Plag

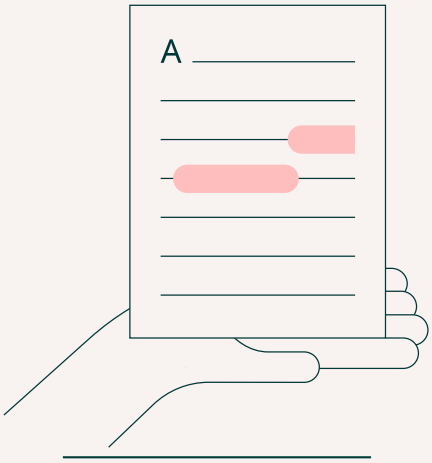
Звіт про оригінальність

● Оцінка схожості %7

● Ризик плагіату СЕРЕДНІЙ

Ігор Кагало 2025-06-10 09:08

Посилання на звіт: 101Gh / Посилання користувача: qfC8



Ось вона –
Ваша звіт про оригінальність!

Ми раді повідомити, що перевірка вашого документа завершена, і результати вже готові! Наші алгоритми старанно працювали, щоб знайти збіги в наших базах даних.

На наступних сторінках ви знайдете результати перевірки:

Бали
Збіги
Посилання

Ваш документ було перевірено за такими джерелами:

- ✓ База даних інтернет-джерел
- ✓ База даних наукових статей
- ○ Глибока перевірка (наш вдосконалений алгоритм)

Бали

Збіги тексту	7%
Перефразування	0%
Цитований текст	0%
Неправильне цитування	0%
Збігів не знайдено	93%

Ризик плагіату СЕРЕДНІЙ

Ризик плагіату вказує, як збіги тексту розподілені по документу. Вищий ризик виникає, коли збіги з'являються близько один до одного, наприклад, у тому самому абзаці або розділі.

Оцінка схожості 7%

Оцінка схожості показує, скільки слів або символів у вашому документі збігаються з текстами інших документів, включаючи перефразовані тексти або неправильні цитати.

Збіги

1 ПРИНЦИПИ ПОБУДОВИ АРХІТЕКТУРИ ВЕБ-ДОДАТКУ

Архітектурні стилі веб-додатку та підходи до їх використання

Архітектура веб-додатку — це спосіб організації внутрішньої структури додатку, що визначає компоненти системи, способи їхньої взаємодії та принципи побудови. Хороша архітектура є не просто шаблоном, який використовується із програми в програму, а комплексний процес, де потрібно не тільки добре знати та розуміти теорію мови програмування, а і досконало розуміти, як в подальшому ця архітектура буде взаємодіяти з іншими технологіями (наприклад, брокерами сповіщень, сторонніми програмними інтерфейсами) та з компонентами своєї програми. Візуалізація переваг та недоліків зображена на рисунку 1.1

Рисунок 1.1 – Візуалізація переваг та недоліків архітектур

У сучасній розробці веб-додатків використовуються кілька основних архітектурних стилів, кожен з яких має свої переваги, недоліки та сфери застосування:

Монолітна архітектура – це архітектура, де вся система збирається в один великий застосунок. В одному виконуваному файлі або процесі знаходиться все: і обробник вхідних запитів, і бізнес-логіка, і доступ до бази даних. З переваг можна виділити наступне:

В першу чергу це простота написання самої програми, адже не потрібно сильно задумуватись, як правильно організовувати комунікацію між сервісами, бо всі сервіси розміщені на одному рівні.

Простота розгортання. В цьому плані монолітна архітектура має перевагу над більшість іншими, адже один програма – це одне середовище, яке з легкістю можна розгорнути на в якомусь ізольованому середовищі.

З недоліків можна виділити наступне:

Складність масштабування. Якщо певна частина системи починає споживати значно

більше ресурсів, неможливо масштабувати тільки її — потрібно запускати додаткові екземпляри всього моноліту, навіть якщо інші частини простоюють.

Складність підтримки в довгостроковій перспективі. Чим більше зростає система, тим важче зберігати чистоту архітектури. Кількість залежностей, побічних ефектів, спільних моделей і конфігурацій зростає, що ускладнює внесення змін. Нерідко змінюючи одну частину, можна зламати іншу, бо вони тісно пов’язані.

Використання монолотної архітектури наведено на рисунку 1.2.

Рисунок 1.2 – Монолітна архітектура

Мікросервісна архітектура – це архітектура, де система складається з 21 набору незалежних сервісів, кожен з яких відповідає за конкретну бізнес-функцію. Кожен сервіс може мати різні технології та навіть нерідко можуть бути написані на різних мовах програмування. З переваг можна виділити наступні пункти:

Архітектура дозволяє індивідуальне масштабування кожного сервісу відповідно до навантаження, що забезпечує раціональне використання обчислювальних ресурсів.

Ізоляція збоїв та підвищення толерантності до відмов- завдяки дистрибутивній природі кожного компонента, відмова одного мікросервісу не призводить до повної деградації системи.

Кожен сервіс може бути реалізований із застосуванням оптимального для задачі стеку технологій (мови програмування, фреймворків, СУБД)

З недоліків можна виділити:

Найбільш виражений недостаток –це складність підтримки зворотної сумісності. Зміни в одному сервісі можуть вплинути на роботу інших. Необхідно забезпечувати сумісність API(інтерфейс програмного забезпечення), підтримку декількох версій та впроваджувати контрактне тестування.

Зростання складності тестування. Повноцінне тестування вимагає наявності або емуляції всіх залежностей. Інтеграційні та контрактні тести потребують додаткових інструментів, ізоляції середовищ та валідації міжсервісної взаємодії.

Нестабільність через мережеву взаємодію. Комунікація між сервісами відбувається по мережі, що створює додаткові точки відмов, потенційну затримку та необхідність робити декіклька запитів, таймаутів та механізмів деградації.

Приклад реалізації мікросервісної архітектури зображений на рисунку 1.3.

Рисунок 1.3 – Приклад реалізації мікросервісної архітектури

Архітектурні рівні та розділення відповідальностей

Надійне програмне забезпечення почало створюватись за загальноприйнятими принципами чистої архітектури, для того щоб забезпечити зрозумілу структуру системи, спростити її розробку та підтримку, а також підвищити гнучкість при зміні вимог або технологій. Є багато принципів, які роблять програмне забезпечення якісним та надійним, один із них це - Separation of Concerns. Розподіл відповідальностей (SoC - separation of concerns) — це принцип у розробці та проектуванні програмного забезпечення, спрямований на розбиття складних систем на більш дрібні, більш керовані частини. Мета полягає в тому, щоб організувати компоненти системи таким чином, щоб кожна частина вирішувала окрему проблему або повний аспект функціональності, а не змішувала кілька проблем разом. Цей підхід покращує модульність, зручність обслуговування та масштабованість програмних систем.

Основні властивості розподілу відповідальностей наступні:

Модульність. SoC заохочує розбивати складні системи 5 на менші, більш керовані частини, кожна з яких вирішує окрему проблему. Такий модульний підхід полегшує розуміння, розробку та підтримку програмних систем.

Обслуговуваність. Завдяки розділенню проблем, зміни та оновлення в одному аспекті системи з меншою ймовірністю вплинуть на інші частини. Це зменшує ризик небажаних побічних ефектів і полегшує підтримку та розвиток програмного забезпечення з часом. Крім того, коли потрібні модифікації.

Масштабованість. SoC сприяє створенню дизайну, який дозволяє легко масштабуватися. Коли вимоги до системи змінюються або зростають, нові проблеми можна вирішити шляхом додавання або модифікації окремих модулів без необхідності внесення значних змін в інші частини системи. Така гнучкість дозволяє програмним системам ефективно адаптуватися до потреб, що змінюються.

Можливість багаторазового використання. Виокремлення проблем часто призводить до створення компонентів багаторазового використання. Після того, як проблему виділено в окремий модуль, його 5 можна повторно використовувати в різних частинах системи або навіть в абсолютно різних проектах. Це скорочує 5 час і зусилля на розробку та сприяє узгодженості між додатками.

Принцип Separation of Concerns є основою для створення 5 програмного забезпечення. Оскільки складні системи можуть мати безліч компонентів і підсистем, які взаємодіють між собою, важливо створити таку архітектуру, яка б дозволила

кожному компоненту працювати незалежно від інших. Архітектурні рівні в контексті SoC — це логічне розділення системи на різні рівні, кожен з яких має свою чітко визначену відповідальність. Нижче наведено ці самі рівні та їх приклад реалізації:

Рівень представлення (Presentation Layer) – це рівень, який відповідає за взаємодія з користувачем або з іншими системами. Основна його мета – це обробити вхідні дані від користувача та відобразити їх. Якщо розглядати в контексті дипломного проєкту, то рівень предаставлення можна розділити на два формати - це веб- інтерфейс та контролер, або ж обробник запитів, на сервері, який обробляє запити користувача. В інших варіантах це може бути як і мобільний додаток або десктоп-клієнт(UI).

Рівень бізнес-логіки (Business Logic Layer) – це рівень, який відповідає за основну логіку роботи системи, тобто може виконувати різні обчислення за потреби, проводити перевірку даних, обробляти запити з рівня представлення. Простими словами цей рівень відображає інтерфейс між вищим рівнем та нижним рівнем, де вищий рівень це обробники запитів(рівень представлення), а нижчий це рівень доступу до даних або рівень інтеграції, про які написано нижче, тобто основна його мета – це зв'язок з іншими модулями для виконання складних операцій. Важливо розуміти, що цей рівень є найважливішим в системі та рідко коли міняється. Отже для коректної комунікації між рівнями, потрібно забезпечити незалежну передачу даних між ними, наприклад: якщо поміняти рівень представлення – це ніяким чином не має повпливати на рівень бізнес логіки, це і є основна перевага розподілу відповідальностей.

Рівень доступу до даних (Data Access Layer) – це рівень, який займається взаємодією з базою даних або іншими обробниками даних. Він містить інтерфейси для обробки даних, а саме – зберігання, отримання та модифікації. Цей рівень повинен зберігати абстракцію над реальними джерелами даних, дозволяючи замінити технології зберігання без впливу на інші частини програми. Тут відбувається щось похоже як в рівні бізнес логіки. Умовно, якщо ми поміняємо СУБД з PostgreSQL на MySQL або ж взагалі на NoSQL, то рівень, який вище цього – тобто бізнес логіка, не повинен ніяк про це знати.

Рівень інтеграції (Integration Layer) – це рівень, який відповідає за комунікацію з іншими системами або зовнішніми службами. Основна його мета полягає в тому, щоб обробляти API-запити, комунікувати з іншими сервісами або обробляти обмін даними між різними частинами програми. Як приклад можна привести комунікація з платіжними сервісами. Тут так само все повинно абстрагуватися, щоб при зміні сервісу оплати – це не повпливало на працездатність сервісів вище рівнем.

Передача даних між рівнями зображено на рисунку 1.4.

Рисунок 1.4 – Передача даних між різними рівнями

Шаблони проєктування

Шаблони проєктування (Design Patterns) - це узагальнені рішення, які описують способи організації коду, що забезпечують повторне використання, розширюваність та зрозумілість архітектури програмного продукту. У контексті веб-розробки шаблони проєктування відіграють важливу роль при побудові масштабованих, підтримуваних і гнучких систем, особливо в умовах мікросервісної архітектури, де важлива чітка взаємодія між компонентами. Вони поділяються на 3 основні каталоги:

Каталог 1. Шаблон створення (Creational patterns) – ці шаблони забезпечують різноманітні механізми створення об’єктів, які підвищують гнучкість і повторне використання існуючого коду. Із шаблонів створення можна виділити наступні:

Шаблон Singleton – це шаблон, який забезпечує існування лише одного екземпляра класу або структури з глобальним доступом до нього. Це особливо корисно для об'єктів, які мають бути єдиними у системі, наприклад, конфігурацій, логерів, підключення до бази даних. У веб-додатках Singleton часто застосовується для ініціалізації пулу з’єднань до бази даних або клієнта для зовнішнього API(інтерфейсу програмного забезпечення), що має бути єдиним у всьому застосунку.

Шаблон Factory Method – це шаблон створення, який надає інтерфейс для створення об’єктів у суперкласі, але дозволяє підкласам змінювати тип об’єктів, які будуть створюватися. Цей шаблон зручно використовувати, коли необхідно створювати різні типи повідомлень (email, SMS, push) залежно від контексту.

Шаблон Prototype - це креативний шаблон проектування, який дозволяє копіювати існуючі об’єкти, не роблячи код залежним від їхніх класів. Це дуже сильно спрощує написання коду, адже не потрібно багато раз переписувати вже існуючий код та можна модифікувати копію об’єкту за потреби.

Каталог 2. Структурний шаблон (Structural patterns) - ці шаблони пояснюють, як збирати об’єкти та класи у більші структури, зберігаючи при цьому ці структури гнучкими та ефективними. Вони полегшують створення складних структур шляхом композиції, а не наслідування, що робить код більш адаптованим до змін. У контексті веб-розробки структурні шаблони допомагають об’єднувати сервіси, компоненти, адаптувати API зовнішніх систем до внутрішньої логіки тощо. Можна виділити декілька основних структурних шаблонів:

Шаблон Adapter – шаблон, шо дозволяє об’єднати несумісні інтерфейси, комунікуючи як посередник між ними. Це особливо корисно, коли треба використати сторонню бібліотеку або зовнішній API, який не відповідає внутрішньому контракту системи.

Шаблон Facade – шаблон, що надає спрощений інтерфейс до складної підсистеми. Він інкапсулює взаємодію між кількома компонентами або сервісами, дозволяючи клієнтському коду працювати лише з одним об’єктом замість кількох. Якщо, умовно, в системі є кілька сервісів, наприклад: авторизації, платіжний, сервіс обробки замовлень. Фасад дозволяє об’єднати логіку взаємодії з цими сервісами в одному об'єкті, щоб зменшити складність на рівні клієнта.

Шаблон Proxy – шаблон, який дозволяє надавати заміну або заповнювач для іншого об’єкта. 11 Проксі-сервер контролює доступ до оригінального об’єкта, дозволяючи виконати щось до або після того, як запит досягне вихідного об’єкта. Найбільш популярний приклад шаблону проксі – це Nginx server, який виступає як і проксі так і load-balancer для системи.

Каталог 3.Шаблони 8 поведінки (Behavioral patterns) - 8 ці шаблони 8 пов’язані з алгоритмами та розподілом відповідальності між об’єктами. Основні шаблони поведінки:

Шаблон Observer – це шаблон, що визначає залежність типу 8 "один до багатьох" між об’єктами, так що при зміні стану одного з них всі залежні повідомляються та оновлюються автоматично. Може добре підійти для реалізації event-driven systems(подієво-орієнтованих систем), наприклад, підписки на Kafka-топіки, де сервіси реагують на появу нових повідомлень.

Шаблон Iterator – це шаблон, що який дозволяє проходити елементи колекції, не відкриваючи її базове представлення (список, стек, дерево тощо). Корисний у випадках, коли необхідно обробити набір об’єктів із різних джерел.

Шаблон Strategy – це шаблон, який дозволяє визначити набір взаємозамінних алгоритмів і вибирати один із них під час виконання. Поведінка об’єкта інкапсулюється в окремі «стратегії», що реалізують спільний інтерфейс. Підходить для задач, де вибір алгоритму залежить від вхідних параметрів або конфігурації, наприклад: вибір способу аторизації: JWT, OAuth.

Короткий опис кожного шаблону зображено на рисунку 1.5

Рисунок 1.5 – Передача даних між різними рівнями

Принципи чистого коду та SOLID

Чистий код - це код, який легко читати, легко розуміти та легко модифікувати. Це код, позбавлений непотрібної складності, надмірності та плутанини. Чистий код відповідає набору конвенцій та найкращих практик, які роблять його більш узгодженим, що полегшує безперешкодну роботу кількох розробників над одним проектом. Важливість

чистого коду полягає в таких властивостях:

Читабельність. Чистий код легко читається, а це означає, що будь-хто, зможе швидко його зрозуміти. Це скорочує час, необхідний для розуміння функціональності коду, що призводить до швидшої розробки та налагодження.

Зручність супроводу. Коли пишеться чистий код, стає легше підтримувати і розширювати додаток з часом. Це має вирішальне значення в життєвому циклі розробки програмного забезпечення.

Зменшення кількості помилок. Чистий код зменшує ймовірність появи помилок. Код, який важко зрозуміти, більш схильний до помилок під час модифікацій або вдосконалень.

Ефективність. Чистий код - це ефективний код. Він зазвичай працює швидше і використовує менше ресурсів, оскільки уникає непотрібних операцій і складності.

Найпоширеніші приклади, в яких місцях код можна покращити:

Змістовні імена змінних та функцій, наприклад: var x int = 24. Набагато краще буде написати, що означає це число: var years_in_months int = 24. Це не тільки полегшить читання код після довгої перерви, а і буде конструктивніше виглядати. Теж саме і з функціями, замість довгої функції: func processUserData(user model.User) {}, Краще розділити її на меншу: func validateUserInput(userInput model.UserInput) {} та функцію func saveUserToDatabase(user mode.User) {}.

Також одна із дуже поширених проблем це повторювання одних і тих самих дій, в цьому може допомогти прицнип DRY(Don't Repeat Yourself) – принцип, що каже 1 уникати дублювання коду. Повторюваний код важче підтримувати і збільшує ризик невідповідностей. Потрібно виділяти загальну функціональність у функції, методи або класи, щоб сприяти повторному використанню коду. Коли потрібно буде внести зміни, то всього лише потрібно буде зробити це в одному місці. Наприклад: func calculateBookPrice(quantity, price float64) float64{return quantity * price} та func calculateLaptopPrice(quantity, price float64) float64{return quantity * price} - Це можна винести в окрему функцію задля того, щоб уникнути повторень.

Окрім загальних рекомендацій чистого коду, важливим підходом до організації якісного програмного забезпечення є дотримання принципів SOLID. Це акронім, що об'єднує п’ять основних принципів об'єктно-орієнтованого програмування, 1 які сприяють створенню розширюваного, 1 підтримуваного та стабільного коду:

16 S – 1 Single Responsibility Principle (Принцип єдиної відповідальності) Клас або

модуль повинен мати 16 лише одну причину для зміни. Тобто кожен клас 1 повинен виконувати лише одну функцію або відповідати за один аспект системи. Приклад: якщо клас відповідає за збереження даних користувача, він не повинен ще й відповідати за логування чи валідацію.

O – Open/Closed 1 Principle (Принцип відкритості/закритості) Програмні сутності повинні бути відкритими для розширення, але закритими для модифікації. Це означає, що ми 16 повинні мати змогу змінювати поведінку класів, не змінюючи їх код напряму – зазвичай це досягається через використання абстракцій. Приклад: замість того, щоб змінювати існуючий код методу обробки платежу, можна реалізувати новий клас StripePayment або PaypalPayment, які реалізовують один інтерфейс PaymentProcessor.

16 L – 1 Liskov Substitution Principle (Принцип підстановки Барбари 16 Лісков) Об'єкти підкласів повинні бути взаємозамінними з об'єктами батьківських класів без порушення логіки програми. Іншими словами, дочірній клас не повинен змінювати очікувану поведінку батьківського. Приклад: якщо клас Bird має метод fly(), то підклас Penguin не повинен його наслідувати, бо пінгвіни не літають — це порушує очікування.

16 I – 1 Interface Segregation Principle (Принцип розділення інтерфейсів) Краще мати багато вузьких інтерфейсів, ніж один великий. 1 Клієнти не повинні залежати від методів, які вони не використовують. Приклад: 13 замість одного великого інтерфейсу Worker з методами Work(), Eat(), Sleep(), краще розділити його на менші – Workable, Eatable тощо.

D 22 – 1 Dependency Inversion Principle (Принцип інверсії залежностей) Модулі високого рівня не повинні залежати від модулів низького рівня. Обидва повинні залежати від абстракцій. Крім того, 13 абстракції не повинні залежати від деталей — деталі повинні 1 залежати від абстракцій. Приклад: 13 замість створення екземпляру класу напряму (db := NewPostgresDB()), краще інжектувати інтерфейс Database, який реалізується PostgresDB.

Мова Go не є об'єктно-орієнтованою в класичному розумінні цього терміну. Вона не має таких понять як класи, успадкування, або абстрактні базові класи, але натомість реалізує деякі об'єктно-орієнтовані концепції через структури (struct) та інтерфейси (interface). У неї немає класичного наслідування, але є композиція, тобто замість того, щоб успадковувати поля та методи батьківської структури, вона включає одну структуру в іншу. Інтерфейси в Go є неявними, тобто структура реалізує інтерфейс, якщо має всі необхідні методи, без необхідності явно імплементувати інтерфейс. Це дозволяє легко використовувати принципи інверсії залежностей та розділення інтерфейсів, що само собою спрощує тестування та розробку систем. Також можна додавати нові типи, які реалізують той самий інтерфейс, не змінюючи існуючий код.

Методи тестування веб-додатку та забезпечення його якості

Забезпечення якості веб-додатку є одним з ключових етапів у процесі розробки програмного забезпечення. Незалежно від складності архітектури чи обраного стеку технологій, тестування відіграє важливу роль у виявленні помилок на ранніх етапах розробки та гарантує стабільність, надійність і передбачуваність роботи системи в умовах реального навантаження. Тестування дозволяє уникнути критичних помилок , зменшити витрати на підтримку та забезпечити позитивний користувацький досвід.

Одним із фундаментальних елементів тестування є юніт-тести. Вони перевіряють окремі одиниці логіки (функції, методи, сервіси) в ізольованому середовищі. Основною метою таких тестів є перевірка правильності роботи конкретного блоку коду без залежностей від бази даних, зовнішніх API або інфраструктурних компонентів. В дипломному проєкті було реалізовано декілька юніт тестів для провірки роботи механізму часу в тесті.

У межах реалізації я орієнтувався на принцип “тестування через поведінку”, тобто перевірку того, як система поводиться в умовах, максимально наближених до реальних. Такий підхід дозволяє сфокусувати увагу не лише на тому, як працює окремий фрагмент коду, а й на тому, як це впливає на всю систему загалом.

ВИБІР ТЕХНОЛОГІЙ ДЛЯ РОЗРОБКИ ВЕБ-ДОДАТКУ

Вибір мови програмування для розробки веб додатку

У процесі вибору технологій для реалізації дипломного проєкту особливу увагу було приділено вибору мови програмування. Основними критеріями були продуктивність, підтримка конкурентного програмування, зручність розробки, а також перспективність самої мови у сучасній розробці високонавантажених веб-додатків. З огляду на ці фактори, було прийнято рішення використовувати мову Go.

Мова програмування Go – це відносно нова мова, якщо порівнювати її з іншими, заснована в 2009 році компанією Google. Основна мета її створення – це простота, швидкодія. В ній є механізми конкурентності, які спрощують написання програм, що отримують максимальну віддачу від багатоядерних і мережевих машин, а нова система типів дозволяє гнучко і модульно будувати програми. Go швидко компілюється в машинний код, але при цьому має зручну систему збирання сміття та потужну систему рефлексії під час виконання. Це швидка, статично типізована, скомпільована мова. Саме конкурентність робить її дуже швидкою та популярною, адже до порівняння з мовою програмування Java, набагато легше та менш ресурсозатратно зробити одну горутину - це назва конкурентної функції в го, мінімальна вага якої всього лише 2КБ, ніж цілого потоку в Java. Також варто зазначити,

що Go активно використовується для розробки високонавантажених систем у провідних компаніях (Google, Uber, Dropbox тощо), що свідчить про його надійність та ефективність у реальних умовах. Він також має просту та ефективну систему управління залежностями (Go Modules), чудову документацію і стандартну бібліотеку, яка охоплює більшість типових задач.

Процесор може виконувати більше одного завдання одночасно, тільки якщо він має більше одного ядра, що дозволяє розподіляти завдання між ядрами. Кожне ядро може мати декілька потоків, а кожен потік може виконувати декілька горутин, перемикаючись між ними залежно від завдання та його стану. Горутини є легкими і управляються за допомогою середовища виконання Go, яке динамічно розподіляє їх між потоками ОС. Це може здатися повільним і неефективним, але процесор працює настільки швидко, що наші очі навіть не помічають різниці.

Саме через швидкодію горутин, вибір на розробку тестових завдань в реальному часі для дипломного проєкту впав на Go. Ця мова програмування також дозволяє синхронізовувати дані, між різними горутинами, задля подальшої обробки, що, само собою, було використано в дипломному проєкті, основна логіка обробки гри базується на комунікації та синхронізації за допомогою, так званих, каналів. Канали - це шляхи, які з'єднують паралельні горутини. Можна надсилати будь які значення у канали з однієї горутини та отримувати ці значення з іншої.

Вибір технології для серверної частини веб-додатку

При створенні серверної частини веб-додатку важливо обрати ефективні технології, які забезпечують стабільну, масштабовану та зручну в розробці інфраструктуру. У дипломному проєкті було використано такі ключові технології: протокол HTTP, фреймворк Gin, а також WebSocket для обробки подій у реальному часі.

Hypertext Transfer Protocol 2 (HTTP) - це протокол прикладного рівня для 2 передачі гіпермедійних документів, 2 таких як HTML. Він був розроблений для зв'язку між веб-браузерами та веб-серверами, але також може використовуватися для інших цілей, таких як міжмашинний зв'язок, програмний доступ до API. Передачу даних по протоколу HTTP зображено на рисунку 2.1.

Рисунок 2.1 – Передача даних по протоколу HTTP

HTTP працює за класичною моделлю клієнт-сервер, 2 коли клієнт відкриває з'єднання, щоб 2 зробити запит, 2 а потім чекає на 2 відповідь від сервера. 2 HTTP - це протокол без стану, 2 тобто 2 сервер не зберігає жодних даних про 2 сеанс 2 між двома запитами, хоча пізніша поява файлів cookie додає стан до деяких взаємодій між клієнтом і сервером.

Цей протокол складається з заголовків, методу запита, тіла запиту, коду-стану відповіді, та тіла відповіді. Заголовки повідомлень використовуються для передачі метаданих про ресурс або HTTP-повідомлення, а також для опису поведінки клієнта або сервера. Методи запиту вказують на мету запиту і на те, що очікується в разі успішного виконання запиту. Найпоширенішими методами є GET і POST для отримання і відправки даних на сервери відповідно, але є й інші методи, які створені для різних цілей. Кожен запит, окрім запитів з методом GET, має своє тіло запиту, що дозволяє передавати дані між сервером та клієнтом. За допомогою заголовків можна також передавати дані у вигляді ключ-значення, наприклад: X-ACCESS-KEY:SECRET_KEY.

Для побудови REST API було використано фреймворк Gin — один із найпопулярніших HTTP-фреймворків у екосистемі Go. Gin є надбудовою над стандартною бібліотекою net/http, яка спрощує маршрутизацію, обробку запитів, логування. Основні переваги Gin:

Висока продуктивність — фреймворк побудований на базі швидкого маршрутизатора http/router.

Зручна структура коду — підтримка групування маршрутів, middleware та контексту.

Гнучкість — легка інтеграція з іншими бібліотеками та пакетами.

Наявність великої кількості прикладів, плагінів і документації, що спростило розробку значною мірою.

Використання Gin дозволило швидко реалізувати зручне та гнучке API для клієнтської частини веб-додатку.

Gin підтримує протокол передачі даних websockets, що буде описано в подальшому.

Однією з ключових можливостей Gin є підтримка middleware — проміжних обробників, які виконуються до або після основної логіки маршруту. У дипломному проєкті було реалізовано middleware для логування запитів, обробки авторизації та контролю доступу, що значно підвищило безпеку та спростило підтримку коду.

Оскільки дипломний проєкт передбачає комунікацію в реальному часі, а саме - це комунікація між сервером та клієнтом наприклад при виборі відповіді, було обрано протокол WebSocket — це дуплексне з’єднання поверх TCP, яке дозволяє клієнту і серверу обмінюватися даними в обох напрямках без необхідності повторного встановлення з’єднання. Цей протокол являється надбудовою над протоколом HTTP та зберігає постійне з’єднання, що значно зменшує затримки та навантаження на мережу при частих оновленнях. У реалізації було використано бібліотеку gorilla/websocket, яка є стандартом для WebSocket у Go. З її допомогою була побудована система обміну

повідомленнями між учасниками гри, що забезпечує швидке реагування на дії користувачів і високу інтерактивність.

1 У порівнянні з альтернативними підходами для реалізації реального часу, такими як long polling або Server-Sent Events (SSE), WebSocket має значні переваги. Наприклад, при використанні long polling кожен запит до сервера створює нове HTTP-з'єднання, що створює зайве навантаження на сервер та мережу. SSE дозволяє лише односторонній зв'язок (від сервера до клієнта), що не підходить для задач з двосторонньою взаємодією, як у випадку взаємодії під час тесту. WebSocket забезпечує двосторонній постійний зв'язок із мінімальними затримками, що є критично важливим для оперативного реагування.

Вибір технології для клієнтської частини веб-додатку

Візуальна частина, тобто фронтенд, є надзвичайно важливою складовою будь-якого веб-додатку, оскільки саме з нею безпосередньо взаємодіє користувач. Незалежно від якості реалізації серверної логіки, недружній або застарілий інтерфейс, як правило, відлякує користувача й формує негативне враження. Тому 1 під час розробки клієнтської частини дипломного проєкту було обрано сучасні та зручні інструменти для створення інтуїтивно зрозумілого і швидкого інтерфейсу.

Для розмітки сторінки було використано HTML. Основною технологією для побудови фронтенду було обрано HTML, CSS, HTMX та в якихось моментах JavaScript, а саме відображення відбувається за допомогою server-side rendering технологією Templ, опис якої буде наведе нижче.

Для стилізації використовувалась CSS з використанням адаптивної верстки, щоб інтерфейс коректно виглядав як на комп'ютерах, так і на мобільних пристроях. Було реалізовано зручне розташування елементів, кольорову гаму, що не перевантажує зір, а також логічне компонування інформації на сторінці.

Для того, щоб інтерфейс був адаптивним, було використано HTMX-бібліотека, яка надає доступ до AJAX з HTML через атрибути, 17 що дає змогу створювати сучасні призначені для користувача інтерфейси, користуючись простотою гіпертексту. Зокрема, вибір впав на HTMX через підтримку websockets, що значною мірою спростило відображення та реалізацію на користувацькій стороні. Реалізація та поєднання всіх цих компонентів буде наведено нижче.

Вибір системи управління базами даних

У будь-якому веб-додатку, і не тільки, важливою складовою є система збереження та обробки даних. Від вибору відповідних технологій для зберігання залежить

стабільність, масштабованість та швидкодія всього додатку. У дипломному проєкті були використані дві основні технології зберігання даних: реляційна база даних PostgreSQL для постійного зберігання структурованих даних та Redis — як високошвидкісне in-memory сховище для тимчасових даних і обміну інформацією в реальному часі.

PostgreSQL 17 — одна з найпопулярніших у світі систем управління реляційними базами даних з відкритим кодом. Вона є стабільною, перевіреною роками. У дипломному проєкті PostgreSQL було обрано через такі ключові переваги:

Надійність та стабільність. PostgreSQL 1 активно використовується в багатьох продакшн-рішеннях. Це забезпечує впевненість у надійності зберігання даних.

Зручна структура даних. Реляційна модель даних дозволяє створювати складні взаємозв'язки між сутностями, що дуже зручно для представлення складних об'єктів, як-от користувачі, сесії, результати, відповіді тощо.

SQL як стандарт. мова SQL дозволяє легко писати запити до бази, працювати з фільтрами, агрегатами, групуваннями — все це є базовим інструментарієм для аналітики та обробки запитів.

PostgreSQL використовувалась для зберігання постійної інформації: профілів користувачів, тестів, питань та відповідей до них. Це дані, які мають довготривалу цінність і потребують надійного зберігання.

3 Redis — це високошвидкісне сховище даних у оперативній пам'яті, яке 3 підтримує різні структури даних: 7 рядки, хеші, списки, множини тощо. Redis не є заміною класичної бази даних, але чудово підходить як допоміжний інструмент для тимчасового кешування. Причини вибору Redis:

Швидкість. Оскільки Redis зберігає дані 7 в оперативній пам'яті, доступ до них майже миттєвий.

Підтримка TTL (time to live). Redis дозволяє встановлювати час життя для кожного ключа, що дуже зручно для тимчасових даних, які повинні автоматично видалятися.

Простота інтеграції. Redis має офіційний Go-клієнт (go-redis), який дозволяє швидко та просто підключитися до сховища і взаємодіяти з ним.

У дипломному проєкті Redis виконує роль кешу для часто запитуваних даних, а саме це тести – щоб не перенавантажувати основну базу даних складними транзакціями, та збереження тимчасових даних про пройдені тестування.

Поєднання PostgreSQL і Redis дозволило побудувати надійний, масштабований і швидкий бекенд. Реляційна база даних забезпечує цілісність та структурованість даних,

а Redis — високу швидкість обробки подій і збереження тимчасової інформації.

Забезпечення безпеки розробленого додатку

Безпека користувача є одним із ключових аспектів під час розробки сучасних веб-додатків. У цьому проєкті було прийнято рішення застосувати такі підходи до захисту даних і аутентифікації користувачів:

Для аутентифікації користувачів використовується механізм веб-токену у форматі JSON 10 (JSON Web Token). Після 10 успішного 10 входу користувача система генерує токен, який містить закодовану інформацію (payload) — наприклад, user_id та інші атрибути. Токен підписується 10 за допомогою секретного ключа, щоб унеможливити його підробку. Усі подальші запити до захищених ресурсів супроводжуються цим токеном у сесії cookie. Використання JWT 3 зображено на рисунку 2.2.

Рисунок 2.2 – Використання веб-токену у форматі JSON

Переваги веб-токену у форматі JSON. Безсерверна аутентифікація — не потрібно зберігати сесії на сервері, зручна передача інформації між клієнтом і сервером та підтримка обмеженого терміну дії (expiration), що дозволяє користувачеві не бути зареєстрованим в системі назавжди.

Для збереження паролів користувачів у базі даних застосовується алгоритм bcrypt. Це сучасний криптографічний алгоритм хешування, який забезпечує: Стійкість до brute-force атак завдяки налаштуванню "cost factor". Можливість безпечної перевірки пароля при логіні, не зберігаючи сам пароль. У процесі реєстрації користувача його пароль хешується за допомогою bcrypt.GenerateFromPassword(), а при вході — порівнюється збережений хеш із введеним паролем за допомогою bcrypt.CompareHashAndPassword().

Також, задля безпеки було використано змінні середовища, які задаються в окремому файлі .env, який є ізольований від навколишнього середовища. В ньому зберігаються такі чутливі дані як: паролі до БД, адреси підключення, секретні ключі. Також в .env є змінні, які не пов’язані з безпекою задля конфігурування наприклад часу, за який кеш в Redis має пропасти.

Вибір архітектури для серверної частини

В даному дипломному проєкті використано монолітну архітектуру для серверної частини розробленого додатку з можливістю подальшого розвитку з іншою архітектурою, а саме з мікросервісною архітектурою задля розділення відповідальностей по сервісам. В цей один монолітний сервіс входить такий функціонал:

Перш за все – це відповідальність за реєстрацію та вхід, обробку користувацьких даних та збереження їх в базу даних. При потребі це можна винести в сервіс authentification, а задля безпечної та швидкої передачі даних між сервісами можна використовувати фреймворк від google - gRPC, який базується на протоколі RPC(Remote Procedure Call). Простими словами - це як викликати функцію на іншій машині, але тільки за умови що дві машини підключені за допомогою протоколу tcp. Приклад наведено на рисунку 2.3.

Рисунок 2.3 – Приклад використання gRPC протоколу для передачі даних

Відповідальність за взаємодію з темами та питаннями, також як і в попередньому випадку – це можна винести в окремий сервіс разом з іншою базою даних, задля збереження хорошого принципу database per service, що означає, що сервіс має мати свою базу даних яка ніяк на пряму не повинна взаємодіяти з іншими, тобто всі отримання та записи даних також потрібно передавати по протоколам передачі даних, в пріоритеті, звісно, gRPC.

Відповідальність за саму логіку тестування, тобто вся логіка початку, завершення всіх нюансів тесту прописана також тут.

РЕАЛІЗАЦІЯ ВЕБ-ДОДАТКУ ДЛЯ ТЕСТУВАННЯ

Проєктування СУБД, реалізація та інтеграція

База даних, як вже зазначалось вище, є важливим компонентом застосунку, що відповідає за зберігання, обробку та доступ до структурованої інформації. У цьому проєкті використовувалась реляційна СУБД PostgreSQL для основного зберігання даних та Redis як високошвидкісний кеш.

Було реалізовано збереження даних, таких як інформація про користувачів та теми - в PostgreSQL, а дані які часто використовуються та результати тестувань в Redis.

Для PostgreSQL було реалізовано механізм міграцій для контрольованого та послідовного оновлення структури бази даних. Це дозволяє уникнути помилок при ручному створенні таблиць, забезпечує збереження історії змін та дає змогу швидко розгортати нові версії схеми. Цей підхід дозволив підвищити стабільність і передбачуваність при оновленні системи і простоту розгортання на різних середовищах. Міграції перед запуском самої системи запускаються як набір впорядкованих запитів в самій базі даних.

Для зручного керування даними, в процесі розробки використовувався pgAdmin 4 - інструмент візуального проектування бази даних, який об’єднує розробку, адміністрування, проектування бази даних, створення та підтримку, що в значній мірі допомогло, адже не потрібно було кожного разу підключатися вручну та з терміналу

вписувати потрібні запити. Водночас, для забезпечення безперебійного доступу до навчальної інформації навіть у випадку відсутності інтернет-з'єднання, реалізовано підтримку офлайн-режиму.

Застосунок автоматично зберігає локальні копії необхідних даних, дозволяючи студенту переглядати розклад, курси, дедлайни та оцінки без підключення до мережі. Після відновлення інтернет-з'єднання всі зміни синхронізуються з сервером. Такий підхід гарантує, що користувач завжди матиме доступ до актуальної інформації незалежно від технічних обставин. Вигляд інтерфейсу pgAdmin 4 наведено на рисунку 3.1.

3 Рисунок 3.1 – Вигляд візуального інтерфейсу програми pgAdmin 4

Інтеграція мови програмування Go з БД PostgreSQL здійснювалась за допомогою драйвера, куди потрібно прописати URI для підключення. Є багато різних варіацій драйверів, кожен зумовлений своїми перевагами та недостатками. В дипломному проєкті було використано драйвер "github.com/jackc/pgx/v5/pgxpool", який дозволяє вручну створювати конфігуровані запити в базу 14 даних.

14 В ході розробки було розділено рівень комунікації з базою даних та інші рівні задля того щоб, в разі потреби замінити або ж, при перенесенні на інший сервіс – позбутися від неї.

Для Redis було використано драйвер “github.com/redis/go-redis/v9”. Він забезпечує зручний інтерфейс для підключення до Redis-серверу, а також підтримує роботу з базовими типами даних. Для керування даними, наприклад для провірки або для тестування було використано середовище docker, в якому і запустився сервер Redis. Про це буде описано нижче. Код реалізації комунікації з базою даних наведено в Додатку А.

Розробка серверної частини

Як і вже зазначалося вище, серверна частина дипломної роботи була реалізована з використанням фреймворку Gin, одного з найпопулярніших HTTP-фреймворків для мови Go. Однією з основних переваг Gin є його висока швидкодія завдяки використанню маршрутизатора на базі Radix tree, а також підтримка middleware, що полегшує обробку запитів, логування, аутентифікацію, кешування та інші задачі. Крім того, фреймворк має зручний механізм групування маршрутів, що дозволяє краще організовувати код, особливо при роботі з великою їх кількістю.

Перед кожним HTTP запитом, окрім маршрутів реєстрації, йде провірка на JWT, задля того, щоб ідентифікувати користувача, щоб в подальшому подавати йому персоніфаковану інформацію і ідентифікувати його за ім’ям на тестах.

Сама логіка тестування реалізована таким чином, що всі клієнти, тобто користувачі, підключаються до однієї кімнати. Кожна кімната має свій так званий контролер подій, в подальшому буде називатися менеджером. Менеджер створюється при створенні кімнати, та чекає подальших вказівок, які будуть описані нижче. Також при створенні кімнати вибирається тема, на яку буде проводитися опитування та кількість учасників за якої почнеться тестування.

При підключенні до кімнати створюється websocket з'єднання на основі протоколу HTTP, по якому в подальшому будуть передаватися дані на користувацький інтерфейс та зчитуватись дані з нього. Кожен клієнт контролюється менеджером, тобто зв'язком "один до багатьох". Рішення щодо вибору архітектури, де є один контролер подій і багато клієнтів які його слухають, полягає в наступному: така модель дозволяє централізовано керувати подіями, що надходять від серверної частини, та ефективно розподіляти їх між усіма підключеними клієнтами. Це спрощує обробку логіки взаємодії 3 в реальному часі, зменшує дублювання обробки подій на стороні клієнта та забезпечує масштабованість. Також, через те, що тестування реалізовано на протоколі websockets, тобто в браузері, надійність з'єднання 3 не може бути пріоритетом, тому, одною із основних причин вибору цієї архітектури – є збереження стану клієнта на момент втрати з'єднання, наприклад – закриття вкладки з тестом чи відставання в мережі. Тому, замість того, щоб зберігати дані в зовнішніх сховищах, що значною мірою вплинуло би на працездатність, було вирішено реалізувати контроль за цим станом зі сторони менеджера. Схема підключення клієнтів до менеджера 3 зображена на рисунку 3.2.

3 Рисунок 3.2 – Схема підключення клієнтів до менеджера

Менеджер відповідає за такі дії як:

Початок та кінець тестування. Кожна кімната має свій відведений час на самоліквідацію. Це зроблено задля зловживання ресурсами сервера. Головний учасник – той, хто створив цю кімнату, має право почати тестування раніше, ніж зайдуть усі учасники. Кінець наступає тоді, коли вичерпаються питання.

Обновлення даних кожному клієнту при кожному оновлені статусу. Наприклад якщо один із учасників відповів на питання – всім іншим покажеться, що він відповів.

Зміна питань. Питання змінюється за двох умов, якщо всі відповіли або вичерпався час.

Обробка часу, яка 7 здійснюється за допомогою внутрішньої бібліотеки Go Time, який раз в секунду – надсилає значення в канал, тим самим повідомляючи всім клієнтам, що час помінявся.

Розрахунок правильних відповідей, їх обробка та розрахунок таблиці лідерів.

Менеджеру при кожній відповіді приходять сигнали.

Окремо може видавати статистику для учасника, який створив це тестування.

Стан учасників – а саме їхня готовність, підключення та відключення.

Також, при підключенні клієнта до кімнати, створюється дві горутини, так звані канали даних –канал запису та канал читання, за допомогою яких і виконується зчитування та запис даних з користувацького інтерфейсу. Також є окремий канал даних для глядача. В канал запису входять наступні можливості:

Відобразити питання.

Відобразити людей, 7 які ще не відповіли на питання.

Відображення таблиці лідерів.

Відображення підключених користувачів перед тестуванням.

Відображення часу.

Канал читання відповідає за:

Зчитування даних з користувацького інтерфейсу за допомогою безкінечного циклу. Цикл закінчується, коли розривається з’єднання між клієнтом та сервером на рівні браузера.

Відправка менеджеру стану клієнта про готовність

Відправка менеджеру відповіді клієнта

Для передачі даних між компонентами в системі самого менеджера - задіяно 21 канал, кожен з яких відповідає за свій функціонал. Канали в Go одразу синхронізують дані, що зменшує потребу обробляти сценарії, де потенційно може виникнути Race Condition. Race Condition – це стан системи, коли декілька горутин одночасно намагаються оперувати з одними й тими самими даними. Для синхронізації даних, які не передаються між каналами, а в більшій мірі – це хеш-таблиці, структури даних, які зберігають дані в форматі ключ-значення, було задіяно механізм синхронізації – mutexes, який позволяє тимчасово блокувати доступ іншим горутинам до конкретного ресурсу. Реалізація менеджера та клієнта продемонстарована в додатку Б.

Комунікація серверної частини та користувацького інтерфейсу залежить напряму від викликаної функції. Канал запису чекає вказівок від менеджера, задля відображення даних на користувацький інтерфейс, наприклад: коли помінявся час – менеджер

сповістив кожного підключеного клієнта атомарно, задля обновлення часу на екрані користувача.

Задля того, щоб не забирати ресурс у менеджера, було вирішено зробити worker pool, який забирає навантаження з менеджера записом у кеш персональні рещультати учасників.

Розробка користувацького інтерфейсу

Користувацький інтерфейс реалізовано з використанням серверного відображення (Server-Side Rendering) на базі бібліотеки a-h/templ, яка дозволяє створювати типобезпечні HTML-компоненти за допомогою Go. Компоненти templ компілюються в Go-код, 23 що забезпечує високий рівень продуктивності та зручність у підтримці. Генерація HTML відбувається безпосередньо на сервері, а вже сформована сторінка надсилається клієнту, що забезпечує швидке завантаження.

Основною перевагою, через яку було обрано саме цю реалізацію – це можливість передавати дані з функцій Go напряму в HTML-код. Templ дозволяє працювати зі структурами, інтерфейсами, каналами, а також включати власні логічні блоки — цикли, умовні конструкції тощо. В середині шаблону можна імпортувати стандартні бібліотеки Go, наприклад fmt, для форматування тексту, або ж будь-які інші імпорти.

Однією з таких бібліотек, яка активно використовувалася під час розробки, була HTMX. HTMX надає можливість виконувати HTTP-запити безпосередньо з HTML-елементів (через атрибути hx-get, hx-post, hx-swap), що значно спрощує динамічну взаємодію з сервером без потреби в JavaScript. У зв’язці з templ, HTMX дозволяє реалізувати реактивні інтерфейси з мінімальними витратами: новий HTML генерується на сервері, а HTMX відповідає за його заміну в DOM.

Такий підхід дозволив обійтись без складного JavaScript-фреймворку, зберігши повну контрольованість логіки на бекенді та спростивши тестування. Крім того, використання HTMX спростило реалізацію таких функцій, як динамічна пагінація, оновлення частин сторінки, підтвердження дій користувача без повного перезавантаження сторінки тощо.

Також в відповіді від сервера при middleware було використано HX-REDIRECT заголовок, реалізований на стороні HTMX, задля перенесення на сторінку реєстрації у випадку, коли користувач не має cookie Authorization.

Контейнеризація, ізоляція та підключення

Для ізоляції середовища виконання було використано технологію контейнеризації

Docker. Усі складові системи, а саме веб сервер та бази даних були винесені в окремі контейнери, що дозволило уникнути конфліктів залежностей та забезпечити відтворюваність середовища розробки.

Контейнеризація — це спосіб ізолювати додаток разом із усіма його залежностями (бібліотеками, середовищем виконання тощо) у незалежний контейнер. Такий контейнер можна запускати на будь-якій системі, де є Docker, і він буде працювати однаково.

Docker — це інструмент для створення, розгортання та запуску контейнерів. Контейнер — це щось середнє між віртуальною машиною та звичайним процесом, тобто він має власну файлову систему, мережу, змінні середовища тощо, але використовує ядро хост-системи, тобто набагато менш затребований в ресурсах, ніж віртуальна машина.

Основні складові середовища Docker:

Dockerfile — описує, як створити образ (наприклад, встановити Go, додати код, налаштувати запуск).

Образ (image) — це шаблон, з якого створюються контейнери. Контейнер — це запущений образ.

Docker Compose — інструмент для запуску кількох контейнерів разом, у випадку розробки дипломного проєкту – це сам сервер, PostgreSQL та Redis.

Вигляд інтерфейсу програми Docker зображено на рисунку 3.3.

Рисунок 3.3 – Інтерфейс програми Docker

Завдяки ізоляції, легко можна підключитися до PostgreSQL або Redis напряму, що в контексті розробки дипломного проєкту сильно полегшило тестування, сумісність та саму розробку.

Також, в разі потреби, для розгортання цього додатку в відкрите середовище інтернет, на більшості хмарних сервісів, потрібно використовувати Dockerfile задля того, щоб перенести всі можливі залежності.

Версія контролю, використання Git та Github

Для керування змінами в кодовій базі була використана 19 система контролю версій Git у поєднанні з хостингом репозиторію на платформі GitHub. Git — це 19 розподілена система керування версіями, яка дозволяє зберігати історію змін у коді, повертатися до будь-якого попереднього стану проєкту та працювати над різними функціями паралельно у гілках. Git працює локально, що дозволяє створювати коміти, гілки,

переглядати історію без доступу до інтернету. Це надає гнучкість і контроль над розробкою.

Використання технології Git в дипломному проєкті дозволило легко та безпроблемно експерментувати над основною логікою тестування, що в значній мірі вплинуло на швидкодію самого тесту, адже було багато варіантів реалізації на етапі розробки, які зберігалися в своїй окремій гілці, тобто незалежно один від одного, що дозволило вибрати найкращий варіант серед усіх зроблених. Також, як основною перевагою використання Git – було збереження поточної версії програми, з можливістю повернутися до неї в любий момент, без можливості втрати теперішнього коду. Це дуже помогло в моменті, коли при добавлянні нового функціоналу, інші в якійсь мірі могли перестати нормально функціонувати, де, якраз таки, і поміг Git.

Також, у процесі розробки проєкту активно використовувалася платформа GitHub, що слугувала хостингом для репозиторію та основним інструментом колаборації над кодом. Git забезпечує збереження повної історії змін у кодовій базі. Це дозволяє повертатися до попередніх версій, порівнювати зміни між комітами, бачити, хто і коли вніс ту чи іншу зміну. Git працює за принципом розподіленої системи, тобто кожен розробник має на своєму комп’ютері повноцінну копію репозиторію з усією історією змін. Це дозволяє працювати автономно, без постійного з'єднання з сервером, і лише згодом зливати зміни до спільного центрального репозиторію.

У цьому проєкті структура роботи з Git будувалась на принципі створення окремих гілок (branches) для кожної нової задачі або функціоналу. Наприклад, для реалізації авторизації створювалася гілка feature/auth, де вся відповідна логіка імплементувалася ізольовано. Після завершення розробки зміни зливалися (merge) у головну гілку проєкту через pull request, що дозволяло переглянути код перед злиттям, перевірити його на наявність помилок або реалізувати покращення. Таким чином забезпечувалась стабільність основної гілки, а кожен функціонал проходив через проміжну перевірку.

GitHub виступав як віддалений сервер для зберігання репозиторію, але його функціонал значно ширший, ніж просто зберігання коду. Платформа дозволяла централізовано організувати розробку, зокрема через pull requests — інструмент, що використовується для пропозиції змін до репозиторію.

Тестування веб-додатку

Тестування дипломного проєкту – це важливий компонент розробки, де виявляють критичні несправності та в подальшому виправляються. У межах роботи було проведено ручне тестування функціоналу веб-додатку.

Основна увага приділялася перевірці правильності взаємодії між модулями, коректності

відображення даних у UI, а також стабільності системи під час основних сценаріїв використання. Для цього я використовував підхід, smoke testing – перевірки, що основний функціонал працює без збоїв. Тестування відбувалося з точки зору кінцевого користувача: вручну перевірявся кожен основний маршрут, форма, кнопка, авторизація, створення та перегляд тем та питань.

При реєстрації протестовано дублікацію імен та електронних пошт задля ідентифікування користувачів. При вході в систему протестовано несумісніть даних. В разі якоїсь несумісності або помилки знизу під формою висвітлюється інформація про це. Такоє протестовано унікальність ввійденого в систему користувача за допомогою cookies та веб-токену у форматі JSON.

Протестовано створення тем для тестів, їх індивідуальність, різні можливі сценарії при яких може виникнути помилка – напркилад, створення однакових тем. Також, завдяки збереженню даних в реляційній базі даних – є можливість уніфікувати та не давати можливість переглядати іншим користувачам, свою тему, що не дає можливість зловживати списуванням та дає змогу зробити тест унікальним. Візуальний вигляд створення, видалення та перегляду тем зображено на рисунку 3.4

Рисунок 3.4 – Візуальний вигляд створення, видалення та перегляду тем

Кімната — це місце, де збираються учасники, щоб одночасно пройти певний набір тестів. Вона містить інформацію про самі тести, учасників та результати їхніх відповідей. Під час її стоврення вибирається кількість учасників, кількість питань, які будуть на тесті та сам тест. При створенні кімнати – було протестовано дублікацію кімнат, та розроблено механізм для запобігання цього. При підключенні, якщо коду кімнати не існує – під формою виствілюється інформація. Форма створення кімнати зображена на рисунку 3.5.

Рисунок 3.5 – Форма створення кімнати

При перевірці основного функціоналу - в пріоритеті було саме тестування. Задля провірки справності роботи, було протестовано різні сценарії випадків, як і з різними користувачами через програму RadminVPN та локально через анонімні вкладки.

Першочергово, через неможливість гарантування стабільного підключення по протоколу, зроблено механізм, який дозволяє зберігати результати на момент відключення та можливість спокійно перепідключитися для подальшого продовження тесту.

Також, реалізовано механізм блокування для декількох підключень, задля запобігання несанкціонованих дій, які можуть призвести до нечесних результатів, тобто користувач може бути підключений лише один раз, з одного браузера та пристрою без можливості

зайти декілька раз з одного акаунта. В такому випадку – йому буде писати відповідний текст, який попередить його про це.

Якщо користувач з якохось причин був вимушений перезайти на тест, при умові, що він його вже виконав – менеджер збереже цю інформацію, та виведе на екран вікно очікування наступного питання. В користувачів, які вже відповіли на питання теж буде відображатися в реальному часі підключення та відключення клієнтів.

Режим спостерігання дозволяє в реальному часі, не відповідаючи на тести, спостерігати за результатами користувачів, дивитися питання та правильну відповідь.

Перед початком тесту, зроблено ініціалізючий віртуальний простір, де збираються всі підключені користувачі. Мета реалізації цього компоненту полягала в очікуванні під'єднання всіх учасників, щоб зробити процес тестування чеснішим, адже без цього мехнізму, учасники, які би підключились раніше – мали би значну перевагу над тими, хто підключився скоріше. на рисунку 3.6.

Рисунок 3.6 – Очікування всіх користувачів

Протестовано, також, звичайні сценарії, наприклад: зміна питання, коли вичерпався час, зміна питання, коли відповіли всі учасники, зміна питання, коли останній учасник вийшов і т.д. Тестування починається за двох умов: якщо підключилась кількість користувачів, яка була вказана при ініціалізації кімнати або коли нажато кнопку "примусовий початок"- в такому випадку, тест розпочнеться одразу, не дочекавшись повної кількість учасників. Учасники, які не приєдналися перед тестуванням, мають змогу сопкійно це зробити посеред нього. Візуальне відображення тестування по мові програмування С зображено на рисунку 3.7.

Рисунок3.7 – Візуальне відображення тестування по мові програмування С

За допомогою результатів тестування, було проаналізовано мінуси та плюси системи, які в подальшому допомогли покращити продуктивність. Один із таких випадків – це перероблення функціоналу часу та відображення даних менеджера. Після тестування було виявлено недолік, що при деяких сценаріх, не відображався час та не був синхронізований з іншими користувачами.

Отже, тестування веб-додатку є не менш важливою частиною розробки програмного забезпечення в цілому. Суть тестування, першочергово, є для усунення несправностей, які в процесі розробки були ненавмисно зроблені, через людський фактор

Посилання

Це джерела виділених збігів у вашому документі. Кожен збіг позначено темно-зеленим числом, яке відповідає вказаному тут джерелу. Джерела впорядковані за схожістю — чим вищий бал, тим сильніше збіг.

#	Джерело	%
1	repository.sspu.edu.ua	1.5%
2	ela.kpi.ua	0.6%
3	docs.vntu.edu.ua	0.5%
4	javascript.org.ua	0.4%
5	polaridad.es	0.3%
6	openarchive.nure.ua	0.3%
7	ela.kpi.ua	0.3%
8	pdf.lib.vntu.edu.ua	0.3%
9	er.nau.edu.ua	0.2%
10	essuir.sumdu.edu.ua	0.2%
11	troyanda19.if.ua	0.2%
12	ela.kpi.ua	0.2%
13	javascript.org.ua	0.2%
14	knuba.edu.ua	0.2%
15	javascript.org.ua	0.2%
16	evnuir.vnu.edu.ua	0.2%
17	dokumen.tips	0.2%
18	it.nuft.edu.ua	0.1%
19	dspace.wunu.edu.ua	0.1%
20	dspace.nuph.edu.ua	0.1%
21	biotechuniv.edu.ua	0.1%
22	openarchive.nure.ua	0.0%
23	mdu.edu.ua	0.0%

Plag

Дякуємо, що перевірили
свій документ за допомогою
Plag!