Міністерство освіти і науки України

НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ «КИЄВО-МОГИЛЯНСЬКА АКАДЕМІЯ»

Кафедра мережних технологій факультету інформатики

Особливості мови та екосистеми Crystal у контексті розробки веб-серверних застосунків

Текстова частина до курсової роботи

за спеціальністю „Інженерія Програмного Забезпечення”

Керівник курсової роботи

Захоженко П.О.

____

(підпис)

“____” _____________ 2024 р.

Виконав студент

Кривошеєв І.С.

“____” _____________ 2024 р.

КИЇВ 2024

Зміст

Вступ 3
Анотація 5
Розділ 1. Огляд існуючих рішень 6
1.1 Spring Boot(Java) 6
1.2 Django(Python) 7
1.3 Laravel (PHP) 8
1.4 Ruby on Rails 9
Розділ 2. Огляд використаних технологій та інструментів 11
2.1 Crystal 11
2.2 RESTful архітектура 13
2.3 Repository Design Pattern 16
2.4 Dependency Injection 17
2.5 JSON Web Token 18
2.6 Автоматизоване тестування на Crystal 20
Розділ 3. Реалізація додатку 22
3.1 Технічне завдання 22
3.2 База даних 24
3.3 Розробка застосунку 26
3.3.1 Початкові налаштування 26
3.3.2 Зв’язок з базою 28
3.3.3 Приклад сервісу 29
3.3.4 Приклад контроллера 30
3.3.5 Авторизація та JWT 32
3.3.6 Розробка API тестів 35
3.3.7 Можливості метапрограмування 36
3.4 Результати 37
Висновок 40
Список джерел 41

Вступ

У сучасному світі існує багато різних фреймворків, написаних на багатьох мовах програмування, для розробки веб-серверних застосунків. Найбільш популярними фреймворками для бекенд розробки, згідно до статистики популярності на Github, є Laravel(PHP), Django(Python), Spring(Java), Flask(Python) Express JS(Javascript) [12]. Ці фреймворки мають розвинену екосистему із допоміжних бібліотек, багату документацію та велику кількість користувачів. Проте кожен має і свої недоліки і кожен рік з'являються та популяризуються нові фреймворки для розробки веб-серверних додатків.

Crystal — це молода статично типізована компільована мова програмування, яка має синтаксис, подібний до Ruby. Вона була створена з метою поєднати виразність Ruby зі швидкістю таких мов, як C. Код Crystal компілюється до ефективного машинного коду, через що її можна ефективно використовувати при розробці веб-серверної архітектури, якій необхідна перш за все швидкість для обробки великої кількості HTTP запитів за короткий час. І хоча ця мова є молодою, перша версія з'явилася у 2014 році, вона вже має достатньо розвинену екосистему для створення веб-серверних застосунків.

Метою цієї роботи є розробка ефективного веб-серверного застосунку з використанням мови та екосистеми Crystal, дослідження переваг та недоліків роботи з молодою та не повністю сформованою екосистемою у контексті розробки RESTful застосунку.

У першій частині роботи буде розглянуто існуючі мови та фреймворки для побудови серверних застосунків, які використовують архітектурний стиль REST API. У другій – використані технології та архітектурні паттерни при розробці

застосунку. І в останній відповідно практичну реалізацію застосунку з відповідною демонстрацією роботи застосунку, його продуктивності.

Анотація

Роботу присвячено розробці серверного застосунку з використанням мови програмування Crystal. Було створено серверний додаток з дотриманням архітектурного стилю REST API, продемонстровано можливості мови та екосистеми Crystal для розробки веб-серверних застосунків з використанням архітектурних паттернів. В якості бази даних було використано PostgreSQL, показано різні способи підключення до бази у Crystal. Також продемонстровано можливості автоматизованого тестування застосунків з Crystal.

Розділ 1. Огляд існуючих рішень

1.1 Spring Boot(Java)

Spring Boot — це потужний фреймворк для розробки додатків на мові Java, який спрощує процес створення самостійних, готових до запуску застосунків. Основною метою Spring Boot є забезпечення швидкого створення простих продуктивних додатків, які можуть бути легко запущені.[9]

Переваги:

- Легкість конфігурації: Spring Boot дотримується дизайн стилю «конвенції над конфігурацією», що дозволяє швидше створювати додатки без великої кількості XML конфігурацій.
- Вбудований контейнер сервлетів: Spring Boot має вбудовані контейнери сервлетів, такі як Tomcat, Jetty або Undertow, що дозволяє запускати додатки без необхідності встановлення додаткового сервера.
- Розвинена екосистема: Spring Boot є частиною екосистеми Spring, що має велику кількість додаткових модулів і бібліотек для різних потреб розробників.

Недоліки:

- Великий розмір дистрибутиву: За замовчуванням дистрибутив Spring Boot може мати досить великий розмір, оскільки включає в себе всі необхідні бібліотеки.
- Складність налаштування за потребою: Хоча Spring Boot намагається спростити багато конфігурацій, іноді додаткові налаштування можуть бути складними.
- Необхідність знань Spring Framework: Щоб ефективно використовувати Spring Boot, потрібно мати базове розуміння Spring Framework.

1.2 Django(Python)

Django — це високорівневий веб-фреймворк для розробки веб-додатків на мові програмування Python. Він спрощує процес створення веб-додатків шляхом надання ряду готових компонентів і структур для швидкої розробки. Основні принципи Django — це продуктивність, чистота дизайну та реюзабельність коду. Цей фреймворк використовує архітектурний принцип MVT (Model-View-Template).[10]

Переваги:

- Швидкість розробки: Django надає багато готових компонентів, таких як аутентифікація користувачів, адміністративна панель, форми, робота з базами даних і багато іншого
- Масштабованість: Django підтримує структуру проектів та додатків, що дозволяє легко масштабувати додатки з додаванням нових функцій чи розширенням існуючих.
- Вбудована безпека: Django має вбудовану захист від багатьох типів атак, таких як SQL ін'єкції, міжсайтові атаки (XSS) та інші, завдяки використанню вбудованих захисних механізмів.
- Розвинена екосистема: Django та Python має велику активну спільноту розробників і багату екосистему додаткових бібліотек і пакетів для різноманітних потреб.

Недоліки:

- Висока складність для початківців: Хоча Django пропонує швидку розробку, він також може бути складним для новачків через велику кількість функціональності.

- Менша гнучкість порівняно з мікрофреймворками: У випадках, коли потрібна максимальна гнучкість та мінімальність, Django може бути занадто важким.

1.3 Laravel (PHP)

Laravel - це веб-фреймворк для розробки веб-додатків на мові програмування PHP. Він створений з метою спрощення розробки веб-додатків, забезпечуючи гнучку архітектуру та багатий набір функцій[11].

Переваги:

- Простота використання: Laravel має чистий та елегантний синтаксис, що робить його дружнім для початківців і досвідчених розробників. Він пропонує швидку розробку завдяки ряду вбудованих функцій.
- Модульність та розширюваність: Laravel підтримує концепцію модульної розробки, де можна використовувати компоненти (пакети) з екосистеми Laravel для розширення функціоналу додатка.
- Автоматична генерація коду: Laravel надає інструменти для автоматичної генерації коду, такі як генерація контролерів, моделей, міграцій і т. д., що полегшує розробку.
- Безпека: Laravel включає вбудовані механізми захисту від таких атак, як XSS (міжсайтовий скриптінг) та SQL-ін'єкції.

Недоліки:

- Навантаження на сервер: Деякі функції Laravel споживають більше ресурсів сервера порівняно з іншими легшими фреймворками.

- Швидкодія: Хоча Laravel набагато покращив швидкодію протягом останніх версій, він все ще може бути менш ефективним порівняно з більш "легковаговими" фреймворками.
- Велика кількість файлів та структурний об'єм: Для деяких проектів структура Laravel може здатися складною і великою.

1.4 Ruby on Rails

Rails — це фреймворк розробки веб-додатків, написаний мовою програмування Ruby. Його розроблено, щоб полегшити програмування веб-додатків, роблячи припущення про те, що потрібно кожному розробнику для початку. Він дозволяє писати менше коду, досягаючи при цьому більшого, ніж багато інших мов і фреймворків. Основними принципами філософії Rails є DRY («Don't repeat yourself») і «конвенція над конфігурацією».[13]

Переваги:

- Швидка розробка: Rails забезпечує швидку розробку завдяки високорівневим абстракціям, генерації коду і стандартній структурі проекту. Він надає багато готових рішень і компонентів, таких як маршрутизація, ORM (Active Record), аутентифікація і багато іншого.
- Конвенція перед конфігурацією: Rails надає стандартну структуру проекту та конфігураційний файл, що дозволяє розробникам швидше розпочати роботу над проектом, не втрачаючи час на повторну конфігурацію.
- Багата екосистема і активна спільнота: Rails має велику кількість готових бібліотек, модулів та плагінів, які дозволяють швидко розширювати функціонал додатків.

Недоліки:

- Високі вимоги до ресурсів: За рахунок великої кількості абстракцій та готових рішень, Rails може вимагати більшої кількості ресурсів сервера порівняно з іншими фреймворками.
- Відсутність гнучкості: Хоча Rails пропонує багато готових рішень, він може бути менш гнучким для деяких випадків, де потрібно значне відхилення від стандартних підходів.

Розділ 2. Огляд використаних технологій та інструментів

2.1 Crystal

Мова програмування Crystal — це високорівнева мова програмування загального призначення з синтаксисом, схожим на Ruby, але з покращеною продуктивністю та статичною типізацією. Crystal розроблена з метою поєднання зручності та елегантності Ruby з ефективністю та безпекою, характерними для компільованих мов.

Синтаксис мови Crystal був успадкований багато в чому від Ruby. Так, для оголошення змінної не потрібно використовувати ключове слово або обов'язково вказувати тип. Ім'я змінної завжди починається з малої літери Unicode (або підкреслення, але це зарезервовано для особливих випадків) і може складатися з буквено-цифрових символів або підкреслення. За конвенцією, імена змінних і методів використовують snake_case, типи - PascalCase. Crystal має стандартний набір ключових слів для керування потоком інформації – if, elif, else, unless, case для умовних переходів, while та until для циклів. Методи оголошуються за допомогою ключового слова def, можуть мати параметри за замовчуванням. На відміну від C-подібних мов, закінчення блоку коду позначається ключовим словом end, а для виклику метода не обов'язково використовувати круглі дужки.

```
def say_hello(recipient = "World")
  puts "Hello #{recipient}!"
end

say_hello
say_hello "Crystal"
```

Рисунок 1. Приклад синтаксису Crystal

Crystal має вбудовані типи для булевих значень(Bool), цілочисельні значення зі знаком і без (Int і UInt), числа з рухомою крапкою (Float32 і Float64), символ Юнікоду(Char), стрічка(String), масив(Array), словник(Hash), регулярні вирази(Regex), типи тільки для читання – кортеж(Tuple) і іменований кортеж(Named Tuple), переліки(Enum).

Crystal – об’єктно-орієнтована мова, всі значення в програмі – об’єкти певного типу. Відповідно можна визначати свої класи, використовувати принципи наслідування та поліморфізму. За замовчуванням всі методи класу публічні, для обмеження доступності можна створювати приватні методи та захищені методи для нащадків класу. Можна визначати статичні методи, статичні та абстрактні класи. Має вбудований автоматизований збирач сміття. Також, як і Ruby надає великі можливості для метапрограмування. Так, Crystal має підтримку макросів, що дозволяє визначати спеціальні конструкції для генерації коду на етапі компіляції. За допомогою макросів розробники можуть динамічно створювати нові мовні конструкції, наприклад визначати нові методи класів. Іще, Crystal підтримує вбудований механізм рефлексії, що дозволяє аналізувати типи та структури даних в часі виконання програми. Рефлексія дає змогу динамічно створювати об'єкти, визначати та викликати методи, читати атрибути класів тощо.

На відміну від Ruby, Crystal дозволяє визначати обмеження типів для параметрів методів, значення, яке повертає метод, обмеження типу змінної при її визначенні, обмеження для полів класів. Це дуже допомагає при розробці складних, великих проектів у командах, а також дозволяє відловити більшість NullPointerException ще при етапі компіляції, бо за замовчуванням значення типів не можуть бути Null. Динамічно отримувати тип змінної у програмі можливо за допомогою ключового слова typeof.

Іще в Crystal реалізовано можливість виклику функцій низькорівневих бібліотек написаних на С і використання небезпечних засобів управління указниками.

Для багатопоточної роботи в Crystal можна використовувати механізм Threads, які дозволяє створювати та управляти виконанням коду в окремих потоках. Також реалізовано підтримку концепції волокон (fibers), яка дає змогу створювати легкі синхронні задачі, які можна відкладати і відновлювати без використання системних потоків. Фібри використовуються для асинхронного програмування та роботи з не блокуючим вводом/виводом. Вони є ефективним механізмом для роботи з великою кількістю легких задач, що не вимагають розділення ресурсів CPU.

Для створення, публікації та імпортування бібліотек у Crystal використовується вбудована система управління пакетами Shards. Для імпортування сторонньої бібліотеки достатньо вказати її як залежність у файлі shards.yml, вказати шлях до ресурсу, де вона була опублікована (наприклад Github) та інсталювати її за допомогою команди shards install.[1]

2.2 RESTful архітектура

REST, що означає REpresentational State Transfer — це архітектурний стиль програмного забезпечення, який визначає набір правил для створення веб-сервіси. Веб-сервіси, які відповідають архітектурному стилю REST, відомі як RESTful веб-сервіси. Це дозволяє запитуючим системам отримувати доступ до веб-ресурсів і маніпулювати ними за допомогою єдиного попередньо визначеного набору правил. Взаємодія в системах на основі REST відбувається через Інтернет-протокол передачі гіпертексту (HTTP).[2]

REST накладає наступні архітектурні обмеження:

- Уніфікований інтерфейс: це ключове обмеження, яке розрізняє REST API і не-REST API. Це означає, що має бути єдиний спосіб взаємодії з певним

сервером незалежно від пристрою чи типу програми (веб-сайт, мобільний додаток).

- Відсутність стану: Це означає, що необхідний стан для обробки запиту міститься в самому запиті, і сервер не зберігатиме нічого, пов’язаного із сеансом.
- Кешування: кожна відповідь сервера має включати інформацію, чи кешується відповідь чи ні, і протягом якого часу відповіді можуть кешуватися на стороні клієнта. Клієнт поверне дані зі свого кешу для будь-якого наступного запиту, і зникає необхідність знову надсилати запит на сервер.
- Клієнт-сервер: програма REST повинна мати архітектуру клієнт-сервер.
- Багаторівнева система: архітектура програми має складатися з кількох рівнів. Кожен рівень не знає нічого про будь-який інший рівень, окрім рівня безпосередньо знизу нього, і між клієнтом і кінцевим сервером може бути багато проміжних серверів.
- Код за запитом: це єдина необов’язкова функція. Згідно з цим, сервери також можуть надавати клієнту виконуваний код. Приклади коду на вимогу можуть включати скомпільовані компоненти, такі як Java Servlets або JavaScript.

Веб-сервер буде побудований з дотриманням архітектурного паттерну Controller-Service-Repository. Цей паттерн розбиває функціональність додатка на три основні компоненти: Контролер (Controller), Сервіс (Service) і Репозиторій (Repository). Кожен з цих компонентів має свої відповідні обов'язки і допомагає досягти принципу розділення відповідальностей (Separation of Concerns) і принципу єдиного обов'язку (Single Responsibility Principle). [3]

Основні компоненти паттерна Controller-Service-Repository:

- Контролер відповідає за обробку HTTP-запитів користувача і взаємодію з веб-інтерфейсом користувача. Контролер приймає дані від користувача, викликає відповідний сервіс для обробки бізнес-логіки, і потім повертає результат користувачеві.
- Сервіс містить основну бізнес-логіку додатка. Вони ізолюють деталі бізнес-логіки від контролера. Якщо бізнес-логіка вимагає отримання/збереження даних, сервіс підключається до репозиторію.
- Репозиторій відповідає за доступ до даних (збереження, вилучення, оновлення, видалення). Це абстракція бази даних або іншого джерела даних, яка використовується сервісами для збереження та отримання даних.

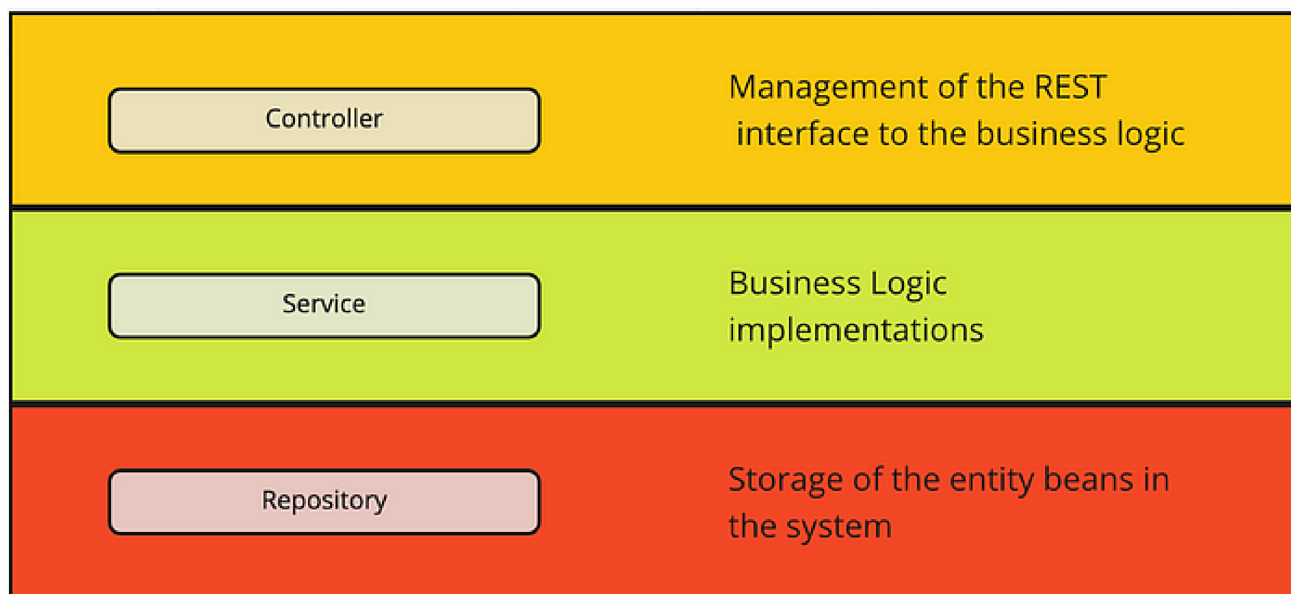

Рисунок 2. Controller-Service-Repository

Одним з переваг такого підходу з розділенням відповідальності є легкість тестування окремих компонентів, з'являється можливість імітувати суміжні шари та турбуватися лише про тестування відповідальності цього конкретного рівня.

Для реалізації Restful архітектури був використаний фреймворк Kemal.

Kemal — це легкий веб-фреймворк для мови програмування Crystal, спроектований для швидкої розробки веб-додатків. Він надає зручний і елегантний спосіб створення веб-серверів і обробки HTTP-запитів за допомогою Crystal. Для обробки запиту достатньо визначити шлях до ресурсу та назву HTTP-методу. Також він підтримує автоматичну подачу статичних файлів, створення фільтрів та проміжних функцій обробки клієнтського запиту для всіх або лише деяких запитів, зберігання контексту запиту між проміжними функціями, роботу з файлами та веб-сокетами.[4]

2.3 Repository Design Pattern

Шаблон проектування Repository — це шаблон проектування програмного забезпечення, який діє як проміжний рівень між бізнес-логікою програми та сховищем даних. Його основна мета — надати структурований і стандартизований спосіб доступу, керування та маніпулювання даними, абстрагуючись від базових деталей технологій зберігання даних. Цей шаблон сприяє чіткому розподілу проблем, роблячи програмне забезпечення більш придатним для обслуговування, тестування та адаптації до змін у джерелах даних, не заплутуючи основну логіку додатка складнощами доступу до даних.

По суті, шаблон проектування репозиторій є схемою для організації та спрощення доступу до даних, підвищення ефективності та гнучкості програмних систем.[5]

Для реалізації цього шаблону була використана бібліотека Jennifer. Основним елементом Jennifer є моделі, які розробники створюють як абстракцію для доступу до таблиці у реляційній базі даних. При створенні моделей доступні наступні функції:

- гнучке визначення схеми моделі
- визначення реляцій бази даних (belongs_to, has_many, has_one, has_and_belongs_to_many) - включаючи поліморфні

- вбудовані валідації полів
- визначення області запиту для конкретної моделі
- зворотні виклики(Callbacks)

Для створення складних SELECT SQL-запитів у Jennifer є інтерфейс Query DSL, який підтримує наступні можливості:

- отримання об'єктів моделі з бази даних
- отримання записів із певної таблиці
- швидке завантаження асоціацій моделі будь-яких рівнів
- підтримка загальних функцій SQL (включаючи функції агрегації)

Також Jennifer має широкі можливості для конфігурації доступу до бази даних та вбудовану систему управління міграціями бази даних.[6]

2.4 Dependency Injection

Dependency Injection (DI) — це шаблон проектування, який використовується в об’єктно-орієнтованому програмуванні при розробці програмного забезпечення для досягнення інверсії керування (IoC) між класами та їхніми залежностями. У DI залежності класу (тобто інші об’єкти, необхідні класу для виконання своїх функцій) «впроваджуються» в клас спеціальним сервісом-ін’єктором, а сам клас не створює залежностей і не керує ними.

Основна ідея Dependency Injection полягає в тому, щоб відокремити класи від їхніх залежностей, сприяючи слабкому зв’язку та роблячи класи більш придатними для повторного використання, тестування та підтримки. Цей шаблон допомагає досягти принципу єдиної відповідальності (SRP). В об’єктно-орієнтованому програмуванні залежності можуть впроваджуватися в конструкторі або в методі класу.[7]

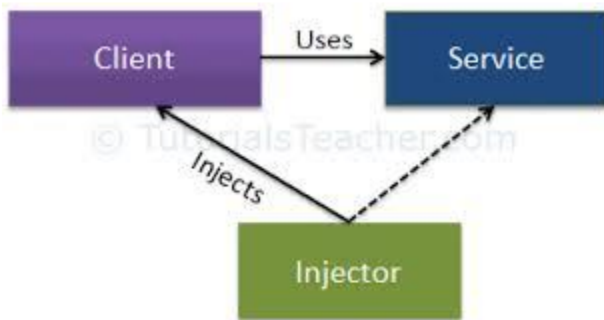

Рисунок 3. Dependency injection

Для реалізації цього шаблону в Crystal використовувалась бібліотека Crystal-DI. Вона підтримує реєстрацію залежностей в окремому модулі, автоматичну ін'єкцію залежностей в конструктори класів, збереження єдиного об'єкту сервісу, якщо необхідно щоб сервіс, що ін'єктиться, був ініціалізований один раз. Також можна ініціалізовувати залежності одного інтерфейсу різними класами-нащадками в залежності від класу для якого проводиться ініціалізація.[8]

2.5 JSON Web Token

Через те, що архітектура RESTful веб-сервіса не передбачає стану, зберігання інформації про користувача, його дозволи покладено на клієнт. Зазвичай для цього використовуються дві технології – HTTP-cookies та JSON Web Token (JWT). В цій роботі було використано JWT.

JWT — це відкритий стандарт (RFC 7519), який визначає компактний і самодостатній спосіб безпечної передачі інформації між сторонами як об'єкт JSON. Цю інформацію можна перевірити та довіряти їй, оскільки вона має цифровий підпис. JWT можна підписати за допомогою секрету (з алгоритмом HMAC) або пари відкритих/приватних ключів за допомогою RSA або ECDSA. JWT складається з трьох частин – заголовку, корисного навантаження та сигнатури. В заголовку вказується алгоритм шифрування використаний для

даного токену. В корисному навантаженні – будь-яка інформація у вигляді пар ключ-значення у форматі JSON. Сигнатура отримується шляхом кодування у форматі Base64Url двох інших частин і застосовування відповідного алгоритму шифрування. Сигнатура використовується для перевірки того, що повідомлення не було змінено на шляху, і, у випадку токенів, підписаних приватним ключем, вона також підтверджує, що відправник JWT є тим, за кого він себе видає.[14]

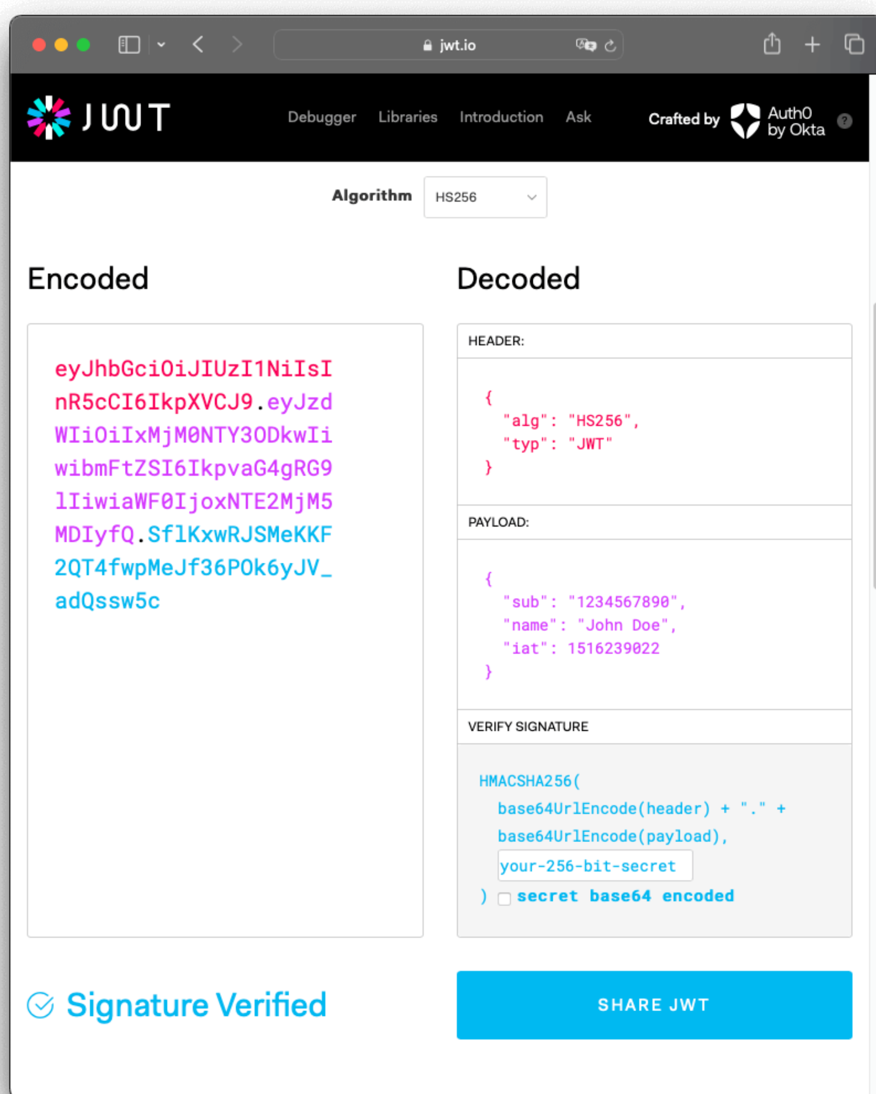

Рисунок 4. Приклад JWT токену

Під час автентифікації, коли користувач успішно ввійде, використовуючи свої облікові дані, додаток повертає веб-токен JSON. Цей токен зберігається та

надсилається з клієнтського застосунку. Надалі, щоразу, коли користувач хоче отримати доступ до захищеного маршруту або ресурсу, агент користувача повинен надіслати JWT, як правило, у HTTP-заголовку Authorization за схемою Authorization: Bearer <token>.

Для роботи з JWT в Crystal була використана бібліотека Crystal JWT. Вона містить функції для кодування, декодування та верифікації JWT токенів.[15]

2.6 Автоматизоване тестування на Crystal

Взагалі існує кілька типів автоматизованого тестування програмного продукту, кожен з яких спрямований на перевірку певних його аспектів. Основні типи автоматизованих тестів включають наступне:

- Модульне тестування (Unit Testing) спрямоване на перевірку окремих модулів, компонентів або функцій програмного коду. Метою є перевірка коректності роботи окремих частин програми без їх взаємодії з іншими компонентами.
- Інтеграційне тестування (Integration Testing) перевіряє взаємодію між різними модулями або компонентами програми.
- Стрес-тестування (Stress Testing) оцінює стійкість програми до навантаження та великої кількості запитів або користувачів.
- Тестування API (API Testing) – це тип тестування програмного забезпечення, який перевіряє інтерфейс програмування додатків (API) і перевіряє функціональність, безпеку та надійність інтерфейсу програмування.

Так як в рамках цієї роботи був створений REST API застосунок, буде розглянуто можливості для Тестування API на Crystal.

Для автоматизованого тестування у Crystal є вбудована бібліотека Spec. В рамках цієї бібліотеки визначаються окремі блоки для тестування поведінки

програми. Блок it містить приклад, який має викликати код, який потрібно перевірити, і визначити, що від нього очікується. Кожен приклад може містити кілька очікувань, але він має перевіряти лише одну конкретну поведінку.[16]

Також кожен об'єкт має методи екземпляра #should і #should_not. Ці методи викликаються для значення, яке перевіряється, із очікуванням як аргументом. Якщо очікування виправдано, виконання коду продовжується. Інакше тест завершився невдало і подальший код не виконується.

У тестових файлах специфікації структуровані за групами тестів, які визначаються розділами describe. Як правило, describe визначає зовнішню одиницю (наприклад, клас), яку необхідно перевірити специфікацією.

Для спрощення тестування REST API ендпоітів було також використано бібліотеку Mass Spec. Вона має методи для надсилання HTTP запитів за певним шляхом і отримання та перевірки інформації про відповідь включно з кодом відповіді, заголовків, тіла та автоматичного читання тіла, якщо воно прийшло у форматі JSON. Також є додаткові методи перевірки match та contain для полегшення перевірки відповіді API.[17]

Розділ 3. Реалізація додатку

3.1 Технічне завдання

В рамках практичної частини цієї роботи було реалізовано REST API за темою онлайн-квізу. Клієнтський інтерфейс реалізований за допомогою JavaScript та бібліотеки React. Користувач має можливість реєструватися/входити в систему якщо вже зареєстрований за своїм іменем та паролем. Після входу відкривається можливість проходити випадкові онлайн-тести. Питання та відповіді отримуються з використанням публічних API.

До початку проходження тесту користувачу надається можливість встановити параметри тесту :

- Кількість питань (від 5 до 20)
- API питань – одне з трьох варіантів: Trivia API, Quiz API, Open Trivia DB
- (Необов'язковий) складність – один з трьох варіантів: Проста, Середня, Складна. Якщо не вказано питання можуть бути будь-якої складності.
- (Необов'язковий) категорія – в залежності від API надається можливість додати категорію питань. Якщо не вказано питання можуть бути будь-якої категорії.

Після вибору параметрів інформація про цей тест надсилається на REST API для збереження. Після старту квізу користувачу буде показано перше питання тесту із варіантами відповідей. Після натискання на варіант користувач побачить правильний варіант та зможе перейти на наступне питання. Після переходу на питання відповідь та оцінка за питання також надсилаються на REST API для збереження.

Після завершення тесту користувач може побачити список всіх питань тесту, загальну оцінку за тест. Також користувач може перейти до сторінки своїх результатів і побачити всі свої результати тестів.

Загальна схема роботи застосунку(Рисунок 5):

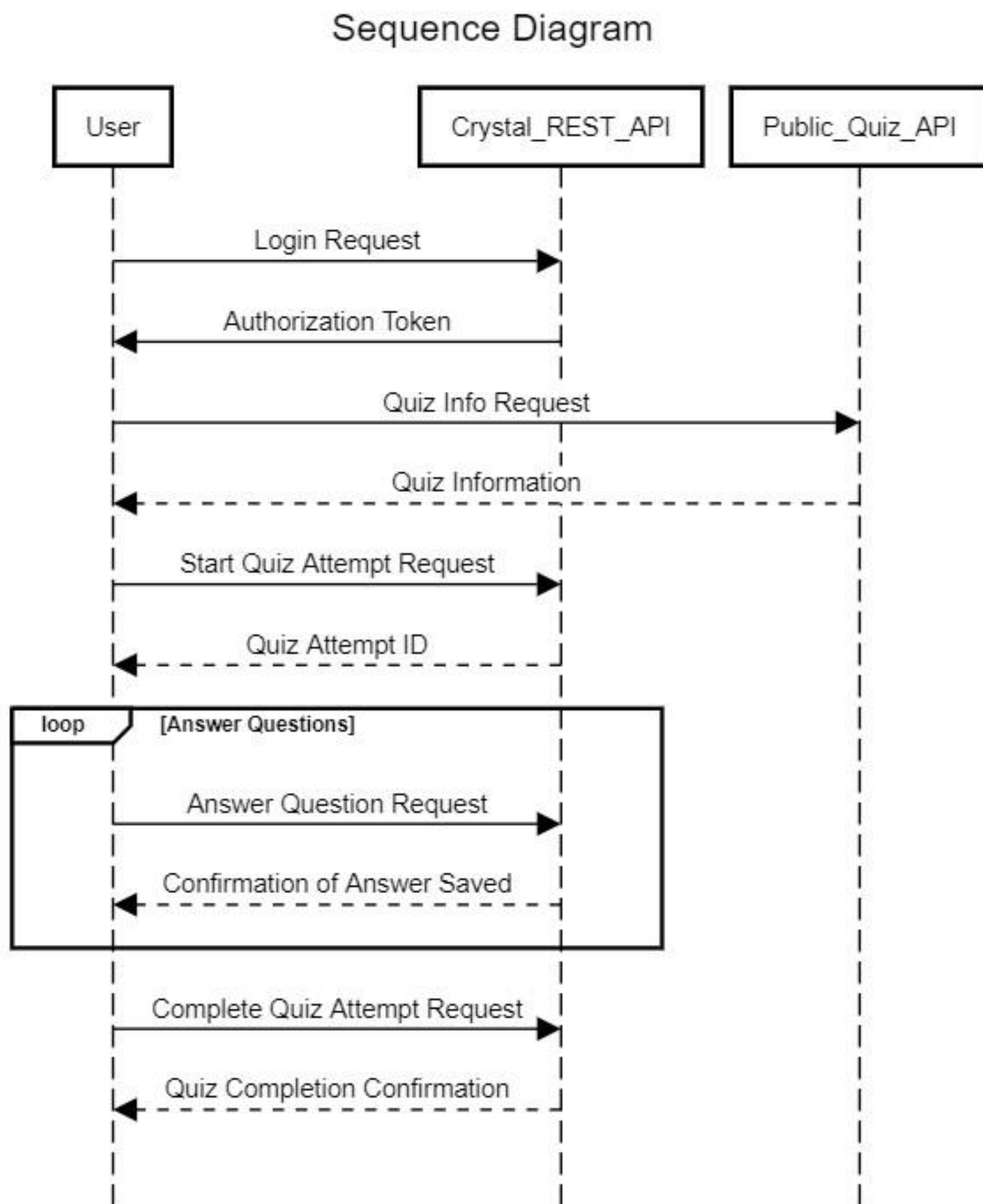

Рисунок 5. Sequence diagram

3.2 База даних

Для збереження даних було створено реляційну базу даних PostgreSQL. PostgreSQL – безкоштовна система керування базами даних яка використовується багатьма продуктовими проектами, має високу швидкість та багатий набір можливостей, тому є гарним вибором при розробці бази даних для REST API. В Crystal для підключення до PostgreSQL використовується бібліотека crystal-pg.[18]

Для збереження інформації про користувача використовується таблиця «users» з полями для збереження імені, email та паролю. Для збереження результатів тесту – «quiz_results». В ній зберігається інформація про тест та його результати – назва API квізу, категорія, складність, максимально можливий бал, статус (два варіанти – “Started” якщо тест розпочато, але не завершено; “Completed” якщо тест було завершено) дата початку, дата завершення. Таблиця «users» має зв’язок «один-до-багатьох» до «quiz_results», бо один користувач може мати багато результатів. Для збереження інформації про індивідуальний результат відповіді на питання використовується таблиця «question_results». Вона містить наступну інформацію – текст питання, категорія питання, складність питання, бал за відповідь користувача, правильна відповідь, неправильні відповіді. Таблиці «users» та «quiz_results» мають зв’язок «один-до-багатьох» до «question_results», бо користувач і квіз можуть мати багато результатів відповіді на питання.

Загальна діаграма бази даних(Рисунок 6):

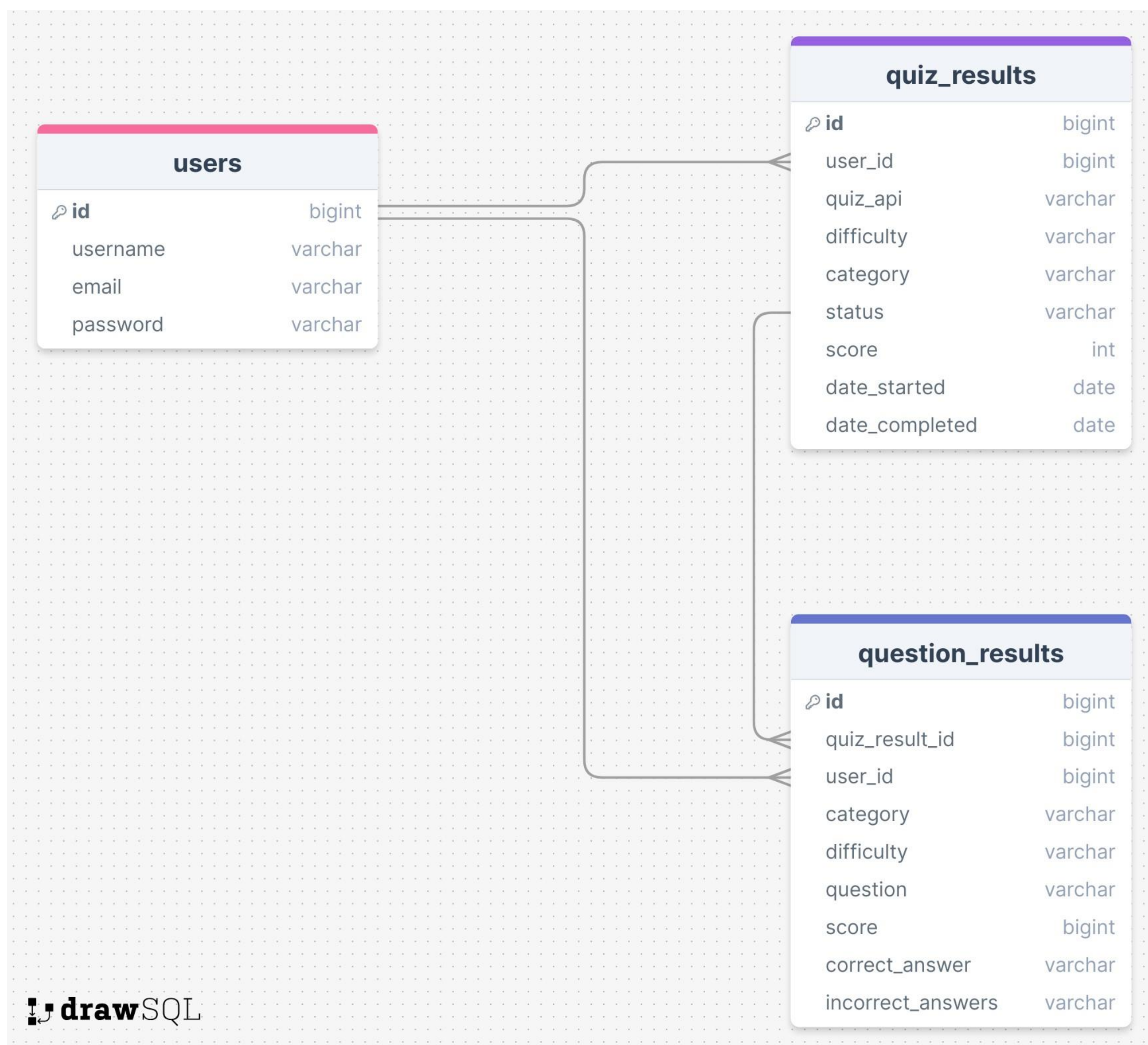

Рисунок 6. Database diagram

3.3 Розробка застосунку

3.3.1 Початкові налаштування

Для створення програм на Crystal необхідно встановити компілятор Crystal. Хоча Crystal дозволяє встановлювати компілятор на платформу Windows, цей варіант наразі зазначений як нестабільний і велика частина додаткових бібліотек може не підтримуватись. Тому для розробки було обрано платформу Windows Subsystem for Linux (WSL). WSL – це функціонал, що надається в операційній системі Windows 10 та 11, який надає можливість для запуску виконуваних файлів операційної системи Linux. Відомий редактор коду Visual Studio Code підтримує можливість розробки з WSL за допомогою розширення Visual Studio Code WSL, тому для розробки було обрано саме цей редактор.

Після встановлення Crystal є можливість виконувати консольні команди для створення проекту (*crystal init*), створення executable файлу (*crystal build*), запуску програми (*crystal run*), запуску тестів (*crystal spec*).

Всі залежності нашої програми встановлюються за допомоги вбудованої системи управління пакетами Shards і відображаються у файлі `*shards.yml*`. У цьому файлі зазначаються два типи залежностей – залежності які необхідні для коректної роботи програми і ті, які необхідні лише при розробці (наприклад бібліотеки для тестів). Також в цьому файлі зазначається інформація про репозиторій, його автора, використовувана версія Crystal. Встановлення залежностей відбувається за допомогою команди `*shards install*`.

Створений файл `*shards.yml*` (Рисунок 7):

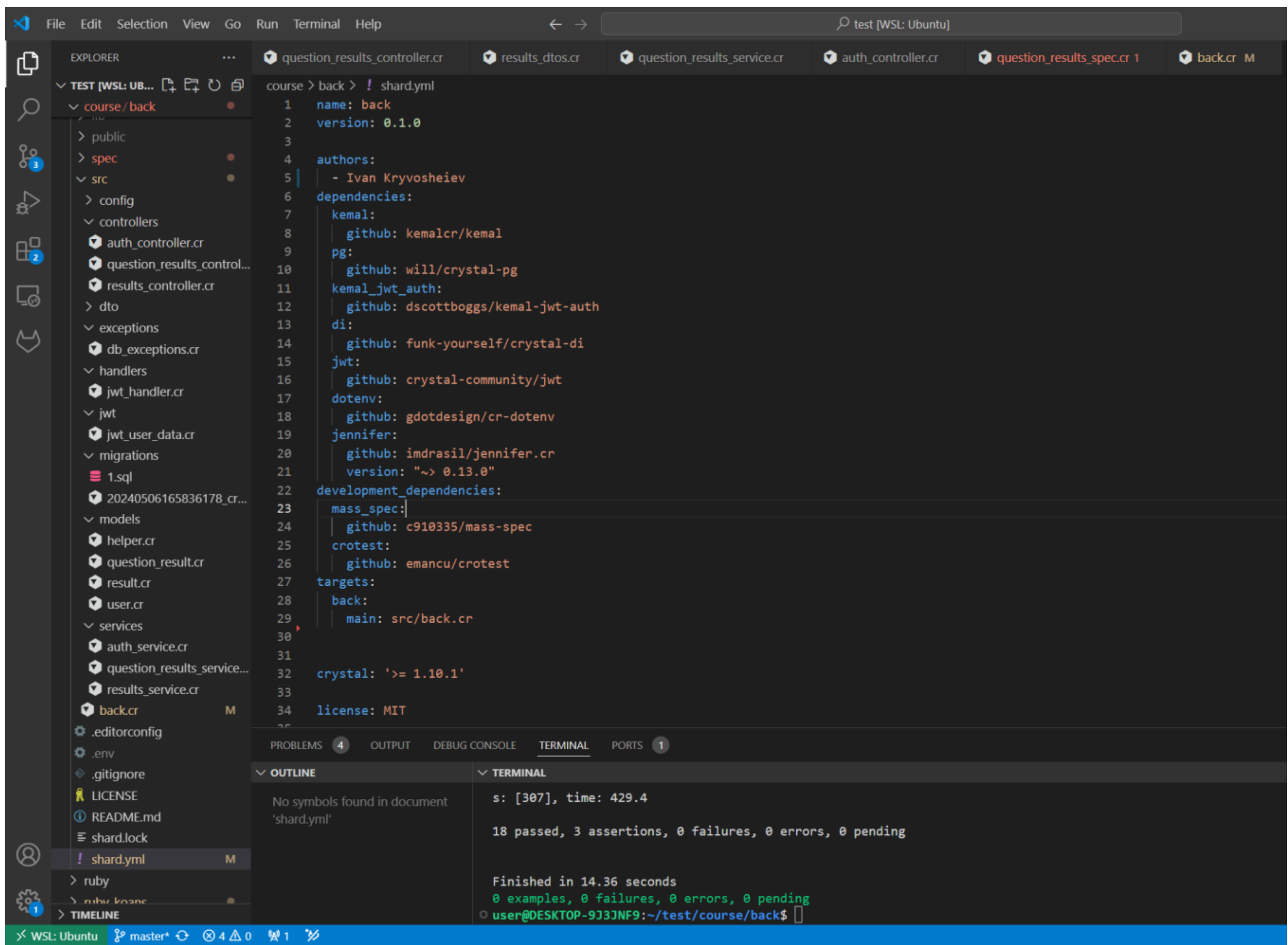

Рисунок 7. shards.yml

Загальна структура проекту відповідна – в директорії src загальний код проекту, розділений на піддиректорії за категоріями – контролери, сервіси, моделі, проміжні функцій при запитах(middleware), файли кофігурацій. В директорії spec знаходяться файли тестів, в public – статичні файли (html, js), які автоматично обслуговуються фреймворком Kemal.

Для конфігурації доступу до бази даних та збереження інших кофіденційних даних використовується файл .env. Взагалі, під час роботи з локальними секретами найкраще зберігати їх у окремому файлі конфігурації, а не закодувати ці секрети в програмі. Загальноприйнятий варіантом цього є використання файлу "dotenv", оскільки він підтримується багатьма мовами програмування та

часто з'являється в шаблонах файлів .gitignore. Для використання даних з файлу .env у Crystal використовується бібліотека cr-dotenv.[19]

3.3.2 Зв'язок з базою

Для кожної таблиці, яку ми визначили при створенні бази даних створюється відповідна модель з використанням бібліотеки Jennifer. За допомогою виклику методів table_name встановлюється посилання на таблицю SQL, в mapping зазначаються назви стовпців таблиці та тип даних для цих стовпців. Також можливо задавати валідацію для збережених даних. Так, наприклад за допомогою виклику 'validates_length : password, minimum: 6' перевіряється що всі паролі, які будуть збережені в таблицю матимуть не менше 6 символів. Також задаються зв'язки між таблицями за допомогою викликів 'has_one', 'has_many' та 'belongs_to'. (Рисунок 8).

```
course > back > src > models > user.cr > ...
1  require "./result.cr"
2  require "./question_result.cr"
3  class User < Jennifer::Model::Base
4    table_name "users"
5
6    mapping(
7      id: Primary32,
8      username: { type: String, default: "" },
9      email: { type: String, default: "" },
10     password: { type: String, default: "" },
11   )
12
13   validates_length :password, minimum: 6
14   has_many :results, Result, dependent: :destroy
15   has_many :question_results, QuestionResult, dependent: :destroy
16 end
17
```

Рисунок 8. модель User

Для початкового підключення Jennifer до бази при старті застосунку використовується допоміжний файл з викликом 'Jennifer::Config.from_uri(Config::DB_CONNECTION)'. Параметр 'DB_CONNECTION' ініціалізується в конфігураційному файлі, а значення для нього отримується з файлу .env.

3.3.3 Приклад сервісу

У файлах сервісів прописані функції, які потім використовуються контроллерами для відповідних ендпоінтів. Сенс цих функцій задається бізнес-логікою та потрібністю певним контроллерам. Задача сервісів – прописати цю логіку при взаємодії з відповідними моделями Jennifer. Таким чином досягається розподілення обов'язків, контроллер оперує лише функціями сервісів і не залежить від конкретних моделей, а моделі і рівень репозиторію не залежить від конкретної логіки певного ендпоінту.

Для прикладу розглянемо results_service(Рисунок 9). В ньому прописані відповідні функції:

- get_result_by_id – отримати конкретний результат за ідентифікатором. Для реалізації цього викликається Jennifer модель з використанням методу find_by, який повертає максимум один результат з бази даних.
- get_user_results – отримати всі результати за певним користувачем. Для цього викликається Jennifer метод where і результат приводиться до масиву відповідним методом to_a.
- has_result_by_id – перевірка чи результат існує за ідентифікатором. Використовуються методи where та exists.
- set_result_completed – переведення результату у completed статус. У методі отримується результат за ідентифікатором, перевіряється чи існує, виставляються відповідні нові параметри і виконується метод Jennifer save.
- insert_result – створення нового результату тесту для користувача. Для цього використовується Jennifer метод create для створення нового рядка у базі даних.

course > back > src > services > results_service.cr > ...

```
require "db"
require "pg"
require "../config/*"
require "crypto/bcrypt/password"
require "../models/result"
require "../exceptions/*"

module Services
  class ResultsService
    def result_statuses
      { started: "Started", completed: "Completed" }
    end

    def get_result_by_id(id : Int32)
      Result.find_by ({:id => id})
    end

    def get_user_results(user : Int32)
      Result.where { c("user_id") == user }.to_a
    end

    def has_result_by_id(id : Int32)
      Result.where { c("id") == id }.exists?
    end

    def set_result_completed(id : Int32, date_completed : String)
      statuses = result_statuses
      res = get_result_by_id(id)
      if (!res)
        raise NoSuchRecordException.new(id)
      end

      res.status = statuses[:completed]
      res.date_completed = date_completed
      res.save
    end

```

Рисунок 9. results_service.cr

3.3.4 Приклад контроллера

У контроллерах прописується логіка взаємодії з HTTP-запитами користувача, повертаються дані у форматі JSON, або помилка з відповідним HTTP кодом. У використаному фреймворку Kemal для створення REST API ендпоінту використовуються методи, які відповідають методам HTTP-запитів, наприклад get, post, patch, put, delete та інші. Разом з ними вказується шлях для HTTP-запиту та надається доступ до параметрів запиту через відповідну змінну.

Для прикладу розглянемо question_results_controller(Рисунок 9). У ньому вказується два ендпоінти:

- get "/questionResults/:quiz_id" – отримати всі результати відповідей на питання за ідентифікатором результату тесту
- post "/questionResults/:quiz_id" – створити новий результат відповіді на питання за ідентифікатором результату тесту

Спочатку ініціалізуються класи сервісів, які використовує question_results_controller. Для цього викликається метод інжектора залежностей Inject.resolve(Services::QuestionResultsService). В даному випадку ініціалізація могла відбуватися і простим створенням нового об'єкту сервісу, але якщо для сервісу необхідні були б додаткові налаштування або сервіс мав декілька імплементацій, то такий за допомогою даного паттерну ці проблеми вирішуються.

У методі get контроллера зі змінної env отримується інформація про користувача запиту, значення URL параметру quiz_id. Після цього перевіряється чи такий квіз існує за допомогою виклику відповідного методу сервісу. Якщо ні – за допомогою виклику halt зупиняється подальше виконання запиту і повертається HTTP код 400 «Bad Request». Потім перевіряється чи такий користувач може мати доступ до відповідного результату тесту – чи значення поля user_id збігається з ідентифікатором користувача. Після цього отримуються результати відповіді на питання для даного тесту та перетворюються у формат JSON викликом to_json(Рисунок 10).

```
course > back > src > controllers > question_results_controller.cr > ...
1   require "../services/*"
2   require "json"
3   require "../dto/*"
4   require "../jwt/*"
5   require "../config/inject"
6
7   question_results_service = Inject.resolve(Services::QuestionResultsService)
8   results_service = Inject.resolve(Services::ResultsService)
9
10  get "/questionResults/:quiz_id" do |env|
11    user = (env.get "user").as(JWTUserData::User)
12
13    quiz_id = env.params.url["quiz_id"].to_i?
14    if (!quiz_id || !results_service.has_result_by_id(quiz_id))
15      halt env, status_code: 400, response: "No such quiz result"
16    end
17
18    quizResult = results_service.get_result_by_id(quiz_id)
19    if (!quizResult || quizResult.user_id != user.id)
20      halt env, status_code: 403, response: "Forbidden"
21    end
22
23    res = question_results_service.get_results_by_quiz_result_id(quiz_id)
24    res.to_json
25  end
26
27  post "/questionResults/:quiz_id" do |env|
28    user = (env.get "user").as(JWTUserData::User)
29
30    quiz_id = env.params.url["quiz_id"].to_i?
31    if (!quiz_id || !results_service.has_result_by_id(quiz_id))
32      halt env, status_code: 400, response: "No such quiz result"
33    end
34
35    quizResult = results_service.get_result_by_id(quiz_id)
36    if (!quizResult || quizResult.user_id != user.id)
37      halt env, status_code: 403, response: "Forbidden"
```

Рисунок 10. question_results_controller.cr

3.3.5 Авторизація та JWT

Як можна було помітити в описі контроллера, інформацію про користувача ми отримуємо з змінної запиту. Але стандартно там такої інформації не існує. Для того, щоб кожен запит міг отримати інформацію про користувача та для фільтрації не авторизованих користувачів була використана можливість Kemal для створення проміжних функцій обробки запиту (middleware). Для збереження інформації про користувача використовується JSON Web Token. Ендпоінти логіну не мають інформації про користувача з JWT, вони отримують нікнейм та пароль користувача, перевіряють чи вони співпадають зі збереженими у базі значеннями, формують новий JWT токен для користувача, який відправляється у тілі відповіді на запит та зберігається на клієнті. Для використання інших ендпоінтів користувач має бути авторизованим, тому необхідно фільтрувати всі

неавторизовані запити. Для цього створений файл JWTHandler (Рисунок 11), у якому описується middleware, який застосовується для ендпоінтів не пов'язаних з реєстрацією/логіном. Щоб ця логіка не застосовувалась до цих ендпоінтів викликається метод Kemal exclude, у який передається шляхи запитів, з якими middleware не працює. Після цього з HTTP-заголовка 'Authorization' витягується токен та декодується. Логіка кодування/декодування токену та інформація про користувача з токену описується у файлі JWTUserData (Рисунок 12). Для цього використовуються відповідні методи JWT.encode і JWT.decode. Якщо токен не валідний чи його не існує викликається exception, який потім перехоплюється блоком rescue. В цьому блоці користувачу повертається HTTP-код 401 'Unathorized'. Якщо токен валідний, то за допомогою виклику env.set "user", user інформація про користувача зберігається в змінній запиту, а викликом call_next(env) запит переходить до наступного обробника (в даному випадку до кінцевих ендпоінтів). Для того, щоб зареєструвати хендлери при запуску застосунку у файлі back.cr викликається Kemal метод add_handler JWTHandler.new. Також у цьому файлі реєструється тип JWTUserData як тип даних, які зберігаються у контексті запиту, для того, щоб потім отримувати інформацію про користувача як зареєстрованим типом JWTUserData. Це відбувається викликом add_context_storage_type(JWTUserData::User).

```
course > back > src > handlers > jwt_handler.cr > ...
1  require "kemal"
2  require "../jwt/*"
3
4  class JWTHandler < Kemal::Handler
5    exclude ["/auth/*"], "POST"
6
7    def call(env)
8      return call_next(env) if exclude_match?(env)
9
10     if (env.request.method === "OPTIONS")
11       env.response.status_code = 200
12       env.response.headers.add("Access-Control-Allow-Headers", "authorization")
13       return
14     end
15
16     begin
17       auth_header = env.request.headers["Authorization"]
18       if (!auth_header.starts_with?("Bearer "))
19         raise "Incorrect Authorization header"
20       end
21       token = auth_header[7..-1]
22
23       user = JWTUserData.decode_user(token)
24       env.set "user", user
25     rescue ex
26       puts ex.message
27       env.response.status_code = 401
28       return
29     end
30     call_next(env)
31   end
32 end
```

Рисунок 11. jwt_handler.cr

```
course > back > src > jwt > jwt_user_data.cr > ...
1  require "../config/*"
2  require "jwt"
3
4  EXP_TIME = 60 * 60 * 24 # 1 day
5
6  module JWTUserData
7    class User
8      property username : String
9      property id : Int32
10     def initialize(@username : String, @id : Int32)
11     end
12   end
13
14   def self.encode_user(user : User) : String
15     exp = Time.utc.to_unix + EXP_TIME
16     payload = {"username" => user.username, "id" => user.id, "exp" => exp}
17     JWT.encode(payload, Config::SECRET_KEY, JWT::Algorithm::HS256)
18   end
19
20   def self.decode_user(token : String) : User
21     payload, header = JWT.decode(token, Config::SECRET_KEY, JWT::Algorithm::HS256)
22     User.new(payload["username"].as_s, payload["id"].as_i)
23   end
24 end
```

Рисунок 12, jwt_user_data.cr

3.3.6 Розробка API тестів

Для кожного контроллера були прописані автоматизовані тести. Загальна структура тестів наступна – у блоках before і after, які виконуються після кожного тесту була прописана логіка видалення існуючих даних з бази. Так, як при цих тестах використовується реальна база даних, була створена тестова база даних з такою ж структурою як і база даних додатку. Перед виконанням тестів зміна середовища "KEMAL_ENV" змінюється на значення “test”, а у файлі конфігу прописується, що якщо середовище тестове, то для підключення до бази даних потрібно використати значення “TEST_DB_CONNECTION” з файлу .env. Таким чином досягається тестування на реальній базі даних без зміни існуючої.

Також у блоці before відбувається виклик ‘MassSpec.configure do

headers({"Authorization" => "Bearer #{token}"})

end’ для того, щоб встановити токен авторизації для подальших тестів.

У кожному файлі для кожного ендпоінту прописується блок describe з назвою цього ендпоінта. В блоках describe були створені блоки it у яких вже відбувається виконання індивідуальних тест-кейсів. Кожен кейс має на меті перевірити один певний функціонал даного едпоінту. Для цього викликаються методи post/get/put бібліотеки MassSpec з параметрами шляху та тіла запиту(якщо необхідно). Ці методи надсилають HTTP-запит на створений REST API. Після виконання стають доступні геттери status_code та json_body для перевірки відповіді. Для валідації використовуються методи should eq() для строгої відповідності та should contain() для перевірки, що очікувані поля у JSON тілі відповіді існують (Рисунок 13).

```
course > back > spec > question_results_spec.cr > ...
45
46  describe "POST /questionResults/:quiz_id" do
47    it "Throws 400 BAD_REQUEST if no such result exists" do
48      post "/questionResults/#{-1}", body: question_result.to_json
49
50      status_code.should eq(400)
51    end
52
53    it "Throws 400 BAD_REQUEST if request dto is invalid" do
54      dto = { "score" => 1 }.to_json
55
56      post "/questionResults/#{test_result_id}", body: dto
57
58      status_code.should eq(400)
59    end
60
61    it "Creates a question result with given params" do
62      post "/questionResults/#{test_result_id}", body: question_result.to_json
63
64      status_code.should eq(200)
65      json_body.should contain(question_result)
66
67      get "/questionResults/#{test_result_id}"
68
69      status_code.should eq(200)
70      assert_equal 1, json_body.size
71
72      json_body[0].should contain(question_result)
73    end
74  end
75
76  describe "GET /questionResults/:quiz_id" do
77    it "Returns empty array if no results were created" do
78      get "/questionResults/#{test_result_id}"
79
80      status_code.should eq(200)
81      json_body.should eq([] of JSON::Any)
```

Рисунок 13. question_results_spec.cr

3.3.7 Можливості метапрограмування

Crystal, як і Ruby надає великі можливості для метапрограмування. Як приклад застосування цього було створено клас-обгортку JsonHash над класом JSON::Any. У розробленому класі було використано макрос method_missing, за допомогою якого стало можливо отримувати дані з полів JSON використовуючи назви ключів як геттери. Це в подальшому використовувалось в тестах для більш лаконічного отримування даних з JSON(Рисунок 14).

```
14  class JsonHash
15    getter obj : JSON::Any
16
17    def initialize(json : JSON::Any)
18      @obj = json
19    end
20
21    macro method_missing(key)
22      def {{ key.id }}
23        obj[{{ key.id.stringify }}]
24      end
25    end
26
27    def ==(other)
28      obj == other
29    end
30  end
31
```

Рисунок 14. JsonHash

3.4 Результати

В результаті була розроблена REST API з відповідними ендпоінтами:

- POST /auth/register – реєстрація користувача за переданими параметрами нікнейму, пошти та пароля. Повертається токен авторизації
- POST /auth/login – логін користувача за переданими параметрами нікнейму та пароля. Повертається токен авторизації
- GET /results/user – отримати всі результати тестів користувача
- POST /results – створення спроби тесту користувача.
- PUT /completed/:id – переведення тесту у закінчений стан за ідентифікатором
- GET /questionResults/:quiz_id – отримати всі результати відповідей користувача на питання певного тесту
- POST /questionResults/:quiz_id – створення результату відповіді користувача на питання певного тесту.

Результати використання даних ендпоінтів при використанні фронтенд React застосунку(Рисунок 15):

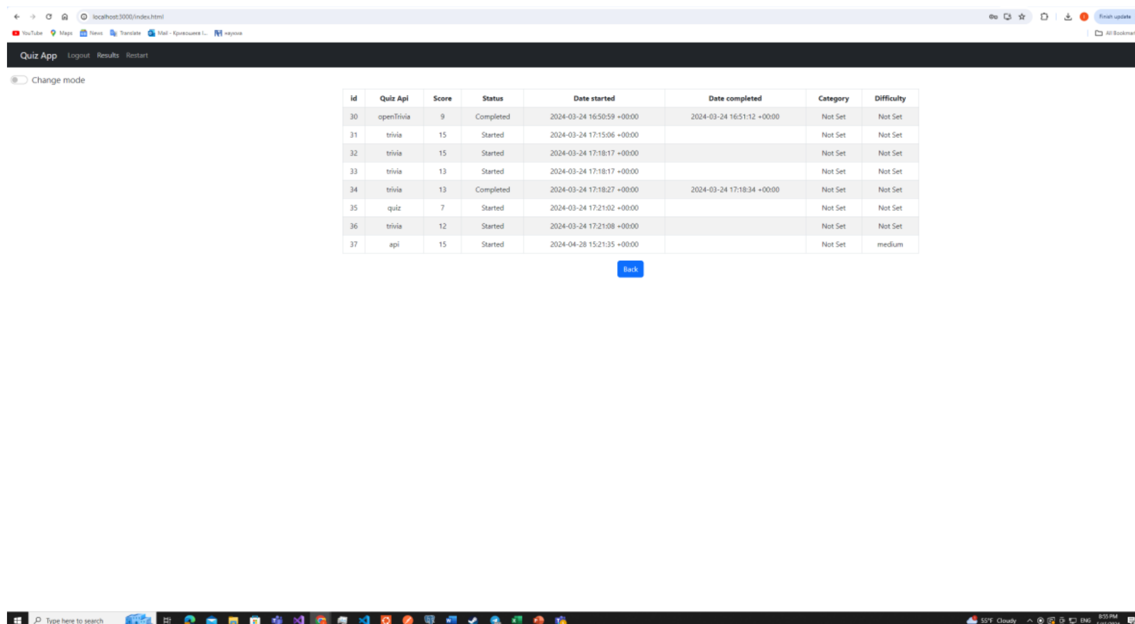

Рисунок 15. UI додатку

Під час розробки були створені API тести, які дуже допомагали швидко перевірити, чи внесені зміни у коді не спричинили небажаних змін у логіці роботи додатку. Приклад успішного виконання тестів(Рисунок 16):

```
d\"", args: ["user", "user@gmail.com", "$2a$10$l0W9mROaFZco3djbbiZTjuK0w1cFNEALMIkdrQmxOIiKh2Y9ZE6HC"], time: 299.8
2024-05-14T20:26:40.495531Z  DEBUG - db:  -- query: "COMMIT", time: 0.1
335
eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJ1c2VybmFtZSI6InVzZXIiLCJpZCI6MzM1LCJleHAiOjE3MTU4MDQ4MDB9.tu-qqp5NbEgIRi3pro7-sl41ObmXmoRBibDyvwrjb30
2024-05-14 20:26:40 UTC 200 POST /auth/register 484.1ms
2024-05-14T20:26:40.502080Z  DEBUG - db:  -- query: "START", time: 0.2
2024-05-14T20:26:40.502483Z  DEBUG - db:  -- query: "INSERT INTO \"quiz_results\"(\"score\", \"quiz_api\", \"difficulty\", \"category\", \"user_id\"
, \"status\", \"date_started\", \"date_completed\") VALUES ($1, $2, $3, $4, $5, $6, $7, $8)  RETURNING \"id\"", args: [10, "Quiz API", "easy", "Scie
nce", 335, "Started", "2024-05-14 20:26:40 +00:00", nil], time: 373.3
2024-05-14T20:26:40.502485Z  DEBUG - db:  -- query: "COMMIT", time: 0.1
2024-05-14 20:26:40 UTC 200 POST /results 2.27ms
2024-05-14T20:26:40.504677Z  DEBUG - db:  -- query: "SELECT \"quiz_results\".* FROM \"quiz_results\" WHERE \"quiz_results\".\"id\" = $1 LIMIT 1 ", a
rgs: [254], time: 348.8
2024-05-14T20:26:40.504908Z  DEBUG - db:  -- query: "SELECT \"quiz_results\".* FROM \"quiz_results\" WHERE \"quiz_results\".\"id\" = $1 LIMIT 1 ", a
rgs: [254], time: 204.5
2024-05-14T20:26:40.505094Z  DEBUG - db:  -- query: "START", time: 0.2
2024-05-14T20:26:40.505427Z  DEBUG - db:  -- query: "UPDATE \"quiz_results\" SET \"status\"= $1, \"date_completed\"= $2 WHERE \"id\" = $3", args: ["
Completed", "2024-05-14 20:26:40 +00:00", 254], time: 313.4
2024-05-14T20:26:40.505429Z  DEBUG - db:  -- query: "COMMIT", time: 0.1
2024-05-14 20:26:40 UTC 200 PUT /results/completed/254 4.84ms
2024-05-14 20:26:40 UTC 200 GET /results/user 673.8µs
.2024-05-14T20:26:40.509868Z  DEBUG - db:  -- query: "SELECT \"quiz_results\".* FROM \"quiz_results\" WHERE \"quiz_results\".\"user_id\" = $1 ", arg
s: [335], time: 493.0

18 passed, 3 assertions, 0 failures, 0 errors, 0 pending


Finished in 16.47 seconds
0 examples, 0 failures, 0 errors, 0 pending
```

Рисунок 16. Приклад виконання тестів

Також було проведено тестування продуктивності додатку за допомогою інструменту wrk[20]. За допомогою цього інструменту вдалося знайти проблему з початковим підходом до підключення до бази даних без Jennifer ORM. У початковій реалізації виконувалися запити SQL напряму, що мало би лише покращити продуктивність, але через не налаштований пул підключень до бази даних більшість запитів не були оброблені через таймаут, частину сервер намагався обробити, але не міг і видавав 500 помилки(Рисунок 17).

```
user@DESKTOP-9J3JNF9:~/test/wrk/wrk$ wrk -t12 -c400 -d30s --header "Authorization: Bearer eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJ1c2VybmFtZ
6InRlc3QzNTUiLCJpZCI6OCwiZXhwIjoxNzE0OTI2MDkzfQ.8_DTbsutz4YQLEXfVjV63xCVFNlGDJV_tqwCyy6v2XM" http://127.0.0.1:3000/results/user
Running 30s test @ http://127.0.0.1:3000/results/user
  12 threads and 400 connections
  Thread Stats   Avg      Stdev     Max   +/- Stdev
    Latency   569.44ms  391.43ms   2.00s    69.37%
    Req/Sec    17.33     19.43   230.00     89.50%
  4071 requests in 30.09s, 98.83MB read
  Socket errors: connect 0, read 0, write 0, timeout 2014
  Non-2xx or 3xx responses: 884
Requests/sec:    135.29
Transfer/sec:      3.28MB
```

Рисунок 17. Невдалий тест на продуктивність

Через це було вирішено використовувати бібліотеку Jennifer. Згідно до нових тестувань продуктивності сервер зміг успішно захендлити 15459 реквестів за 30 секунд з середнім часом на виконання 218мс(Рисунок 18).

```
user@DESKTOP-9J3JNF9:~/test$ wrk -t2 -c100 -d30s --latency --header "Authorization: Bearer eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJ1c2VybmFtZSI6InRl
c3QzNTUiLCJpZCI6OCwiZXhwIjoxNzE1ODgzODYwfQ.Ei0CXnii7xRcNpXrB0_f5exEdI9DdHXC4M4mu8EBiIU" http://127.0.0.1:3000/results/user
Running 30s test @ http://127.0.0.1:3000/results/user
  2 threads and 100 connections
  Thread Stats   Avg      Stdev     Max   +/- Stdev
    Latency   218.17ms  171.84ms   1.98s    82.93%
    Req/Sec   260.25     68.05   555.00     74.79%
  Latency Distribution
     50%  121.57ms
     75%  325.80ms
     90%  517.46ms
     99%  558.40ms
  15459 requests in 30.03s, 25.20MB read
Requests/sec:    514.72
Transfer/sec:    859.04KB
user@DESKTOP-9J3JNF9:~/test$
```

Рисунок 18. Performance test

Також за допомогою команди ‘shards build’ було створено єдиний бінарний executable файл. Це дозволяє дуже легко розгортати сервер на віддаленому середовищі, бо не потрібно додатково встановлювати всі залежності, достатньо лише скопіювати цей executable файл і створити файл .env з відповідними параметрами підключення до бази даних і сервер запускається. Це яскраво контрастує з багатьма іншими фреймворками розробки веб-додатків, у яких процес розгортання на іншому середовищі набагато складніший.

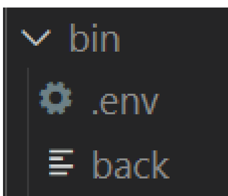

Рисунок 19. Бінарний файл

```
user@DESKTOP-9J3JNF9:~/test/course/back/bin$ ./back
[development] Kemal is ready to lead at http://0.0.0.0:3000
```

Рисунок 20.виконання бінарного файлу

Висновок

Під час виконання роботи спочатку було розглянуто найбільш популярні існуючі фреймворки та екосистеми для створення REST API сервісів. Потім було проаналізовано можливості мови Crystal, архітектурні паттерни та підходи до розробки REST API сервісів такі як Controller-Service-Repository, Repository Design Pattern та Dependency Injection. Також було показано які бібліотеки та фреймворки існують у екосистемі Crystal для реалізації даних шаблонів проектування. Іще було проаналізовано можливості Crystal для автоматизованого тестування коду.

У практичній частині роботи було реалізовано REST API застосунок з використанням відповідних паттернів на тему платформи для онлайн-квізів. Було використано фреймворк Kemal, авторизацію на основі JWT токенів, бібліотеку Jennifer для підключення до бази даних. Також були розроблені автоматизовані API тести для даного застосунку. Було продемонстровано роботу даного застосунку, виконання тестів, проведено аналіз продуктивності додатку.

З результатів виконання роботи можна зробити висновок, що хоча Crystal ще є досить молодою мовою програмування з не повністю розвиненою екосистемою, він добре підходить для ефективної розробки REST API застосунків.

Список джерел

1. Formal specification of the Crystal language [Електронний ресурс]. – Режим доступу: https://crystal-lang.org/reference/1.12/syntax_and_semantics/index.html
2. REST API Architectural Constraints [Електронний ресурс]. – Режим доступу: https://www.geeksforgeeks.org/rest-api-architectural-constraints/
3. Controller-Service-Repository [Електронний ресурс]. – Режим доступу: https://tom-collings.medium.com/controller-service-repository-16e29a4684e5
4. Kemal Guide [Електронний ресурс]. – Режим доступу: https://kemalcr.com/guide/
5. Repository Design Pattern [Електронний ресурс]. – Режим доступу: https://www.geeksforgeeks.org/repository-design-pattern/
6. Jennifer documentation [Електронний ресурс]. – Режим доступу: https://github.com/imdrasil/jennifer.cr
7. Dependency Injection [Електронний ресурс]. – Режим доступу: https://stackify.com/dependency-injection/
8. Crystal-DI documentation [Електронний ресурс]. – Режим доступу: https://github.com/vladislav-yashin/crystal-di
9. Spring Boot - Introduction [Електронний ресурс]. – Режим доступу: https://www.tutorialspoint.com/spring_boot/spring_boot_introduction.htm
10. Python Web Development With Django [Електронний ресурс]. – Режим доступу: https://www.geeksforgeeks.org/python-web-development-django/
11. Laravel documentation [Електронний ресурс]. – Режим доступу: https://laravel.com/docs/11.x
12. Популярність бекенд фреймворків [Електронний ресурс]. – Режим доступу: https://statisticsanddata.org/data/most-popular-backend-frameworks-2012-2023/

13. Getting Started with Rails [Електронний ресурс]. – Режим доступу: https://guides.rubyonrails.org/getting_started.html#what-is-rails-questionmark
14. Introduction to JSON Web Tokens [Електронний ресурс]. – Режим доступу: https://jwt.io/introduction
15. Crystal JWT documentation [Електронний ресурс]. – Режим доступу: https://github.com/crystal-community/jwt
16. Testing Crystal Code [Електронний ресурс]. – Режим доступу: https://crystal-lang.org/reference/1.12/guides/testing.html
17. Mass Spec Documentation [Електронний ресурс]. – Режим доступу: https://github.com/c910335/mass-spec
18. Crystal-pg Documentation [Електронний ресурс]. – Режим доступу: https://github.com/will/crystal-pg
19. cr-dotenv Documentation [Електронний ресурс]. – Режим доступу: https://github.com/gdotdesign/cr-dotenv
20. wrk Github Page [Електронний ресурс]. – Режим доступу: https://github.com/wg/wrk