_as(@other_user)
          patch :update, id: @user, user: { name: @user.name, email:
@user.email }
          assert_redirected_to root_url
        end

        test "should redirect index when not logged in" do
          get :index
          assert_redirected_to login_url
        end
      end
```

Рисунок 2.17 – Приклад функціонального тестування контролера User

Інтеграційне тестування. Для перевірки на правильність входу в систему реалізовані наступні тести:

- перевірка на те, що неможливо увійти в систему з некоректною інформацією;

- перевірка на правильність виходу, входу з коректною інформацією і відображення посилань на вхід вихід в верхньому меню;

- перевірка коректності збереження сесій.

Інтеграційні тести зазвичай об’ємні, тому наведемо лише частину коду на рисунку 2.18.

```
class UsersLoginTest < ActionDispatch::IntegrationTest
  test "login with invalid information" do
    get login_path

    post login_path, session: {email: "", password: ""}

    assert_not flash.empty?
    get root_path
    assert flash.empty?
  end

  test "login with valid information followed by logout"  do
    get login_path
    post login_path, session:{ email: @user.email, password: 'pass-
word'}
    assert is_logged_in?
    assert_redirected_to @user
    follow_redirect!
    assert_template 'users/show'
    assert_select "a[href=?]", login_path, count: 0
    assert_select "a[href=?]", logout_path
    delete logout_path
    assert_not is_logged_in?
    assert_redirected_to root_url
    # Simulate a user clicking logout in a second window.
    delete logout_path
    follow_redirect!
    assert_select "a[href=?]", login_path
    assert_select "a[href=?]", logout_path, count: 0
    assert_select "a[href=?]", user_path(@user), count: 0
  end

  test "login with remembering" do
    log_in_as(@user, remember_me: '1')
    assert_not_nil cookies['remember_token']
  end
end
```

Рисунок 2.18 – Приклад інтеграційного тестування входу користувача в систему

Необхідно відмітити, що результат тестування сервісу для тестування хмарних додатків– позитивний.

Для розгортання web-сервісу була обрана хмарна PaaS платформа Heroku (рисунок 2.19). Ця платформа дозволяє безкоштовно розгортати до 5 ruby додатків і легко інтегрується з Git.

Для розміщення сервісів в хмарі використовувався наступний алгоритм:

1) зареєструватися на сайті heroku.com;

2) встановити модуль Heroku Toolbelt в систему: https://toolbelt.heroku.com/install-ubuntu.sh | sh;

3) увійти в профіль з командного рядка: heroku login;

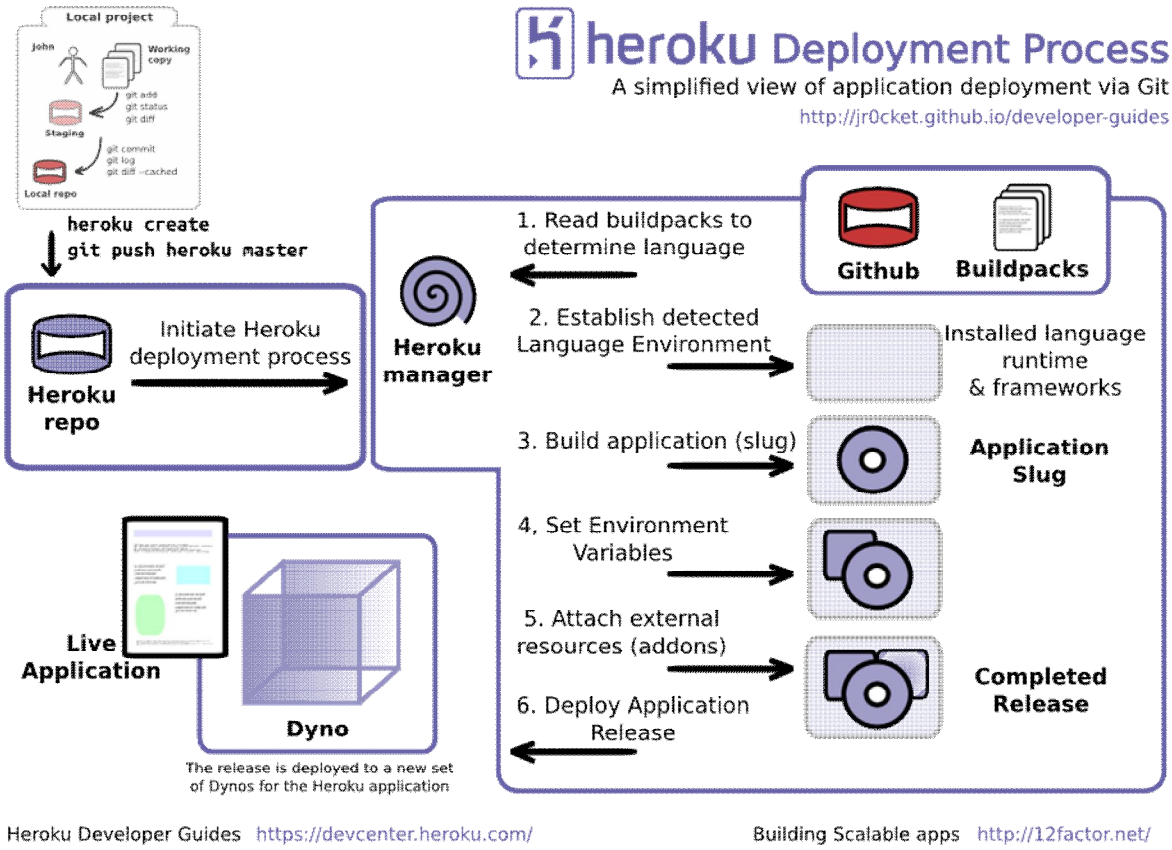

Рисунок 2.19 – Процедура розгортання додатку на платформі Heroku

4) ініціювати heroku в папці з проектами: heroku init;

5) запустити проект на heroku: git push heroku master;

6) відкрити сторінку з додатком: heroku open.

Розробка інтерфейсу web-сервісу враховує принцип доброзичливого ставлення до користувача. Інтерфейс повинен бути зручним і інтуїтивно зрозумілим навіть початківцю. Основною складовою в плані зовнішнього вигляду програми, колірних рішень і використання компонентів для призначеного для користувача взаємодії з системою є можливість легкого освоєння.

На рисунку 2.20 форму реєстрації користувача для повноцінного використання сервісу.

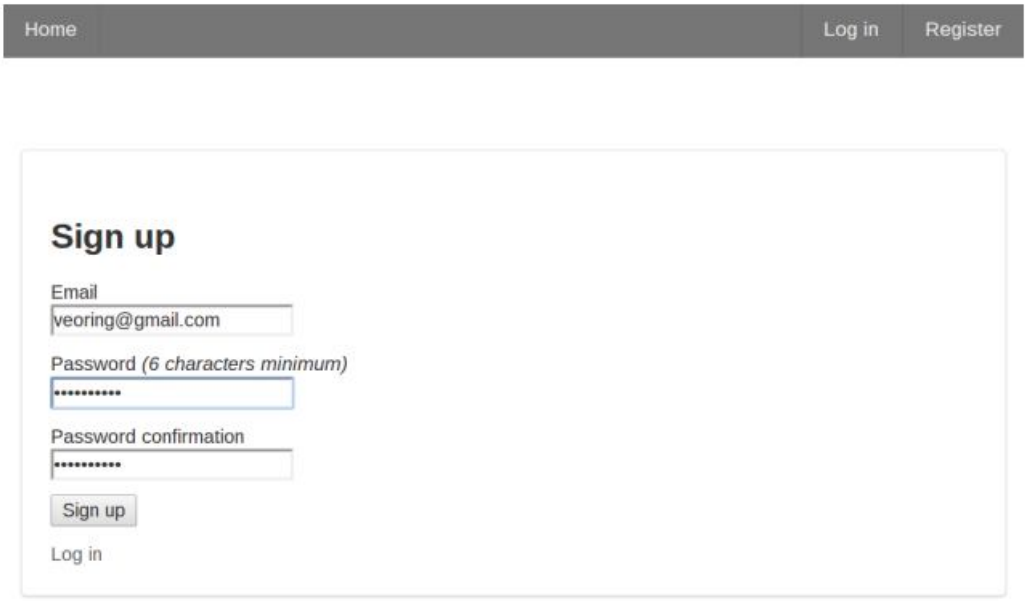

Рисунок 2.20 – Сторінка реєстрації користувача

Зовнішній вигляд сервісу можна умовно поділити на п’ять розділів: головна сторінка; взаємодія з неавторизованих користувачем; сторінки користувача, сторінки проектів, сторінки методів тестування. Розглянемо розробку кожного розділу докладніше.

На рисунку 2.21 представлено сторінку для додавання нового проекту для тестування.

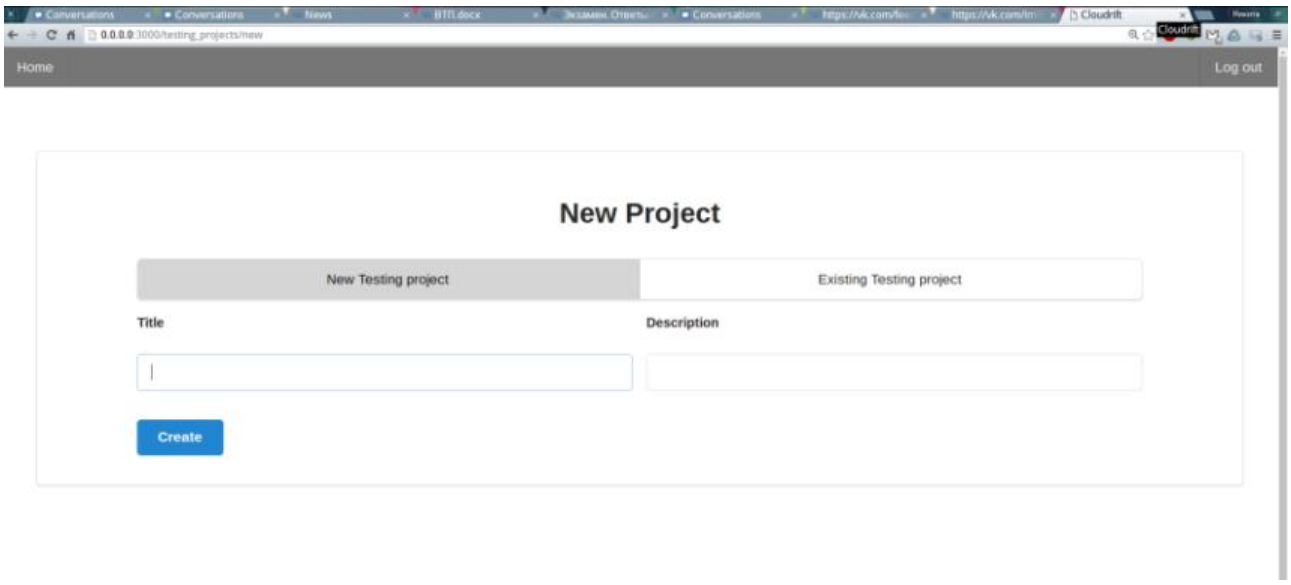

Рисунок 2.21 – Додавання нового проекту тестування

Короткий опис функціоналу додатку, який присвячений управлінню проектами тестування. Даний функціонал розділено на 4 категорії:перегляд проектів (рисунок 2.22), додавання нового проекту, редагування проекту та його видалення.

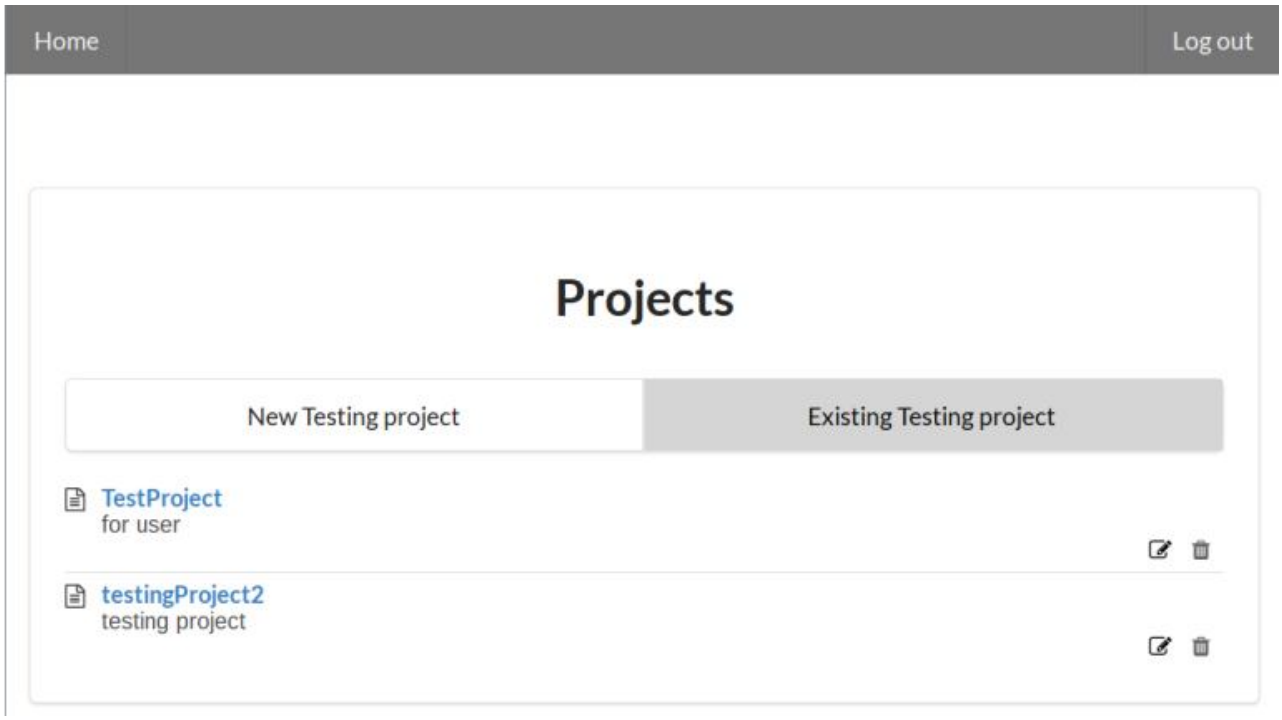

Рисунок 2.22 – Сторінка перегляду проектів

Для створення нової мети тестування необхідно заповнити відповідні поля, які представлено на формі рисунку 2.23. На рисунку 2.24 представлено

сторінку з докладним описом мети тестування, вибору відповідних параметрів тестування.

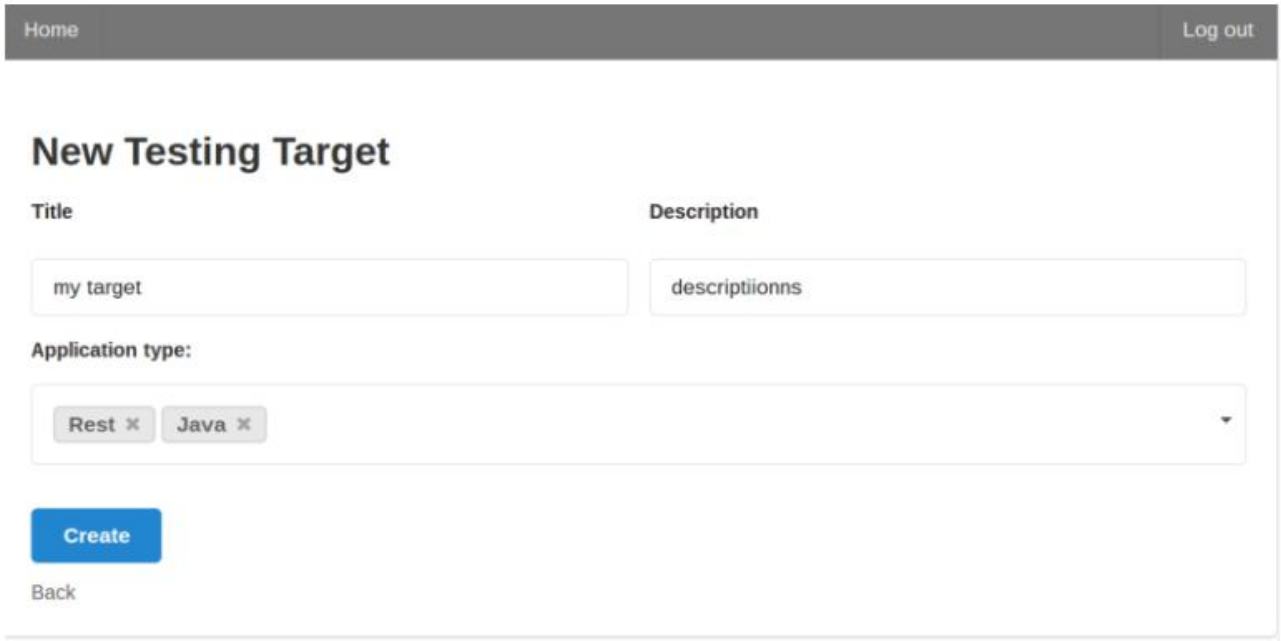

Рисунок 2.23 – Сторінка додавання мети тестування

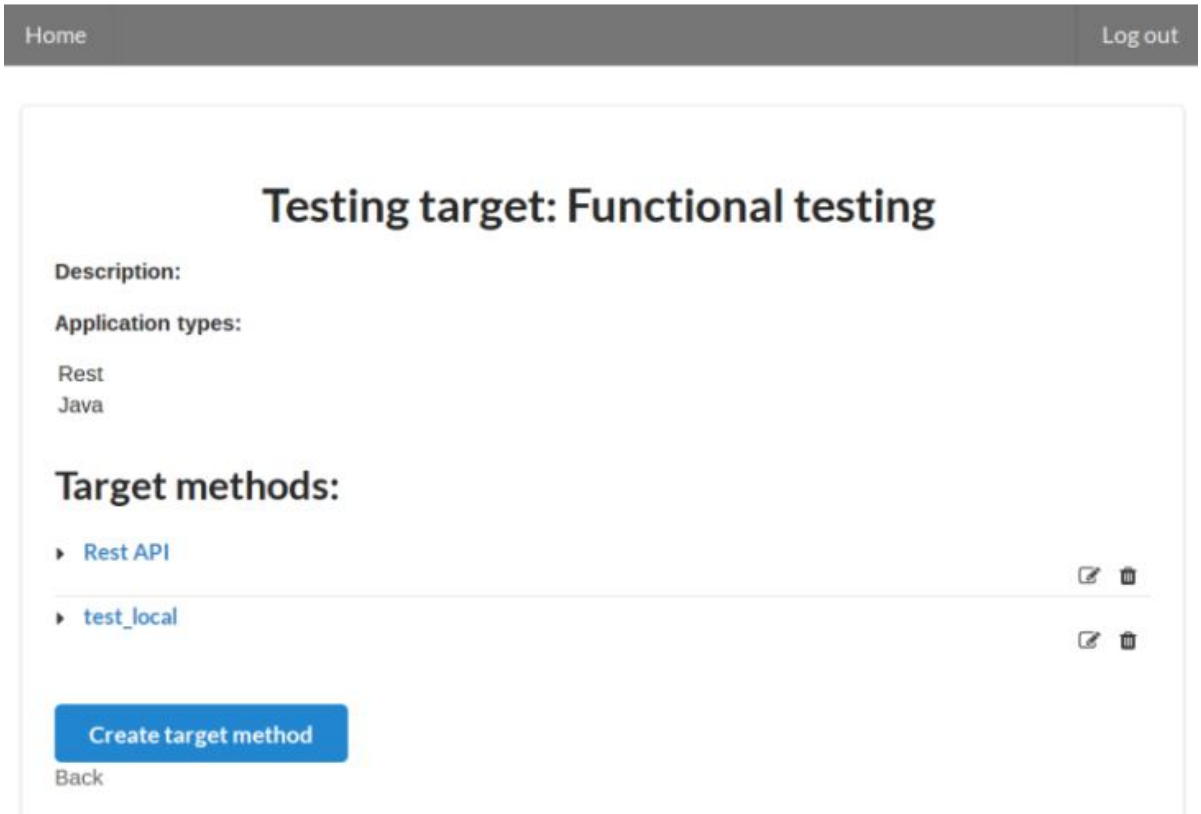

Рисунок 2.24 – Сторінка мети тестування

Сторінка налаштування методу тестування виконана в звичному для всіх додатків варіанті. Користувачеві пропонується вибрати відповідний метод та відповідні параметри. Процедуру налаштування представлено на рисунку 2.25.

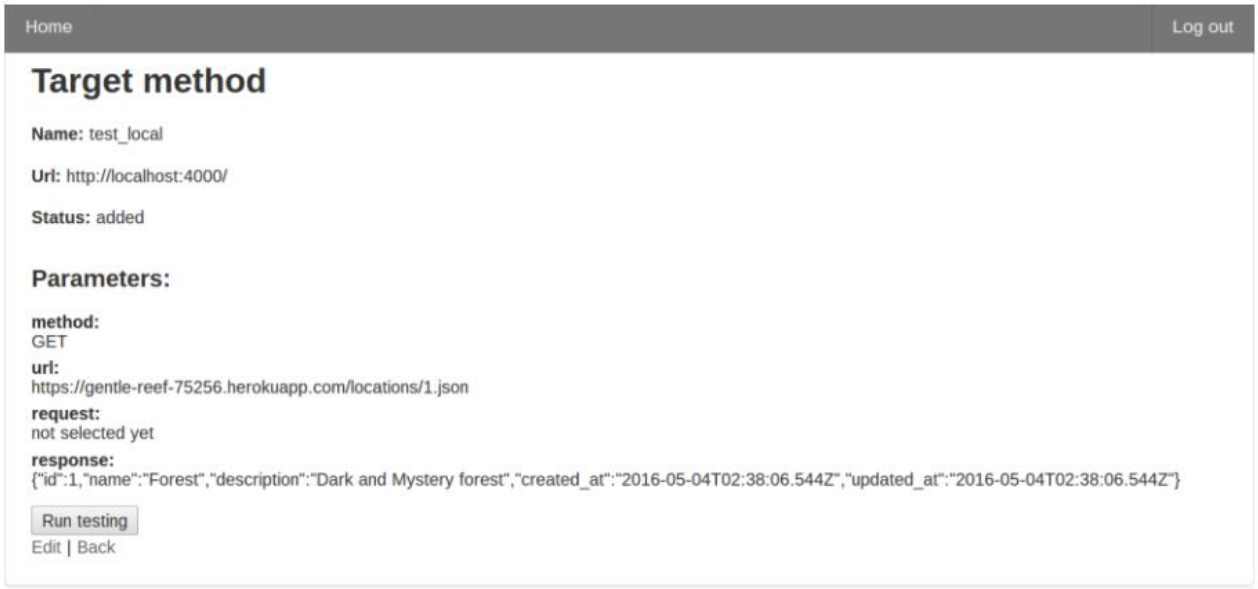

Рисунок 2.25 – Сторінка налаштування методу тестування

Розробка інтерфейсу велася з урахуванням зручності використання програми як на ПК, так і на планшетах. Для досягнення однаковості і впізнаваності інтерфейсу необхідно було знайти компроміс, який задовольняє два варіанти.

1.4 Висновки до другого розділу

У даному розділі обґрунтовано технологію, мову програмування та розроблено програмний севіс. Обґрунтовано засоби розробки та створено програмну реалізацію сервісу для тестування хмарних додатків.

Здійснено опис процедур тестування та їхніх результатів, описані тест-вимоги до програмного забезпечення, а також виявлені дефекти. Розкрито питання встановлення та налаштування програмного забезпечення на хмарній

платформі, а також вказані вимоги, дотримання яких необхідно для користування сервісом,описана інструкція користувача для роботи із системою.

.

РОЗДІЛ 3. ОХОРОНА ПРАЦІ ТА БЕЗПЕКА У НАДЗВИЧАЙНИХ СИТУАЦІЯХ

3.1 Основні принципи конструювання робочого місця користувача ЕОМ.

Ергономіка (від грецьк. ἔργον наука про пристосування посадових– у традиційному розумінні –роботи») обов‘язків, робочих місць, обладнання та комп‘ютерних програм задля створення найбільш безпечних та ефективних умов праці для людини, виходячи з фізичних і психічних особливостей людського організму.

Більш широке визначення ергономіки, яке було прийняте в 2010 році Міжнародною асоціацією ергономіки (IEA) (Міжнародною ергономічною це наукова дисципліна, що вивчає–асоціацією), звучить так: «Ергономіка взаємодію людини та інших елементів системи, а також сфера діяльності щодо застосування теорії, принципів, даних і методів цієї науки для забезпечення благополуччя людини та оптимізації загальної продуктивності системи».

З цього визначення випливають такі головні завдання ергономіки:

1. Проведення досліджень, спрямованих на пристосування елементів системи "людина – трудовий процес" до природних фізичних і психічних можливостей працівника.

2. Прагнення до забезпечення таким шляхом умов для максимальної ефективності праці.

3. Прагнення запобігти всім можливим загрозам для здоров‘я працівника.

4. Прагнення до оптимальної витрати біологічних ресурсів у процесі праці.

Загальні ергономічні вимоги для організації робочого місця користувача ПЕОМ (ГОСТ 12.2.049-80, ГОСТ 122032-78, ГОСТ 22269-76). Ці вимоги

встановлюють основні параметри робочого місця, оснащеного дисплеєм, і враховують особливість виконуваних робіт.

Параметри робочого місця повинні бути наступними.

Площа кабінету, в якому буде проходити робота повинна бути не менш 6 м², а об'єм не менш 24 м³. Для внутрішньої обробки приміщення повинні використовуватися дифузно-відбивні матеріали з коефіцієнтами відбиття для стелі – 0,7-0,8; для стін – 0,5-0,6; для підлоги – 0,3-0,5.

Конструкція робочого столу повинна забезпечувати оптимальне розміщення на робочій поверхні використовуваного обладнання. Конструкція крісла повинна забезпечувати підтримку раціональної робочої пози під час роботи з відео-дисплейним терміналом (Далі ВДТ) і ПЕОМ, дозволяти змінювати позу з метою зниження статичного напруження м'язів шийно-плечової області і спини для попередження розвитку втоми працюючого (згідно з ГОСТ 12.2.032-78). Поверхня сидіння, спинки та інших елементів стільця (крісла) повинна бути напівм'якою, з покриттям, що не електризується, неслизьке та повітронепроникне, що забезпечує легке очищення від забруднення.

Висота робочої поверхні столу, за відсутності можливості її регулювання повинна складати 725 мм. Робочий стіл повинен мати простір для ніг висотою не менше 600 мм, шириною – не менше 500 мм, не менше 450 мм в глибину на рівні колін і на рівні простягнутої ноги – не менше 650 мм. Робоче місце має бути обладнане підставкою для ніг, має ширину не менше 300 мм, глибину не менше 400 мм, регулювання по висоті в межах 150 мм за кутом нахилу опорної поверхні підставки до 20 градусів.

Відстань від очей користувача до екрану дисплея має становити 500-700 мм. Кут зору 10-20°, але не більше 40°; кут між верхнім краєм дисплея і рівнем очей користувача має становити не менше 10°. Кращим є розташування екрану перпендикулярно до лінії зору користувача.

Робочі місця по відношенню до світлових прорізів повинні розташовуватися не ближче 3 м так, щоб природне світло падало збоку,

переважно зліва. Освітленість також впливає на стан здоров'я і працездатність людини. У відповідності зі СНіП 11-4-79 встановлені наступні вимоги до освітленості:

Для штучного освітлення:

- Комбіноване освітлення – освітленість 1500 лк;
- Загальне освітлення – освітленість 400 лк.

Для природного освітлення:

- Верхнє або комбіноване освітлення – коефіцієнт природної освітленості (далі КПО) 10%;
- Бічне освітлення – КПО 3.5%.

Для суміщеного освітлення:

- Верхнє або комбіноване освітлення – КПО 3-6%;
- Бічне освітлення – КПО 1.1-2%.

До основних показників, що визначають умови здорової роботи, належать: фон, контраст об'єкта з фоном, видимість, показник осліпленості, коефіцієнт пульсації освітленості.

Фон характеризується коефіцієнтом відбиття. Контраст об'єкта з фоном (К) характеризується співвідношенням яскравості розглянутого об'єкта (точки, лінії, знаки) і фону. Оскільки роботи користувача ПЕОМ відносяться до категорії 1а – легкі фізичні роботи (роботи проводяться сидячи і супроводжуються незначним фізичним напруженням, з енерговитратами до 120 ккал / годину), необхідно дотримуватися наступних норм: коефіцієнт відображення більше 0,4, тобто світлий фон; контраст об'єкта з фоном великий і середній при К більше 0,2 (згідно СНіП 11-4-79).

У полі зору користувача ПЕОМ має бути забезпечений відповідний розподіл яскравості. Відношення яскравості екрана до яскравості оточуючих його поверхонь не повинно перевищувати у робочій зоні 3:1 (СНіП 11-4-79). У зв'язку з цим дисплей ПЕОМ повинен відповідати наступним вимогам:

- Яскравість свічення екрану не менше 100 кд/м;

- Мінімальний розмір світної точки для кольорового дисплея не більше 0,6 мм ;
- Контрастність зображення знаку – не менше 0,8;
- Низькочастотне тремтіння зображення в діапазоні 0,05-1,0 Гц повинно знаходитися в межах 0,1 мм;
- Екран повинен мати покриття антивідблиску;
- Відеомонітор повинен бути обладнаний поворотним майданчиком, що дозволяє переміщати відеотермінал в горизонтальній і вертикальній площинах в межах 130-220 мм і змінювати кут нахилу на 10-15 мм.

Коефіцієнт відбиття світла матеріалами і обладнанням всередині приміщень має велике значення для освітлення: чим більше світла відбивається від поверхонь, тим вище освітленість. Коефіцієнт відображення відповідно повинен бути для: стелі 60-70%, стін 40-50%, підлоги 30%, для інших поверхонь 30-40%.

Результати досліджень показують, що найбільшою мірою негативний фізіологічний вплив на операторів ПК пов'язаний з дискомфортними зоровими умовами через неправильно спроектоване освітлення. Згідно СНіП II-4-79 освітленість на горизонтальній площині робочого місця оператора ЕОМ повинна складати 400 лк при висоті цій площині 0,8 м над підлогою.

3.2 Забезпечення захисту працівників суб’єкта господарювання від іонізуючих випромінювань

Іонізуюче випромінювання або радіоактивність є небезпечним явищем для людського організму. При взаємодії впливу іонізаційних випромінювань у навколишнє середовище можуть відбутись різні утворення зарядів . Існують два різновиди випромінювання – «альфа» та «бета».

В залежності від носія та енергії, вони мають різну проникаючу здатність. Альфа це випромінювання яке проявляється важкими частинами складеними з протонів і нейтронів.

В свою чергу бета випромінювання являє собою ланцюг електронів та позитронів які є більшу здатність проникати у середовище. Працюючи на таких територіях, де існує радіаційна атмосфера можуть виникнути різні випадки.

На підприємстві можуть виникнути інциденти при користуванні ядерними матеріалами , зберіганні радіоактивних відходів в наслідок чого працівники можуть отримати травму у вигляді дози опромінення, використання іонізуючих джерел випромінювання.

Також у випадку такої радіаційної аварії забруднюється навколишнє середовище, люди можуть отримати травму у вигляді потужної дози опромінення. Призвести аварію на підприємстві може також якщо активна реакційна речовина знаходиться у роботі та це відбувається незаконно.

Це може привезти до опромінення жителів та перевищити межу дози опромінення. Частинки з цього випромінювання можуть залишати сліди на дихальній системі на травній системі людського організму. Також ці елементи можуть бути у водних каналах, які постачають питну воду людям.

На підприємстві де проводяться роботи з радіаційними речовинами обов’язково мають вживатись заходи проти радіації. Протирадіаційні захисти це така система правових, організаційних норм та санітарної гігієни.

До переліку таких захистів можна включити медичні заходи для забезпечення радіаційної безпеки персоналу та проектно-конструкторські. Для організації заходів проти іонізації опромінювання підприємство має ввести обов’язкові методи щоб подбати про безпеку працюючого персоналу. До таких методів можуть належати заходи які обмежують допуск працівників до джерел які випромінюють радіацію.

До таких працівників можемо віднести таких, які не підходять за віком, за статтю та працівники які вже отримали дозу випромінення. Підприємство мусить створити сприятливі умови що дотримуються встановлених норм та вимог для працівників та застосовувати індивідуальні засоби для захисту працівника цього підприємства.

Організація повинна контролювати рівні опромінювання та вести інформаційну систему про стан радіації на підприємстві та призначених місць для праці.

На підприємстві повинні бути проведені заходи щодо організації безпеки для робіт які проводяться у радіаційних ділянках а саме: -організація роботи нарядів та розпоряджень; -організація та перевірка пропусків до робочих місць; -оформлення контролю за процесом виконання роботи; -введення примусового часу на перерву та вчасне закінчення робочого процесу.

Реалізувати заходи проти радіації за певний відрізок часу можливо, тим що працівники , які працюють з іонізованими випромінюваннями можуть виконувати вчасно свою роботу ,відповідно керівництво може за якісну роботу зменшити кількість робочих днів у тижні.

Цим самим вони застереженням вони зменшать знаходження працівників у зоні випромінювання та відповідно буде менше контактування з радіаційними приладами. Захистити працівників за допомогою відстані підприємство може шляхом доцільного розміщення приміщення, правильно розставити та розрахувати робочі місця для працівників а також забезпечити приладами, які зможуть контактувати, керувати робочим процесом з технікою яка має радіаційний вплив на відстані.

Слугувати захистом може покриття свинцем меблів які присутні у приміщенні (двері, вікна, робочі столи), створення перекриття між поверхами та перегородки. Працівникам обов’язково має бути виданий спеціальний одяг ,такі як фартухи, шапочки та рукавиці зшиті з просвинцевої тканини.

Розміщення робочих місць повинно мати правильний розрахунок на загальну кімнату, не робити перенабір та забезпечити відповідним та необхідним обладнанням робочі кабінети. При користуванні відкритими приладами іонізованого опромінення провести герметизації цих систем, при можливості використовувати роботу техніки. Підприємство повинне вжити усіх санітарно-гігієнічних заходів та соціальних, а також важливо необхідний є медичний захист робочих на об’єкті.

3.3 Висновок до третього розділу

В даному розділі описано основні принципи конструювання робочого місця користувача ЕОМ, зазначено діючі вимоги щодо ергономіки робочого місця. А також визначені заходи та методи із забезпечення радіаційних впливів та іонізації опромінювання на підприємствах. Описані вимоги для керівництва та підлеглих працюючих на об’єктах щодо їхніх дій в разі виникнення радіації .

ВИСНОВКИ

В ході проведеної роботи були вирішені наступні завдання.

1. Проаналізовано відомі рішення і методології в області хмарного тестування.

2. Визначено вимоги до хмарного сервісу для тестування мікросервісних додатків.

3. Розроблено архітектуру хмарного сервісу для тестування мікросервісних додатків, і на основі неї описані деталі реалізації окремих модулів розроблюваного рішення.

4. Реалізовано хмарний сервіс для тестування мікросервісних додатків.

5. Протестовано розроблений додаток.

Подальшим напрямком розвитку буде впровадження даної системи і розробка нових методів тестування, що дозволяють в подальшому проводити хмарне тестування не тільки мікросервісних, а також десктопних, мобільних і інших додатків.

ПЕРЕЛІК ДЖЕРЕЛ

1. Angular Development with TypeScript 2nd Edition/ Yakov Fain. Manning; 2nd edition 2017. 560p.

2. Angular: Up and Running: Learning Angular/ Seshadri S.. Step by Step. O'Reilly Media, 2018. 312p.

3. Clean Code: A Handbook of Agile Software Craftsmanship/Robert C. Martin. Pearson 2008. 464p.

4. Effective Java, 3rd Edition/ Joshua Bloch. Upper Saddle River, NJ : AddisonWesley, 2017. 416p.

5. Fowler M. Monolith First / Martin Fowler [Електронний ресурс] – режим доступу: https://martinfowler.com/bliki/MonolithFirst.html#footnotetypical-monolith.

6. Hands-On Software Architecture with Java: Learn key architectural techniques and strategies to design efficient and elegant Java applications/ Giuseppe Bonocore. Packt Publishing 2021. 510p.

7. Introduction to Algorithms, fourth edition 3th Edition/ Thomas H. Cormen. MIT Press 2009. 1292p.

8. Java Design Patterns: A Hands-On Experience with Real-World Examples 2nd ed. Edition/ Vaskaran Sarcar. Apress 2018. 533p.

9. Java: The Complete Reference, Eleventh Edition/Herbert Schildt. McGraw Hill Education 2018. 1248p.

10. Modern Java in Action: Lambdas, streams, functional and reactive programming 2nd Edition/ Raoul-Gabriel Urma, Mario Fusco, Alan Mycroft. Manning – 2018. 592p.

11. Practical PostgreSQL/Joshua D. Drake, John C. Worsley. O'Reilly Media 2002. 640p.

12. PostgreSQL: Documentation: 14: Index Types [Електронний ресурс]. — режим доступу: https://www.postgresql.org/docs/14/functions.html 60

13. RESTful Java Web Services: A pragmatic guide to designing and building RESTful APIs using Java, 3rd Edition 3rd Revised edition/ Bogunuva Mohanram Balachandar. Packt Publishing 2017. 420p.

14. Spring in Action, 5th edition/Craig Walls. Manning Publications 2018. 520p.

15. Spring Framework 5.3.20: [Електронний ресурс]. – режим доступу: https://spring.io/projects/spring-framework

16. Spring Security 5.7.1: [Електронний ресурс]. – режим доступу: https://spring.io/projects/spring-security

17. Test Driven Development: By Example 1st Edition/ Kent Beck. Addison-Wesley Professional 2002. 240p.

18. Angular Development with TypeScript 2nd Edition/ Yakov Fain, Anton Moiseev. Manning; 2nd edition 2018. 560p.

19. Angular: Up and Running: Learning Angular, Step by Step 1st Edition/

20. Web Engineering: Modelling and Implementing Web Applications / G. Rossi, P. Oscar, S. Daniel. – New York: Springer-Verlag, 2008. 476p

21. Про затвердження Вимог щодо безпеки та захисту здоров'я працівників під час роботи з екранними пристроями [Електронний ресурс] – Режим доступу до ресурсу: https://zakon.rada.gov.ua/laws/show/z0508-18#Text.

22.. Державні санітарні правила і норми роботи з візуальними дисплейними терміналами електронно-обчислювальних машин [Електронний ресурс] – Режим доступу до ресурсу: https://zakon.rada.gov.ua/rada/show/v0007282-98#Text.

23. Правові аспекти охорони праці [Електронний ресурс] – Режим доступу до ресурсу: https://minjust.gov.ua/m/str_3107.

24. Законодавство України про охорону праці. Законодавча та нормативна база України про охорону праці; стандартизація в галузі охорони праці; основні положення Конституції України, законів України «Про охорону праці», «Про пожежну безпеку», Кодексу законів про працю в Україні [Електронний ресурс] – Режим доступу до ресурсу: https://studfile.net/preview/5242039/.

25. 1. Кулаков М. А. Цивільна оборона : навч. посіб. для студентів ви-щих навчальних закладів / М. А. Кулаков, В. О. Ляпун, В. О. Мягкий та ін..; за ред. проф. В. В. Березуцького – Харків: Факт, 2008. – 312 с.

26. Стеблюк М. І. Цивільна оборона: підручник / М. І. Стеблюк – К.: Знання, 2006. – 487 с.

27. Депутат О. П. Цивільна оборона : навч. посіб. / О. П. Депутат, І. В. Коваленко, І. С. Мужик; за ред. полк. В. С. Франчука – Львів: Афіша, 2000. – 336 с.

28. Методичні вказівки до виконання самостійної та практичної роботи «Визначення осередків ураження у надзвичайних ситуаціях» з курсу «Цивільний захист» для студентів усіх спеціальностей та форм навчання з курсу «Цивільний захист» для студентів усіх спеціальностей та форм навчання. / Уклад. Г. Ю. Бахарєва, О. В. Толстоусова, Н. О. Букатенко, І. В. Гуренко– Х.: НТУ «ХПІ», 2015. – 12 с.

29. Закон України Про охорону праці (Відомості Верховної Ради України (ВВР), 1992, № 49, ст.668)

30. Закон України «Про пожежну безпеку» введено в дію з дня опублікування — 29 січня 1994 року згідно з Постановою Верховної Ради України від 17 грудня 1993 року № 3747-XII.

ДОДАТКИ

ДОДАТОК А

ЛІСТИНГ ВСТАНОВЛЕННЯ ЗВ'ЯЗКУ МІЖ POSTGRESQL ТА RUBYON RAILS

```
classPostgreSQLAd
apter<
AbstractAdapter
                    ADAPTER_NAME="PostgreSQL".freeze


                    NATIVE_DATABASE_TYPES= {
                    primary_key:"bigserial primary key",
                    string:      { name:"character varying" },
                    text:        { name:"text" },
                    integer:     { name:"integer", limit:4 },
                    float:       { name:"float" },
                    decimal:     { name:"decimal" },
                    datetime:    { name:"timestamp" },
                    time:        { name:"time" },
                    date:        { name:"date" },
                    daterange:   { name:"daterange" },
                    numrange:    { name:"numrange" },
                    tsrange:     { name:"tsrange" },
                    tstzrange:   { name:"tstzrange" },
                    int4range:   { name:"int4range" },
                    int8range:   { name:"int8range" },
                    binary:      { name:"bytea" },
                    boolean:     { name:"boolean" },
                    xml:         { name:"xml" },
                    tsvector:    { name:"tsvector" },
                    hstore:      { name:"hstore" },
                    inet:        { name:"inet" },
                    cidr:        { name:"cidr" },
                    macaddr:     { name:"macaddr" },
                    uuid:        { name:"uuid" },
                    json:        { name:"json" },
                    jsonb:       { name:"jsonb" },
                    ltree:       { name:"ltree" },
                    citext:      { name:"citext" },
                    point:       { name:"point" },
                    line:        { name:"line" },
                    lseg:        { name:"lseg" },
                    box:         { name:"box" },
                    path:        { name:"path" },
                    polygon:     { name:"polygon" },
```

```
circle:      { name:"circle" },
bit:         { name:"bit" },
bit_varying: { name:"bit varying" },
money:       { name:"money" },
interval:    { name:"interval" },
oid:         { name:"oid" },
      }


OID=PostgreSQL::OID#:nodoc:


includePostgreSQL::Quoting
includePostgreSQL::ReferentialIntegrity
includePostgreSQL::SchemaStatements
includePostgreSQL::DatabaseStatements


defsupports_bulk_alter?
true
end


defsupports_index_sort_order?
true
end


defsupports_partial_index?
true
end


defsupports_expression_index?
true
end


defsupports_transaction_isolation?
true
end


defsupports_foreign_keys?
true
end
```

```
defsupports_validate_constraints?
true
end


defsupports_views?
true
end


defsupports_datetime_with_precision?
true
end


defsupports_json?
        postgresql_version >=90200
end


defsupports_comments?
true
end


defsupports_savepoints?
true
end


defindex_algorithms
        { concurrently:"CONCURRENTLY" }
end


classStatementPool< ConnectionAdapters::StatementPool# :nodoc:
definitialize(connection, max)
super(max)
@connection= connection
@counter=0
end


defnext_key
"a#{@counter+1}"
end
```

```
def[]=(sql, key)
super.tap { @counter+=1 }
end


private
defdealloc(key)
@connection.query "DEALLOCATE #{key}"if connection_active?
rescuePG::Error
end


defconnection_active?
@connection.status ==PG::CONNECTION_OK
rescuePG::Error
false
end
end


# Initializes and connects a PostgreSQL adapter.
definitialize(connection, logger, connection_parameters, config)
super(connection, logger, config)


@connection_parameters= connection_parameters


# @local_tz is initialized as nil to avoid warnings when connect tries
to use it
@local_tz=nil
@max_identifier_length=nil


        connect
        add_pg_encoders
@statements=StatementPool.new@connection,
self.class.type_cast_config_to_integer(config[:statement_limit])


if postgresql_version <90100
raise"Your version of PostgreSQL (#{postgresql_version}) is too old.
Active Record supports PostgreSQL >= 9.1."
end


        add_pg_decoders
```

```
@type_map=Type::HashLookupTypeMap.new
        initialize_type_map
@local_tz= execute("SHOW TIME ZONE", "SCHEMA").first["TimeZone"]
@use_insert_returning=@config.key?(:insert_returning)
?self.class.type_cast_config_to_boolean(@config[:insert_returning]) :
true
end


# Clears the prepared statements cache.
defclear_cache!
@lock.synchronize do
@statements.clear
end
end


deftruncate(table_name, name=nil)
        exec_query "TRUNCATE TABLE #{quote_table_name(table_name)}",
name, []
end


# Is this connection alive and ready for queries?
defactive?
@lock.synchronize do
@connection.query "SELECT 1"
end
true
rescuePG::Error
false
end


# Close then reopen the connection.
defreconnect!
@lock.synchronize do
super
@connection.reset
          configure_connection
end
end


defreset!
@lock.synchronize do
          clear_cache!
```

```
          reset_transaction
unless@connection.transaction_status == ::PG::PQTRANS_IDLE
@connection.query "ROLLBACK"
end
@connection.query "DISCARD ALL"
          configure_connection
end
end


# Disconnects from the database if already connected. Otherwise, this
# method does nothing.
defdisconnect!
@lock.synchronize do
super
@connection.close rescuenil
end
end


defdiscard!# :nodoc:
@connection.socket_io.reopen(IO::NULL) rescuenil
@connection=nil
end


defnative_database_types#:nodoc:
NATIVE_DATABASE_TYPES
end


defset_standard_conforming_strings
        execute("SET standard_conforming_strings = on", "SCHEMA")
end


defsupports_ddl_transactions?
true
end


defsupports_advisory_locks?
true
end


defsupports_explain?
true
```

```
end


defsupports_extensions?
true
end


defsupports_ranges?
# Range datatypes weren't introduced until PostgreSQL 9.2
        postgresql_version >=90200
end


defsupports_materialized_views?
        postgresql_version >=90300
end


defsupports_foreign_tables?
        postgresql_version >=90300
end


defsupports_pgcrypto_uuid?
        postgresql_version >=90400
end


defget_advisory_lock(lock_id) # :nodoc:
unless lock_id.is_a?(Integer) && lock_id.bit_length <=63
raise(ArgumentError, "PostgreSQL requires advisory lock ids to be a
signed 64 bit integer")
end
        query_value("SELECT pg_try_advisory_lock(#{lock_id})")
end


defrelease_advisory_lock(lock_id) # :nodoc:
unless lock_id.is_a?(Integer) && lock_id.bit_length <=63
raise(ArgumentError, "PostgreSQL requires advisory lock ids to be a
signed 64 bit integer")
end
        query_value("SELECT pg_advisory_unlock(#{lock_id})")
end


defenable_extension(name)
```

```
        exec_query("CREATE EXTENSION IF NOT EXISTS \"#{name}\"").tap {
          reload_type_map
        }
end


defdisable_extension(name)
        exec_query("DROP EXTENSION IF EXISTS \"#{name}\" CASCADE").tap
{
          reload_type_map
        }
end


defextension_enabled?(name)
        res = exec_query("SELECT EXISTS(SELECT * FROM
pg_available_extensions WHERE name = '#{name}' AND installed_version IS
NOT NULL) as enabled", "SCHEMA")
        res.cast_values.first
end


defextensions
        exec_query("SELECT extname FROM pg_extension",
"SCHEMA").cast_values
end


# Returns the configured supported identifier length supported by
PostgreSQL
defmax_identifier_length
@max_identifier_length||= query_value("SHOW max_identifier_length",
"SCHEMA").to_i
end
alias table_alias_length max_identifier_length
alias index_name_length max_identifier_length


# Set the authorized user for this session
defsession_auth=(user)
        clear_cache!
        execute("SET SESSION AUTHORIZATION #{user}")
end


defuse_insert_returning?
@use_insert_returning
end
```

```
defcolumn_name_for_operation(operation, node) # :nodoc:
OPERATION_ALIASES.fetch(operation) { operation.downcase }
end


OPERATION_ALIASES= { # :nodoc:
"maximum" =>"max",
"minimum" =>"min",
"average" =>"avg",
      }


# Returns the version of the connected PostgreSQL server.
defpostgresql_version
@connection.server_version
end


defdefault_index_type?(index) # :nodoc:
        index.using==:btree||super
end


private
# See https://www.postgresql.org/docs/current/static/errcodes-
appendix.html
VALUE_LIMIT_VIOLATION="22001"
NUMERIC_VALUE_OUT_OF_RANGE="22003"
NOT_NULL_VIOLATION="23502"
FOREIGN_KEY_VIOLATION="23503"
UNIQUE_VIOLATION="23505"
SERIALIZATION_FAILURE="40001"
DEADLOCK_DETECTED="40P01"
LOCK_NOT_AVAILABLE="55P03"
QUERY_CANCELED="57014"


deftranslate_exception(exception, message)
return exception unless exception.respond_to?(:result)


case exception.result.try(:error_field, PG::PG_DIAG_SQLSTATE)
whenUNIQUE_VIOLATION
RecordNotUnique.new(message)
whenFOREIGN_KEY_VIOLATION
InvalidForeignKey.new(message)
```

```
whenVALUE_LIMIT_VIOLATION
ValueTooLong.new(message)
whenNUMERIC_VALUE_OUT_OF_RANGE
RangeError.new(message)
whenNOT_NULL_VIOLATION
NotNullViolation.new(message)
whenSERIALIZATION_FAILURE
SerializationFailure.new(message)
whenDEADLOCK_DETECTED
Deadlocked.new(message)
whenLOCK_NOT_AVAILABLE
LockWaitTimeout.new(message)
whenQUERY_CANCELED
QueryCanceled.new(message)
else
super
end
end


defget_oid_type(oid, fmod, column_name, sql_type="".freeze)
if!type_map.key?(oid)
            load_additional_types([oid])
end


          type_map.fetch(oid, fmod, sql_type) {
warn"unknown OID #{oid}: failed to recognize type of '#{column_name}'.
It will be treated as String."
Type.default_value.tap do |cast_type|
              type_map.register_type(oid, cast_type)
end
          }
end


definitialize_type_map(m= type_map)
          m.register_type "int2", Type::Integer.new(limit:2)
          m.register_type "int4", Type::Integer.new(limit:4)
          m.register_type "int8", Type::Integer.new(limit:8)
          m.register_type "oid", OID::Oid.new
          m.register_type "float4", Type::Float.new
          m.alias_type "float8", "float4"
          m.register_type "text", Type::Text.new
          register_class_with_limit m, "varchar", Type::String
          m.alias_type "char", "varchar"
          m.alias_type "name", "varchar"
          m.alias_type "bpchar", "varchar"
```

```
            m.register_type "bool", Type::Boolean.new
            register_class_with_limit m, "bit", OID::Bit
            register_class_with_limit m, "varbit", OID::BitVarying
            m.alias_type "timestamptz", "timestamp"
            m.register_type "date", OID::Date.new


            m.register_type "money", OID::Money.new
            m.register_type "bytea", OID::Bytea.new
            m.register_type "point", OID::Point.new
            m.register_type "hstore", OID::Hstore.new
            m.register_type "json", Type::Json.new
            m.register_type "jsonb", OID::Jsonb.new
            m.register_type "cidr", OID::Cidr.new
            m.register_type "inet", OID::Inet.new
            m.register_type "uuid", OID::Uuid.new
            m.register_type "xml", OID::Xml.new
            m.register_type "tsvector",
OID::SpecializedString.new(:tsvector)
            m.register_type "macaddr",
OID::SpecializedString.new(:macaddr)
            m.register_type "citext", OID::SpecializedString.new(:citext)
            m.register_type "ltree", OID::SpecializedString.new(:ltree)
            m.register_type "line", OID::SpecializedString.new(:line)
            m.register_type "lseg", OID::SpecializedString.new(:lseg)
            m.register_type "box", OID::SpecializedString.new(:box)
            m.register_type "path", OID::SpecializedString.new(:path)
            m.register_type "polygon",
OID::SpecializedString.new(:polygon)
            m.register_type "circle", OID::SpecializedString.new(:circle)


            m.register_type "interval"do |_, _, sql_type|
              precision = extract_precision(sql_type)
OID::SpecializedString.new(:interval, precision: precision)
end


            register_class_with_precision m, "time", Type::Time
            register_class_with_precision m, "timestamp", OID::DateTime


            m.register_type "numeric"do |_, fmod, sql_type|
              precision = extract_precision(sql_type)
              scale = extract_scale(sql_type)


# The type for the numeric depends on the width of the field,
```

```
# so we'll do something special here.
#
# When dealing with decimal columns:
#
# places after decimal  = fmod - 4 & 0xffff
# places before decimal = (fmod - 4) >> 16 & 0xffff
if fmod && (fmod -4&0xffff).zero?
# FIXME: Remove this class, and the second argument to
# lookups on PG
Type::DecimalWithoutScale.new(precision: precision)
else
OID::Decimal.new(precision: precision, scale: scale)
end
end


          load_additional_types
end


# Extracts the value from a PostgreSQL column default definition.
defextract_value_from_default(default)
case default
# Quoted types
when/\A[\(B]?'(.*)'.*::"?([\w. ]+)"?(?:\[\])?\z/m
# The default 'now'::date is CURRENT_DATE
if$1=="now".freeze &&$2=="date".freeze
nil
else
$1.gsub("''".freeze, "'".freeze)
end
# Boolean types
when"true".freeze, "false".freeze
            default
# Numeric types
when/\A\(?(-?\d+(\.\d*)?)\)?(::bigint)?\z/
$1
# Object identifier types
when/\A-?\d+\z/
$1
else
# Anything else is blank, some user type, or some function
# and we can't know the value of that, so return nil.
nil
end
end
```

```
defextract_default_function(default_value, default)
          default if has_default_function?(default_value, default)
end


defhas_default_function?(default_value, default)
!default_value
&&%r{\w+\(.*\)|\(.*\)::\w+|CURRENT_DATE|CURRENT_TIMESTAMP}.match?(defau
lt)
end


defload_additional_types(oids=nil)
          initializer =OID::TypeMapInitializer.new(type_map)


if supports_ranges?
            query =<<-SQL
SELECTt.oid, t.typname, t.typelem, t.typdelim, t.typinput,
r.rngsubtype, t.typtype, t.typbasetype
FROM pg_type as t
LEFT JOIN pg_range as r ONoid= rngtypid
            SQL
else
            query =<<-SQL
SELECTt.oid, t.typname, t.typelem, t.typdelim, t.typinput, t.typtype,
t.typbasetype
FROM pg_type as t
            SQL
end


if oids
            query +="WHERE t.oid::integer IN (%s)"% oids.join(", ")
else
            query += initializer.query_conditions_for_initial_load
end


          execute_and_clear(query, "SCHEMA", []) do |records|
            initializer.run(records)
end
end


FEATURE_NOT_SUPPORTED="0A000"#:nodoc:
```

```
defexecute_and_clear(sql, name, binds, prepare:false)
if without_prepared_statement?(binds)
            result = exec_no_cache(sql, name, [])
elsif!prepare
            result = exec_no_cache(sql, name, binds)
else
            result = exec_cache(sql, name, binds)
end
          ret =yield result
          result.clear
          ret
end


defexec_no_cache(sql, name, binds)
          type_casted_binds = type_casted_binds(binds)
          log(sql, name, binds, type_casted_binds) do
ActiveSupport::Dependencies.interlock.permit_concurrent_loads do
@connection.async_exec(sql, type_casted_binds)
end
end
end


defexec_cache(sql, name, binds)
          stmt_key = prepare_statement(sql)
          type_casted_binds = type_casted_binds(binds)


          log(sql, name, binds, type_casted_binds, stmt_key) do
ActiveSupport::Dependencies.interlock.permit_concurrent_loads do
@connection.exec_prepared(stmt_key, type_casted_binds)
end
end
rescueActiveRecord::StatementInvalid => e
raiseunless is_cached_plan_failure?(e)


# Nothing we can do if we are in a transaction because all commands
# will raise InFailedSQLTransaction
if in_transaction?
raiseActiveRecord::PreparedStatementCacheExpired.new(e.cause.message)
else
@lock.synchronize do
# outside of transactions we can simply flush this query and retry
@statements.delete sql_key(sql)
end
retry
```

```
end
end


# Annoyingly, the code for prepared statements whose return value may
# have changed is FEATURE_NOT_SUPPORTED.
#
# This covers various different error types so we need to do additional
# work to classify the exception definitively as a
# ActiveRecord::PreparedStatementCacheExpired
#
# Check here for more details:
#
https://git.postgresql.org/gitweb/?p=postgresql.git;a=blob;f=src/backen
d/utils/cache/plancache.c#l573
CACHED_PLAN_HEURISTIC="cached plan must not change result type".freeze
defis_cached_plan_failure?(e)
          pgerror = e.cause
          code =
pgerror.result.result_error_field(PG::PG_DIAG_SQLSTATE)
          code ==FEATURE_NOT_SUPPORTED&&
pgerror.message.include?(CACHED_PLAN_HEURISTIC)
rescue
false
end


defin_transaction?
          open_transactions >0
end


# Returns the statement identifier for the client side cache
# of statements
defsql_key(sql)
"#{schema_search_path}-#{sql}"
end


# Prepare the statement if it hasn't been prepared, return
# the statement key.
defprepare_statement(sql)
@lock.synchronize do
            sql_key = sql_key(sql)
unless@statements.key? sql_key
              nextkey =@statements.next_key
begin
@connection.prepare nextkey, sql
```

```
rescue => e
raise translate_exception_class(e, sql)
end
# Clear the queue
@connection.get_last_result
@statements[sql_key] = nextkey
end
@statements[sql_key]
end
end


# Connects to a PostgreSQL server and sets up the adapter depending on
the
# connected server's characteristics.
defconnect
@connection=PG.connect(@connection_parameters)
          configure_connection
rescue ::PG::Error => error
if error.message.include?("does not exist")
raiseActiveRecord::NoDatabaseError
else
raise
end
end


# Configures the encoding, verbosity, schema search path, and time zone
of the connection.
# This is called by #connect and should not be called manually.
defconfigure_connection
if@config[:encoding]
@connection.set_client_encoding(@config[:encoding])
end
self.client_min_messages =@config[:min_messages] ||"warning"
self.schema_search_path =@config[:schema_search_path]
||@config[:schema_order]


# Use standard-conforming strings so we don't have to do the E'...'
dance.
          set_standard_conforming_strings


          variables =@config.fetch(:variables, {}).stringify_keys


# If using Active Record's time zone support configure the connection
```

```
to return
# TIMESTAMP WITH ZONE types in UTC.
unless variables["timezone"]
ifActiveRecord::Base.default_timezone ==:utc
              variables["timezone"] ="UTC"
elsif@local_tz
              variables["timezone"] =@local_tz
end
end


# SET statements from :variables config hash
# https://www.postgresql.org/docs/current/static/sql-set.html
          variables.map do |k, v|
if v ==":default"|| v ==:default
# Sets the value to the global or compile default
              execute("SET SESSION #{k} TO DEFAULT", "SCHEMA")
elsif!v.nil?
              execute("SET SESSION #{k} TO #{quote(v)}", "SCHEMA")
end
end
end


# Returns the list of a table's column names, data types, and default
values.
#
# The underlying query is roughly:
#  SELECT column.name, column.type, default.value, column.comment
#    FROM column LEFT JOIN default
#      ON column.table_id = default.table_id
#     AND column.num = default.column_num
#   WHERE column.table_id = get_table_id('table_name')
#     AND column.num > 0
#     AND NOT column.is_dropped
#   ORDER BY column.num
#
# If the table name is not prefixed with a schema, the database will
# take the first match from the schema search path.
#
# Query implementation notes:
#  - format_type includes the column size constraint, e.g. varchar(50)
#  - ::regclass is a function that gives the id for a table name
defcolumn_definitions(table_name)
          query(<<-end_sql, "SCHEMA")
              SELECT a.attname, format_type(a.atttypid, a.atttypmod),
                     pg_get_expr(d.adbin, d.adrelid), a.attnotnull,
a.atttypid, a.atttypmod,
```

```
                    c.collname, col_description(a.attrelid, a.attnum)
AS comment
                FROM pg_attribute a
                LEFT JOIN pg_attrdef d ON a.attrelid = d.adrelid AND
a.attnum = d.adnum
                LEFT JOIN pg_type t ON a.atttypid = t.oid
                LEFT JOIN pg_collation c ON a.attcollation = c.oid AND
a.attcollation <> t.typcollation
               WHERE a.attrelid =
#{quote(quote_table_name(table_name))}::regclass
                 AND a.attnum > 0 AND NOT a.attisdropped
               ORDER BY a.attnum
          end_sql
end


defextract_table_ref_from_insert_sql(sql)
          sql[/into\s("[A-Za-z0-9_."\[\]\s]+"|[A-Za-z0-
9_."\[\]]+)\s*/im]
$1.strip if$1
end


defarel_visitor
Arel::Visitors::PostgreSQL.new(self)
end


defcan_perform_case_insensitive_comparison_for?(column)
@case_insensitive_cache||= {}
@case_insensitive_cache[column.sql_type] ||=begin
            sql =<<-end_sql
              SELECT exists(
                SELECT * FROM pg_proc
                WHERE proname = 'lower'
                  AND proargtypes = ARRAY[#{quote
column.sql_type}::regtype]::oidvector
              ) OR exists(
                SELECT * FROM pg_proc
                INNER JOIN pg_cast
                  ON ARRAY[casttarget]::oidvector = proargtypes
                WHERE proname = 'lower'
                  AND castsource = #{quote column.sql_type}::regtype
              )
            end_sql
            execute_and_clear(sql, "SCHEMA", []) do |result|
              result.getvalue(0, 0)
end
```

```
end
end


defadd_pg_encoders
          map =PG::TypeMapByClass.new
          map[Integer] =PG::TextEncoder::Integer.new
          map[TrueClass] =PG::TextEncoder::Boolean.new
          map[FalseClass] =PG::TextEncoder::Boolean.new
@connection.type_map_for_queries = map
end


defadd_pg_decoders
          coders_by_name = {
"int2" =>PG::TextDecoder::Integer,
"int4" =>PG::TextDecoder::Integer,
"int8" =>PG::TextDecoder::Integer,
"oid" =>PG::TextDecoder::Integer,
"float4" =>PG::TextDecoder::Float,
"float8" =>PG::TextDecoder::Float,
"bool" =>PG::TextDecoder::Boolean,
          }
          known_coder_types = coders_by_name.keys.map { |n| quote(n) }
          query =<<-SQL % known_coder_types.join(", ")
SELECTt.oid, t.typname
FROM pg_type as t
WHEREt.typnameIN (%s)
          SQL
          coders = execute_and_clear(query, "SCHEMA", []) do |result|
            result
              .map { |row| construct_coder(row,
coders_by_name[row["typname"]]) }
              .compact
end


          map =PG::TypeMapByOid.new
          coders.each { |coder| map.add_coder(coder) }
@connection.type_map_for_results = map
end


defconstruct_coder(row, coder_class)
returnunless coder_class
          coder_class.new(oid: row["oid"].to_i, name: row["typname"])
end
```

```
ActiveRecord::Type.add_modifier({ array:true }, OID::Array,
adapter::postgresql)
ActiveRecord::Type.add_modifier({ range:true }, OID::Range,
adapter::postgresql)
ActiveRecord::Type.register(:bit, OID::Bit, adapter::postgresql)
ActiveRecord::Type.register(:bit_varying, OID::BitVarying,
adapter::postgresql)
ActiveRecord::Type.register(:binary, OID::Bytea, adapter::postgresql)
ActiveRecord::Type.register(:cidr, OID::Cidr, adapter::postgresql)
ActiveRecord::Type.register(:date, OID::Date, adapter::postgresql)
ActiveRecord::Type.register(:datetime, OID::DateTime,
adapter::postgresql)
ActiveRecord::Type.register(:decimal, OID::Decimal,
adapter::postgresql)
ActiveRecord::Type.register(:enum, OID::Enum, adapter::postgresql)
ActiveRecord::Type.register(:hstore, OID::Hstore, adapter::postgresql)
ActiveRecord::Type.register(:inet, OID::Inet, adapter::postgresql)
ActiveRecord::Type.register(:jsonb, OID::Jsonb, adapter::postgresql)
ActiveRecord::Type.register(:money, OID::Money, adapter::postgresql)
ActiveRecord::Type.register(:point, OID::Point, adapter::postgresql)
ActiveRecord::Type.register(:legacy_point, OID::LegacyPoint,
adapter::postgresql)
ActiveRecord::Type.register(:uuid, OID::Uuid, adapter::postgresql)
ActiveRecord::Type.register(:vector, OID::Vector, adapter::postgresql)
ActiveRecord::Type.register(:xml, OID::Xml, adapter::postgresql)
end
end
end
```

ДОДАТОК Б

ЛІСТИНГ РЕАЛІЗАЦІЇ ПРОЦЕДУРИ ТЕСТУВАННЯ ХМАРНИХ ДОДАТКІВ

```
npm_version =
version.gsub(/\./).w
ith_index { |s, i| i
>=2?"-" : s }
```

```
(FRAMEWORKS+ ["rails"]).each do |framework|
  namespace framework do
    gem     ="pkg/#{framework}-#{version}.gem"
    gemspec ="#{framework}.gemspec"


    task :cleando
      rm_f gem
end


    task :update_versionsdo
      glob = root.dup
if framework =="rails"
        glob <<"/version.rb"
else
        glob <<"/#{framework}/lib/*"
        glob <<"/gem_version.rb"
end


      file =Dir[glob].first
      ruby =File.read(file)


      major, minor, tiny, pre = version.split(".", 4)
      pre = pre ? pre.inspect : "nil"


      ruby.gsub!(/^(\s*)MAJOR(\s*)= .*?$/, "\\1MAJOR = #{major}")
raise"Could not insert MAJOR in #{file}"unless$1


      ruby.gsub!(/^(\s*)MINOR(\s*)= .*?$/, "\\1MINOR = #{minor}")
raise"Could not insert MINOR in #{file}"unless$1
```

```
      ruby.gsub!(/^(\s*)TINY(\s*)= .*?$/, "\\1TINY  = #{tiny}")
raise"Could not insert TINY in #{file}"unless$1


      ruby.gsub!(/^(\s*)PRE(\s*)= .*?$/, "\\1PRE   = #{pre}")
raise"Could not insert PRE in #{file}"unless$1


File.open(file, "w") { |f| f.write ruby }


require"json"
ifFile.exist?("#{framework}/package.json")
&&JSON.parse(File.read("#{framework}/package.json"))["version"] !=
npm_version
Dir.chdir("#{framework}") do
if sh "which npm"
            sh "npm version #{npm_version} --no-git-tag-version"
else
raise"You must have npm installed to release Rails."
end
end
end
end


    task gem =>%w(update_versions pkg)do
      cmd =""
      cmd +="cd #{framework}&&"unless framework =="rails"
      cmd +="bundle exec rake package &&"unless framework =="rails"
      cmd +="gem build #{gemspec}&& mv #{framework}-#{version}.gem
#{root}/pkg/"
      sh cmd
end


    task build: [:clean, gem]
    task install::builddo
      sh "gem install --pre #{gem}"
end


    task push::builddo
      sh "gem push #{gem}"
```

```
ifFile.exist?("#{framework}/package.json")
Dir.chdir("#{framework}") do
          npm_tag = version =~/[a-z]/?"pre" : "latest"
          sh "npm publish --tag #{npm_tag}"
end
end
end
end
end


namespace :changelogdo
  task :headerdo
    (FRAMEWORKS+ ["guides"]).each do |fw|
require"date"
      fname =File.join fw, "CHANGELOG.md"
      current_contents =File.read(fname)


      header ="## Rails #{version} (#{Date.today.strftime('%B %d, %Y')})
##\n\n"
      header +="*   No changes.\n\n\n"if current_contents =~/\A##/
      contents = header + current_contents
File.open(fname, "wb") { |f| f.write contents }
end
end


  task :release_datedo
    (FRAMEWORKS+ ["guides"]).each do |fw|
require"date"
      replace ="## Rails #{version} (#{Date.today.strftime('%B %d, %Y')})
##\n"
      fname =File.join fw, "CHANGELOG.md"


      contents =File.read(fname).sub(/^(## Rails .*)\n/, replace)
File.open(fname, "wb") { |f| f.write contents }
end
end


  task :release_summarydo
    (FRAMEWORKS+ ["guides"]).each do |fw|
puts"## #{fw}"
      fname    =File.join fw, "CHANGELOG.md"
      contents =File.readlines fname
      contents.shift
```

```
      changes = []
      changes << contents.shift until contents.first =~/^\*Rails
\d+\.\d+\.\d+/
puts changes.reject { |change| change.strip.empty? }.join
puts
end
end
end


namespace :alldo
  task build:FRAMEWORKS.map { |f| "#{f}:build"           } +
["rails:build"]
  task update_versions:FRAMEWORKS.map { |f| "#{f}:update_versions" } +
["rails:update_versions"]
  task install:FRAMEWORKS.map { |f| "#{f}:install"         } +
["rails:install"]
  task push:FRAMEWORKS.map { |f| "#{f}:push"            } +
["rails:push"]


  task :ensure_clean_statedo
unless`git status -s | grep -v
'RAILS_VERSION\\|CHANGELOG\\|Gemfile.lock\\|package.json\\|version.rb\\|t
asks/release.rb'`.strip.empty?
abort"[ABORTING] `git status` reports a dirty tree. Make sure all changes
are committed"
end


unlessENV["SKIP_TAG"] ||`git tag | grep '^#{tag}$'`.strip.empty?
abort"[ABORTING] `git tag` shows that #{tag} already exists. Has this
version already\n"\
"           been released? Git tagging can be skipped by setting
SKIP_TAG=1"
end
end


  task verify::installdo
    app_name ="pkg/verify-#{version}-#{Time.now.to_i}"
    sh "rails _#{version}_ new #{app_name} --skip-bundle"# Generate with
the right version.
    cd app_name


# Replace the generated gemfile entry with the exact version.
File.write("Gemfile", File.read("Gemfile").sub(/^gem 'rails.*/, "gem
```

```
'rails', '#{version}'"))
    sh "bundle"


    sh "rails generate scaffold user name admin:boolean && rails
db:migrate"


puts"Booting a Rails server. Verify the release by:"
puts
puts"- Seeing the correct release number on the root page"
puts"- Viewing /users"
puts"- Creating a user"
puts"- Updating a user (e.g. disable the admin flag)"
puts"- Deleting a user on /users"
puts"- Whatever else you want."
begin
      sh "rails server"
rescueInterrupt
# Server passes along interrupt. Prevent halting verify task.
end
end


  task :bundledo
    sh "bundle check"
end


  task :commitdo
unless`git status -s`.strip.empty?
File.open("pkg/commit_message.txt", "w") do |f|
        f.puts"# Preparing for #{version} release\n"
        f.puts
        f.puts"# UNCOMMENT THE LINE ABOVE TO APPROVE THIS COMMIT"
end


      sh "git add . && git commit --verbose --
template=pkg/commit_message.txt"
      rm_f "pkg/commit_message.txt"
end
end


  task :tagdo
    sh "git tag -s -m '#{tag} release' #{tag}"
    sh "git push --tags"
```

```
end

  task prep_release:%w(ensure_clean_state build bundle commit)

  task release:%w(prep_release tag push)
end

moduleAnnouncement
classVersion
definitialize(version)
@version, @gem_version= version, Gem::Version.new(version)
end

defto_s
@version
end

defprevious
@gem_version.segments[0, 3].tap { |v| v[2] -=1 }.join(".")
end

defmajor_or_security?
@gem_version.segments[2].zero? ||@gem_version.segments[3].is_a?(Integer)
end

defrc?
@version=~/rc/
end
end
end

task :announcedo
Dir.chdir("pkg/") do
    versions =ENV["VERSIONS"] ?ENV["VERSIONS"].split(",") : [ version ]
    versions = versions.sort.map { |v| Announcement::Version.new(v) }

raise"Only valid for patch releases"if
versions.any?(&:major_or_security?)
```

```
if versions.any?(&:rc?)
require"date"
      future_date =Date.today +5
      future_date +=1while future_date.saturday? || future_date.sunday?


      github_user =`git config github.user`.chomp
end


require"erb"
    template =File.read("../tasks/release_announcement_draft.erb")


    match =ERB.version.match(/\Aerb\.rb \[(?<version>[^ ]+) /)
if match && match[:version] >="2.2.0"# Ruby 2.6+
putsERB.new(template, trim_mode:"<>").result(binding)
else
putsERB.new(template, nil, "<>").result(binding)
end
end
end
```