МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
Національний університет «Острозька академія»
Економічний факультет
Кафедра економіко-математичного моделювання та інформаційних технологій

КВАЛІФІКАЦІЙНА РОБОТА/ПРОЄКТ

на здобуття освітнього ступеня бакалавра

на тему: «**Розробка серверної частини веб-застосунку для допомоги у формуванні та позбутті звичок з елементами гейміфікації»**

Виконала: студентка 4 курсу, групи КН-41
першого (бакалаврського) рівня вищої освіти
спеціальності 122 Комп'ютерні науки
освітньо-професійної програми «Комп'ютерні науки»
Лобода Антоніна Олександрівна

Керівник: *старший викладач,*
Красюк Богдан Віталійович

Рецензент: *Front-end Developer "DOODLE", LLC*
Місай Володимир Віталійович

РОБОТА ДОПУЩЕНА ДО ЗАХИСТУ

Завідувач кафедри економіко-математичного моделювання та інформаційних технологій _____ (проф., д.е.н. Кривицька О.Р.)

Протокол № 11 від «18» травня 2023 р.

Острог, 2023

Міністерство освіти і науки України
Національний університет «Острозька академія»

Факультет: *економічний*
Кафедра: *економіко-математичного моделювання та інформаційних технологій*
Спеціальність: 122 Комп'ютерні науки
Освітньо-професійна програма: Комп'ютерні науки

ЗАТВЕРДЖУЮ
Завідувач кафедри економіко-математичного моделювання
та інформаційних технологій

_____ Ольга КРИВИЦЬКА
«____» _____20__ р.

З А В Д А Н Н Я
на кваліфікаційну роботу/проєкт студентки

Лободи Антоніни Олександрівни

1. Тема роботи: "Розробка серверної частини веб-застосунку для допомоги у формуванні та позбутті звичок з елементами гейміфікації"
керівник проєкту Красюк Богдан Віталійович, старший викладач кафедри ЕММІТ.

Затверджено наказом ректора НаУОА від 31 жовтня 2022 року №77.

2. Термін здачі студенткою закінченої роботи/проєкту: 31 травня 2023 року.

3. Вихідні дані до роботи/проєкту: у своїй роботі я використовувала такі технології: ASP.Net Core, мова програмування C#, MS SQLServer, відкритий стандарт безпечної передачі інформації JWT, ORM-фреймворк для роботи з даними Entity Framework.

4. Перелік завдань, які належить виконати: провести аналіз предметної області; спроектувати роботу програми, базу даних та архітектуру рішення; обрати технології та реалізувати продукт, а саме: розробити серверну частину додатку, зокрема функції для роботи зі звичками, відслідковування прогресу, перегляду статистики та нарахування балів; розробити API на основі цих функцій.

5. Перелік графічного матеріалу: рисунки, таблиці.

6. Консультанти розділів роботи:

Розділ	Прізвище, ініціали та посада Консультанта	Підпис, дата	
		Завдання видав	Завдання прийняв
1	Красюк Б.В.	01.12.2022р.	01.12.2022р.
2	Красюк Б.В.	01.12.2022р.	01.12.2022р.
3	Красюк Б.В.	01.12.2022р.	01.12.2022р.

7. Дата видачі завдання: 01.12.2022 р.

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів кваліфікаційної роботи	Строк виконання етапів	Примітка
1.	Затвердження теми роботи/проєкту	до 31.10.2022 р.	
2.	Постановка технічного завдання	до 01.12.2022 р.	
3.	Розробка архітектури проєкту, погодження функціоналу	до 15.01.2023 р.	
4.	Розробка бази даних	до 01.02.2023 р.	
5.	Розробка сервісів та API для створення, редагування, перегляду та видалення звичок	до 15.03.2023 р.	
6.	Розробка сервісів та API для виконання щоденних завдань	до 15.04.2023 р.	
7.	Розробка сервісів та API для відслідковування прогресу та гейміфікації	до 01.05.2023 р.	
8.	Тестування системи	до 20.05.2023 р.	
9.	Попередній захист кваліфікаційної роботи/проєкту	до 31.05.2023 р.	
10.	Здача кваліфікаційної роботи/проєкту на кафедру	31.05.2023 р.	

Студент: _____ Антоніна ЛОБОДА

Керівник кваліфікаційної роботи: _____ Богдан КРАСЮК

АНОТАЦІЯ
кваліфікаційної роботи/проєкту
на здобуття освітнього ступеня бакалавра

Тема: *“Розробка серверної частини веб-застосунку для допомоги у формуванні та позбутті звичок з елементами гейміфікації”.*

Автор: *Лобода Антоніна Олександрівна*

Науковий керівник: *Красюк Богдан Віталійович, викладач кафедри економіко-математичного моделювання та інформаційних технологій.*

Захищена «……»…… 20__ року.

Пояснювальна записка до кваліфікаційної роботи: *81 с., 18 рис., 1 табл., 13 джерел.*

Ключові слова: *звички, челенджі, трекер, веб-застосунок, база даних, серверна частина.*

Короткий зміст праці:

Метою кваліфікаційної роботи/проєкту була розробка алгоритму для серверної частини трекеру набування та позбуття звичок “Habit-Rabbit” і реалізація його у вигляді програмного забезпечення. Цей проєкт реалізує собою можливість створення, перегляду, редагування та видалення звичок, проходження завдань на щодень з елементами гейміфікації та функцією відслідковування прогресу. Користувачами додатку є особи, які мають на меті зробити своє життя кращим, шляхом формування чи позбуття звичок.

Для реалізації серверної частини даного проекту було використано середовище розробки JetBrains Rider. Основною технологією, на якій ґрунтується проект, є фреймворк ASP.NET Core. Для забезпечення доступу до даних використовувався ORM-фреймворк - Entity Framework Core. У процесі розробки було застосовано кілька корисних бібліотек та інструментів, таких як Swagger, Automapper та Json Web Token. Контроль версій здійснювався за допомогою системи Git та платформи Github.

Для проектування інтерфейсу було використано графічний редактор Figma. В організації командної роботи допоміг інструмент планування задач ClickUp.

The purpose of the qualification work/project was to develop an algorithm for the server side of the Habit-Rabbit habit tracker and create a corresponding software implementation. This project enables users to create, view, edit, and delete habits, complete daily tasks with gamification elements, and track their progress. The target audience for this application is individuals seeking to improve their lives by establishing or breaking habits.

The server side of the project was implemented using the JetBrains Rider development environment. The ASP.NET Core framework served as the foundation for this project. To facilitate data access, the Entity Framework Core ORM framework was utilized. Throughout the development process, various valuable libraries and tools were employed, including Swagger, Automapper, and Json Web Token. Version control was managed using the Git system and the Github platform.

For interface design, the Figma graphic editor was utilized. In addition, the ClickUp task scheduling tool played a crucial role in organizing team collaboration.

ЗМІСТ

ВСТУП 7
РОЗДІЛ 1. ЗАГАЛЬНІ ПОЛОЖЕННЯ 9
1.1. Опис існуючих рішень 9
1.2. Постановка задачі 12
1.3. Організація командної роботи 13
Висновки до розділу 1 17
РОЗДІЛ 2. ІНФОРМАЦІЙНЕ ЗАБЕЗПЕЧЕННЯ 19
2.1. Проектування системи 19
2.1.1. Розробка UML діаграм 19
2.1.2. Проектування бази даних 23
2.2. Архітектура рішення 28
2.2.1. Чиста архітектура 28
2.2.2. Переваги використання чистої архітектури 30
2.2.3. Реалізація чистої архітектури у проєкті 31
2.2.4. Патерн Репозиторій 34
Висновки до розділу 2 36
РОЗДІЛ 3. ПРОГРАМНЕ ТА ТЕХНІЧНЕ ЗАБЕЗПЕЧЕННЯ 38
3.1. Засоби розробки 38
3.1.1. Опис програмного забезпечення 38
3.1.2. Аналіз стеку технологій 42
3.2. Опис програмної реалізації 49
3.2.1. Автентифікація та авторизація 49
3.2.2. Опис бізнес-процесів 57
Висновки до розділу 3 60
ВИСНОВОК 62
СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ 65
ДОДАТОК А 67
ДОДАТОК Б 71
ДОДАТОК В 75

ВСТУП

Понад 40 відсотків дій, які людина виконуєте щодня, - це не рішення, а звички. Звички формують наше життя значно більше, ніж можна собі уявити. Людський мозок здатен ігнорувати все навколо, включаючи здоровий глузд, коли існує певна вироблена звичка.

Звички – це ті дрібниці, які допомагають нам справлятися з повсякденними завданнями та дозволяють зосередитися на більш важливих справах. Вони можуть бути як позитивними, так і негативними, і часто визначають нашу якість життя. Від щоденної кави на роботі до чищення зубів перед сном, звички займають вагому частину нашого життя.

Деякі звички, такі як регулярна фізична активність та здорове харчування, можуть мати позитивний вплив на наше фізичне та психічне здоров'я. Але існують і такі, які завдають шкоди, зокрема зловживання алкоголем чи куріння. Проте, доклавши трохи зусиль, можна змінити негативні звички та створити нові, корисні.

Формування звичок - це процес, що забезпечує автоматизацію поведінки. Звички можуть формуватися без наміру людини їх набувати, але їх також можна свідомо культивувати або позбуватися. Зазвичай старі звички тримаються міцно, а формування здорових звичок може бути дуже складним процесом. Проте, повторення дій може сприяти створенню нових звичок та збереженню їх у майбутньому. А старих звичок, які шкодять здоров'ю та благополуччю, можна позбутися, якщо підійти до цього розумно та з достатньою рішучістю.

Спроби сформувати нову звичку без необхідних інструментів можна порівняти зі спробами займатися сільським господарством без знарядь праці. Хоча це можливо, такий підхід є надзвичайно тяжким та неефективним. Інструменти, такі як плани, нагадування та підтримка соціального середовища, можуть значно полегшити процес формування нової звички. Саме таким інструментом і покликаний стати веб-додаток Habit-Rabbit.

Існує декілька популярних способів відстеження прогресу у формуванні звичок. Серед них - ведення календаря з відмітками про виконання завдань на аркуші паперу або в блокноті. Цей спосіб доволі поширений та має свої переваги. Такий календар

можна тримати на видному місці - завжди «перед очима», замість звичайної «галочки» чи хрестика можна клеїти наклейки, малювати і робити процес відстеження яскравим та приємним.

Оскільки цифровізація продовжує розширювати свій вплив на всі сфери нашого життя, популярність набирають трекери звичок саме у вигляді додатків, яким поступається звичний спосіб планування на папері. Використання таких програм дозволяє відслідковувати свій прогрес у будь-який час та у будь-якому місці. А процес створення трекеру більше не займає багато часу – можна в декілька кліків створити трекер нової звички і почати її формування. Це зручно, бо на малювання чи друк підходящого шаблону у паперовому варіанті потрібно значно більше часу та зусиль.

Більшість застосунків створені для використання на Android чи iOS. І, безперечно, це дуже зручно, адже сучасна людина має із собою телефон практично завжди і всюди. Проте, проєкт Habit-Rabbit розроблений для використання здебільшого на комп'ютері чи ноутбуці, але за потреби ним можна користуватися через і на телефоні. Хоч телефон і став незмінним помічником у нашому житті, надмірне його використання може заважати концентруватися на важливих справах/роботі. Таким чином, застосунок більше підходить для людей, які працюють за комп'ютером або ж використовують його впродовж дня. З веб-додатком Habit-Rabbit не потрібно відволікатися на телефон. Замість цього можна закріпити вкладку браузера із нашим трекером і, не відриваючись від робочого процесу, формувати свої звички.

Мета проекту – розробка та тестування трекеру звичок.

Задачі проекту:

- аналіз предметної області;
- проектування роботи програми та архітектури рішення;
- вибір технологій та їх використання для створення продукту.

Об'єкт дослідження: серверна частина веб-застосунку для допомоги у формуванні та позбутті звичок з елементами гейміфікації

Предметом дослідження є технології: веб-фреймворк ASP.Net Core, база даних MS SQLServer, мова програмування C#.

РОЗДІЛ 1
ЗАГАЛЬНІ ПОЛОЖЕННЯ

1.1. Опис існуючих рішень

Паперовий формат

В інтернеті можна знайти безліч шаблонів та ідей (Рис. 1.1.1), які можна кастомізувати, роздрукувати, або ж використати для створення власного трекеру звичок на папері.

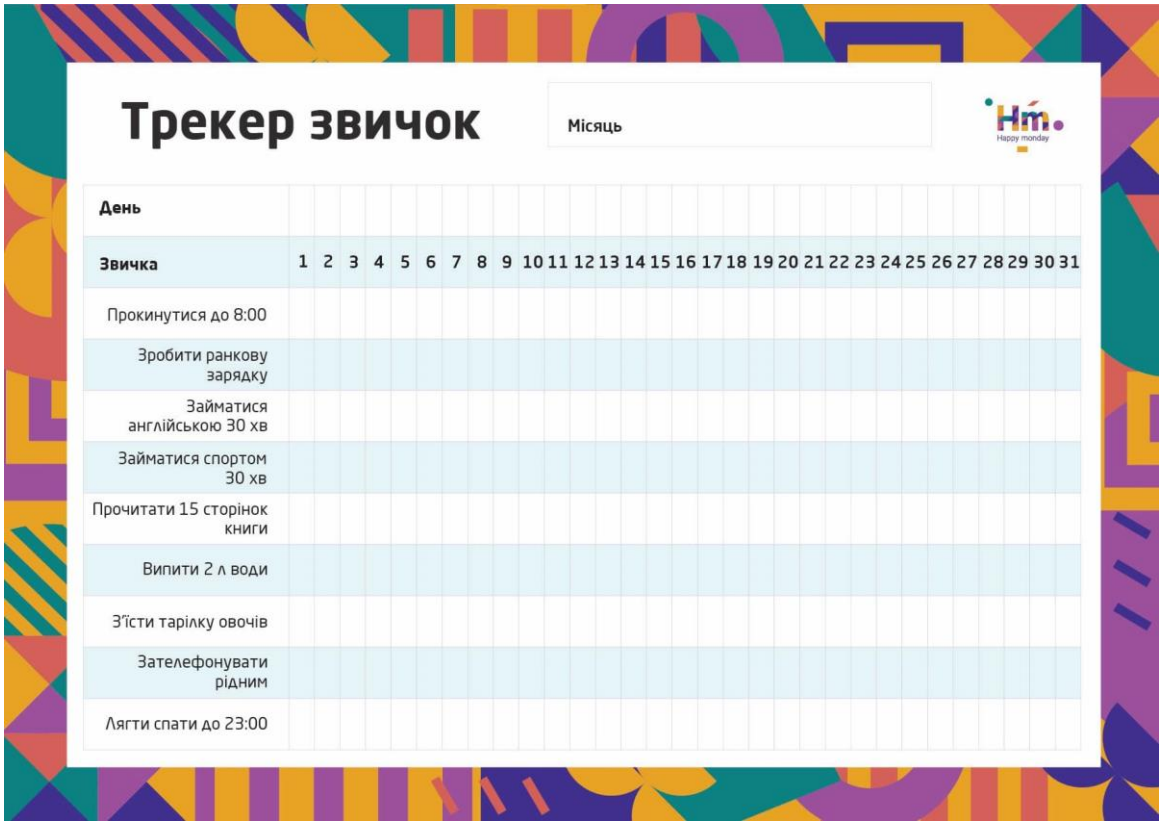

Рис. 1.1.1 Макет трекеру звичок від Happy Monday

Цей спосіб доволі поширений та має свої переваги. Такий трекер можна тримати на видному місці - завжди «перед очима», замість звичайної «галочки» чи хрестика можна клеїти наклейки, малювати і робити процес відстеження яскравим та приємним.

Проте, він має і свої недоліки. Створення такого трекеру власноруч може зайняти багато часу. У цьому випадку із роздруківкою трохи простіше. Окрім того, не дуже зручно носити його завжди з собою, найчастіше такий «календарик» прикріплюється на одному місці, де його буде добре видно. А це виключає можливість

відмітити свій прогрес одразу після пробіжки, нагадати собі завдання на день, їдучи в метро, чи додати нову звичку в момент виникнення ідеї, якщо ви не вдома.

Confetti

Розробники Confetti вірять у те, що святкування кожної маленької перемоги допомагає відчувати себе більш задоволеним та мотивованим. Тому вони створили свій додаток навколо цієї концепції та розробили функції, які допомагають досягати маленьких цілей та отримувати задоволення від кожного досягнення.

Додаток пропонує такий функціонал:

1. Відстеження звичок

- Додавання та керування звичками в одному інтуїтивно зрозумілому інтерфейсі, який запускає конфетті для кожного завершення звички

- Перегляд статистики про звички щотижня, щомісяця, щороку та весь час

- Можливість відстежувати прогрес позбуття негативної звички

2. Ланцюжки регулярності

- Виконані завдання кілька днів поспіль утворюють ланцюжок, що є додатковою мотивацією не розірвати його

Перевагою Confetti є також привабливий інтерфейс.

Застосунок доступний у Android, iOS та веб-версіях, а також як розширення Chrome.

Недоліки:

- Безкоштовно можна відстежувати лише 3 звички
- Немає можливості обрати власне оформлення (колір, іконка)
- Немає можливості вимірювати прогрес у різних одиницях вимірювання (напр. години, метри, літри і т.д.) та відслідковувати виконання завдання частинами
- Інтерфейс недостатньо інтуїтивно зрозумілий

Habitica

Habitica — це гра, з допомогою якої Ви можете поліпшити свої звички у реальному житті (Рис. 1.1.2). Вона „гейміфікує“ Ваше життя, перетворюючи всі Ваші завдання (звички, щоденні справи та завдання) на маленьких потвор, яких вам потрібно

побороти. Чим ліпше Вам це вдаватиметься, тим більше Ви просуватиметеся грою. Кожна ваша помилка у реальному житті відкидатиме назад Вашого персонажа у грі.

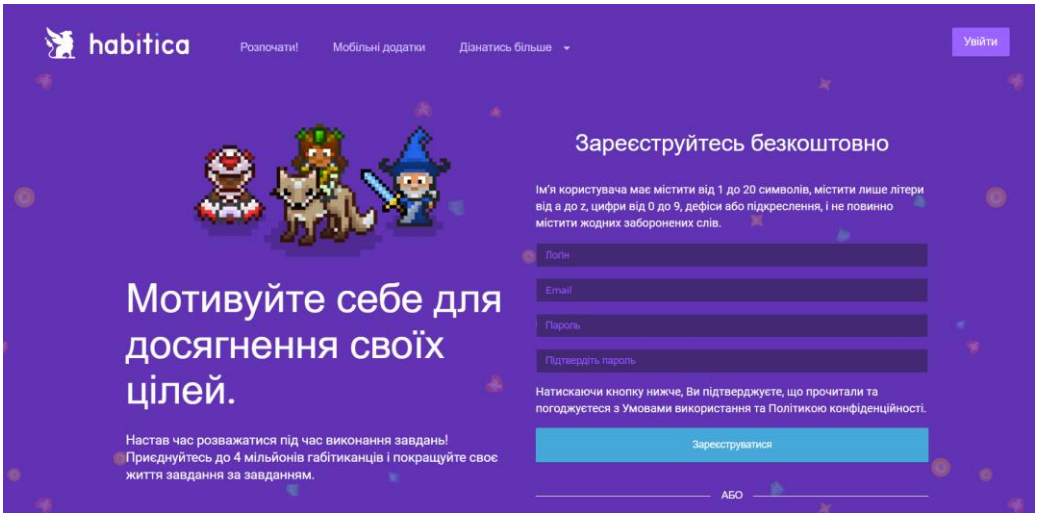

Рис. 1.1.2 Головна сторінка додатку Habitica

Відстежуйте свої звички та цілі: залишайтеся відповідальними, відстежуючи та керуючи своїми звичками, щоденними цілями та списком справ за допомогою простих у використанні мобільних додатків та веб-інтерфейсу Habitica.

Отримуйте нагороди за свої цілі: відзначайте завдання, щоб підвищити рівень свого аватара та розблокувати такі ігрові функції, як броня, таємничі тварини, магічні навички та навіть квести!

Бийтесь проти монстрів разом з друзями: боріться з монстрами разом з іншими габітиканцями! Використовуйте зароблене золото, щоб купувати внутрішньоігрові або власні нагороди, як-от перегляд епізоду улюбленого телешоу. [1]

Однозначно, головною перевагою Habitica є гейміфікація та безкоштовний базовий функціонал. Застосунок створений як повноцінна і дуже добре продумана гра. В першу чергу, це веб-додаток, але є також версії для Android та iOS.

Розробниками передбачено можливість формувати звички у командах разом із друзями та однодумцями. Щоправда, ця функція є платною.

Недоліком може бути складність інтерфейсу. Застосунок не підійде для тих, хто шукає простий та інтуїтивно зрозумілий трекер звичок лише з необхідним функціоналом.

Habitify

Habitify - це прекрасний додаток з простим, але гарним інтерфейсом, який приверне увагу користувачів (Рис. 1.1.3). Він дозволяє легко відстежувати свої звички за допомогою списку на кожен день, дозволяючи відзначати їх, коли вони виконані. Такий простий, але ефективний підхід робить Habitify прекрасним інструментом для формування та підтримки звичок.

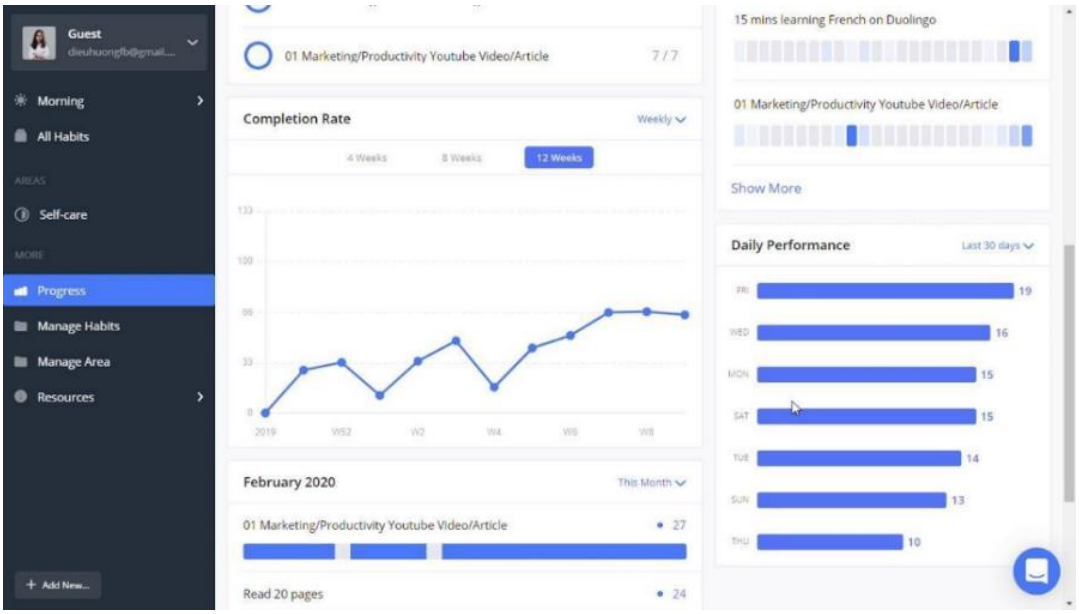

Рис. 1.1.3 Вигляд застосунку Habitify

Крім основних функцій відстеження звичок, Habitify має темну тему, яка дозволяє знизити навантаження на очі, а також безліч крутих графіків і діаграм, що дозволяють триати звички під контролем та стежити за їх прогресом. Однак, ці функції частково є платними, як і можливість відстежувати більше трьох звичок одночасно.

Ціна: безкоштовно обмежений функціонал, преміум-версія за 4,99 доларів США на місяць. Платформи: Android, iOS, Web.

1.2. Постановка задачі

Після проведення аналізу наведених аналогів було прийнято рішення об’єднати переваги кожного з варіантів у створенні власного додатку. Особливостями застосунку Habit-Rabbit було виділено:

- привабливий мінімалістичний інтерфейс
- простота та інтуїтивність використання

- можливість впливати на вигляд свого трекеру (вибір кольору та іконки для кожної звички)
- розширений функціонал для відстеження прогресу. Зокрема, можливість відмічати виконання завдань частинами, вибір одиниць вимірювання, створення підзадач.
- помірна гейміфікація для заохочення та підвищення мотивації
- можливість безкоштовно користуватися всім функціоналом програми

Додаток має містити такий функціонал:

1. Авторизація та автентифікація користувача
2. Створення нової звички (челенджу)
3. Можливість використати готовий шаблон, запропонований програмою для створення звички
4. Можливість бачити завдання на сьогодні та будь-який інший день
5. Функціонал як для формування, так і для позбуття звички
6. Можливість відмітити завдання, як виконане або частково виконане, а також відмінити виконання
7. Перегляд, редагування та видалення звичок
8. Статистика успішності у формуванні звичок. Для кожної звички окремо та для всіх разом, а також для різних проміжків часу.
9. Винагорода за успіхи

1.3. Організація командної роботи

Проект Habit-Rabbit розроблявся у команді з двох людей, і в порівнянні з індивідуальною роботою, колективна має свої особливості. Для успішної спільної роботи над проектом потрібно правильно організувати процес розробки, щоб уникнути непорозумінь, наприклад, при одночасному редагуванні функціоналу декількома людьми або коли частина коду одного члена команди може пошкодити проект десь в іншому місці.

Розробка проекту Habit-Rabbit велася командою з двох людей, і це вимагало особливого підходу до організації процесу розробки. Необхідно було уникнути

можливих непорозумінь, таких як конфлікти при одночасному редагуванні коду декількома людьми або пошкодження деяких частин проекту через внесені зміни в інших його частинах.

Для успішної командної роботи над проектом Habit-Rabbit ми використовували GitHub - платформу для спільної розробки програмного забезпечення. Завдяки GitHub ми змогли забезпечити ефективну співпрацю та уникнути можливих конфліктів при редагуванні коду та інших файлів проекту.

Основні принципи, яких ми дотримувалися при командній розробці проєкту:

1. Контроль версій: ми використовували систему контролю версій Git, яка дозволяла нам зберігати різні версії коду та відслідковувати всі зміни, внесені в проект.

2. Розподіл завдань: ми створили дошку завдань у ClickUp, які необхідно було виконати, і розподілили їх між усіма членами команди. Кожен член команди був відповідальний за виконання своїх завдань.

3. Код-ревю: ми проводили код-ревю, щоб перевірити код кожного члена команди та визначити можливі проблеми або помилки. Це допомогло нам вчасно виявляти та виправляти проблеми, що покращувало якість проекту в цілому.

Використання системи контролю версій. GitHub надало нам можливість зберігати версії коду та зберігати його історію. Кожен учасник команди може зробити свій внесок у проект, зберігаючи його відокремлено від інших версій, працюючи в різних гілках задля уникнення помилок сумісності коду. Крім того, система контролю версій дозволяє зручно порівнювати та об'єднувати зміни, що зроблені кількома учасниками проекту. Оскільки у проекті були сервер і клієнт, то на GitHub було створено два репозиторії.

Другий важливий елемент - це послідовність написання окремих функцій. Існує багато підходів, таких як Agile, Waterfall та інші, але використання якогось конкретного підходу було недоречним через малу кількість людей у команді та неможливість розпланувати, скільки часу вдасться приділяти саме кваліфікаційній роботі посеред навчального процесу. Тому наша команда вирішила розділити проект на малі частини - підзадачі, і поступово їх виконувати.

Для того, щоб було легше відслідковувати прогрес та пріоритетність задач, ми створили дошку в ClickUp (Рис. 2.1).

ClickUp - мінімалістичний інструмент управління проектами, що є прекрасною альтернативою більш популярній Jira. Один з головних факторів, який зробив ClickUp нашим вибором - це приємний та інтуїтивно зрозумілий інтерфейс, а також можливість змінювати тему на світлу або темну. Однак, основною перевагою цього сервісу є його багатий та гнучкий функціонал, що дозволяє створювати та керувати проектами будь-якої складності.

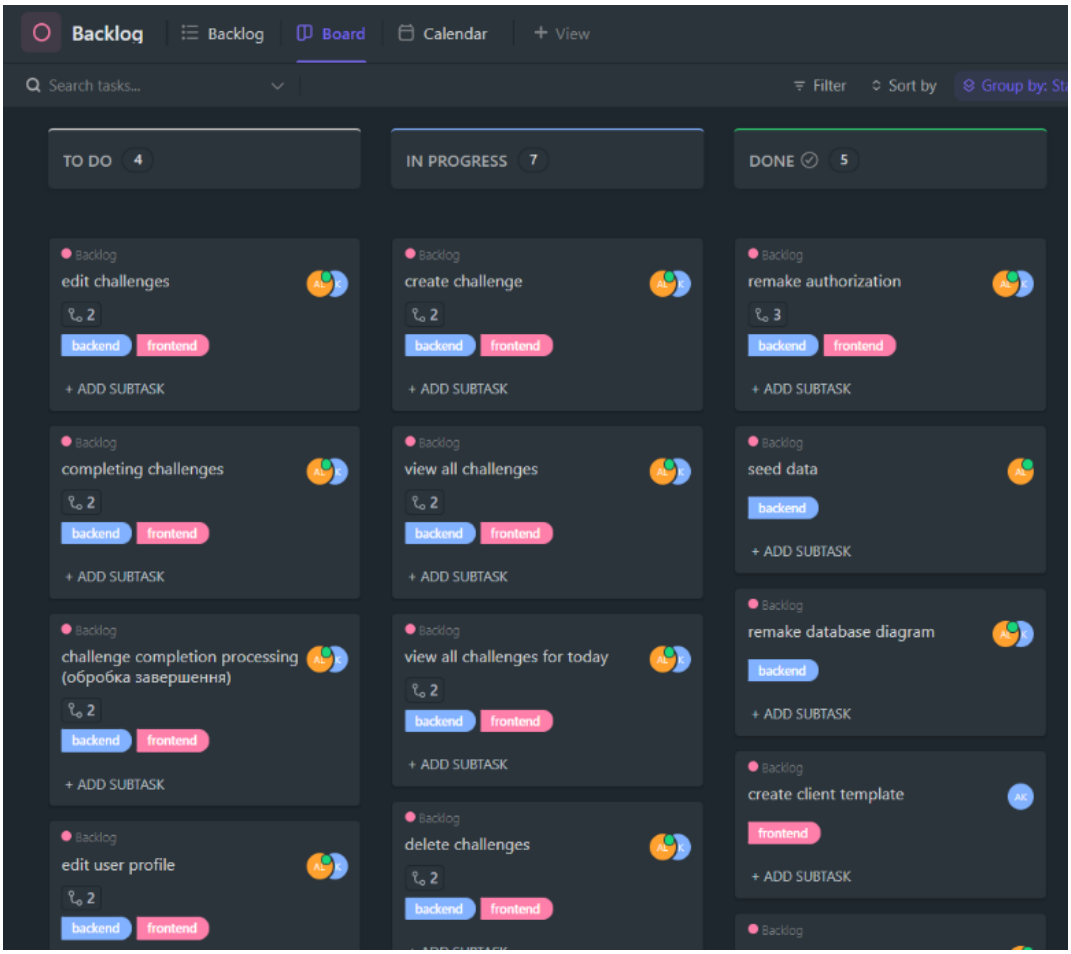

Рис. 2.1. Дошка в ClickUp

На зображенні (Рис. 1.4.1) можна побачити, що в усіх задачах проекту є три стани: ToDo, InProgress та Done. Стан ToDo вказує на те, що задача створена, але ще не розпочата. Стан InProgress показує, що хтось уже почав виконувати задачу, а Done - що задача виконана і є коміт або змерджений pull-request на гітхабі. Цей простий, але ефективний підхід дозволяє легко зорієнтуватися в стані проекту та детальніше

відстежувати робочі процеси. Це забезпечує кращу координацію між членами команди та дозволяє швидше досягти поставлених цілей.

Організація роботи в команді є однією з ключових складових успішного проекту. Незалежно від того, над чим працює, важливо мати чіткий план дій і систему управління проектом. Успішна організація роботи в команді забезпечує зручний робочий процес, ефективне використання ресурсів та досягнення поставлених цілей.

Правильна організація включає в себе розподіл обов'язків, зв'язок між членами команди, планування робочого часу, використання проектних інструментів, моніторинг прогресу та вирішення конфліктних ситуацій. У цьому процесі роль тімліда, який відповідає за управління командою, є визначальною. Тімлід повинен мати здатність керувати командою, визначати пріоритети, контролювати робочий процес і бути лідером, який знаходить рішення в складних ситуаціях.

Оскільки проєкт доволі об’ємний та складний, ми вирішили, що команда потребує наявності такого лідера. Таким чином моїми обов’язками, як тімліда, було:

1. Планувати та контролювати робочі процеси в команді з урахуванням обсягу завдань та строків їх виконання.

2. Визначати пріоритети задач та забезпечувати їх виконання відповідно до термінів.

3. Керувати процесом розробки веб-застосунку, контролювати дотримання технічних вимог та відповідність розробки вимогам проекту.

5. Організовувати регулярну комунікацію з метою обговорення результатів роботи, вирішення поточних питань та планування наступних кроків.

6. Аналізувати виконання планів, виявляти та вирішувати проблеми, які можуть виникнути в процесі розробки.

Висновки до розділу 1

У першому розділі було проведено детальний аналіз існуючих аналогів проекту, який був запланований до реалізації. Для цього були вивчені переваги та недоліки кожної з програм, їх функціональні можливості та особливості використання.

На основі цього аналізу було сформульовано технічне завдання з докладним переліком вимог та задач, які необхідно було виконати під час реалізації проєкту.

Постановка задачі є надзвичайно важливим етапом у розробці будь-якої програми. Він визначає цілі та обмеження проекту, визначає функціональні вимоги та відповідає вимогам користувачів. Правильно сформульована постановка задачі дозволяє ефективно працювати над розробкою та забезпечує якісний результат.

Наш додаток має ряд важливих переваг перед аналогами. По-перше, він має зручний і інтуїтивно зрозумілий інтерфейс, що полегшує роботу з ним. Користувачам не потрібно мати глибоких знань про цільову область програми, щоб ефективно її використовувати.

По-друге, наша програма пропонує широкі можливості налаштування та персоналізації. Користувачі можуть адаптувати його до своїх потреб і вимог, вибравши необхідні функції та параметри. Це дозволяє забезпечити максимальну відповідність програми індивідуальним потребам кожного користувача.

Таким чином, метою кваліфікаційної роботи було визначено розробку веб-застосунку Habit-Rabbit, який має вирізнятися простим, зрозумілим та привабливим дизайном, помірною гейміфікацією процесу набування звички, функціоналом для відслідковування широкого спектру звичок та можливістю спостерігати за своїм прогресом за допомогою візуалізованої аналітики.

Визначено також, що застосунок буде орієнтований на людей, які переважну більшість свого часу проводять за комп’ютером, працюючи чи навчаючись, і не хочуть зайвий раз відволікатись на телефон.

У процесі розробки проекту Habit-Rabbit в команді з двох людей було виявлено, що організація командної роботи має свої особливості порівняно з індивідуальною. Для успішної спільної роботи над проектом важливо було правильно організувати процес розробки, щоб уникнути можливих непорозумінь і проблем.

Одним із ключових аспектів організації командної роботи стало використання системи контролю версій, такої як Git і платформи GitHub. Це дозволяло зберігати різні версії коду, відстежувати зміни та уникати конфліктів при редагуванні коду кількома людьми. Крім того, GitHub дозволив зручно порівнювати та об’єднувати зміни, зроблені різними учасниками проекту.

Панель завдань ClickUp використовувалася для ефективного призначення завдань і відстеження прогресу. Це дозволяло нам створювати підзадачі та розподіляти їх між членами команди. Кожен міг стежити за станом виконання завдань і вчасно їх виконувати.

У правильній організації роботи над проєктом також є незамінною роль тімліда. Тімлід відповідає за керівництво командою, розподіл обов'язків, планування робочого часу, забезпечення ефективного використання ресурсів та вирішення конфліктів. Він здатен приймати рішення в складних ситуаціях та забезпечувати виконання поставлених цілей. Як тімлід, я була відповідальна за планування, контроль робочого процесу, керування командою та розв'язання поточних проблем.

РОЗДІЛ 2
ІНФОРМАЦІЙНЕ ЗАБЕЗПЕЧЕННЯ

2.1. Проектування системи

2.1.1. Розробка UML діаграм

Що таке UML?

Уніфікована мова моделювання (UML) - це мова моделювання загального призначення. Основна мета UML - визначити стандартний спосіб візуалізації того, як була спроектована система. Вона дуже схожа на креслення, що використовуються в інших галузях інженерії.

Замість того, щоб займатись деталями реалізації, UML дозволяє розробникам програмного забезпечення сконцентруватись на візуалізації процесів та роботи систем. Вона є стандартом для створення діаграм та дозволяє розробникам та інженерам мовити однією мовою, щоб зрозуміти та описати структуру системи. Крім того, складання діаграм за допомогою UML допомагає швидко розуміти складні ідеї та структури для інших людей.

Існує декілька типів UML діаграм, серед них:

- діаграма прецендентів (use case diagrams)
- діаграми класів (class diagrams)
- діаграми об'єктів (object diagrams)
- діаграми взаємодії (interaction diagrams), у тому числі
    - діаграми послідовності (sequence diagrams)
    - діаграми кооперації (collaboration diagrams)
- діаграми станів (statechart diagrams)
- діаграми діяльності (activity diagrams)
- діаграми компонентів (component diagrams).
- діаграми розгортання (deployment diagrams)

У цій роботі буде розглянуто лише першу, use-case діаграму, оскільки вона була створена під час розробки проєкту.

Use-case діаграма

Діаграма варіантів використання використовується для представлення динамічної поведінки системи. Вона інкапсулює функціональність системи шляхом включення варіантів використання, акторів та їхніх взаємозв'язків. Вона моделює завдання, сервіси та функції, необхідні системі/підсистемі додатку. Вона відображає високорівневу функціональність системи, а також показує, як користувач працює з системою.

Use-case діаграми складаються з 4 об’єктів: актор, прецедент (варіант використання), система, зв’язок (Рис.3.2). [8]

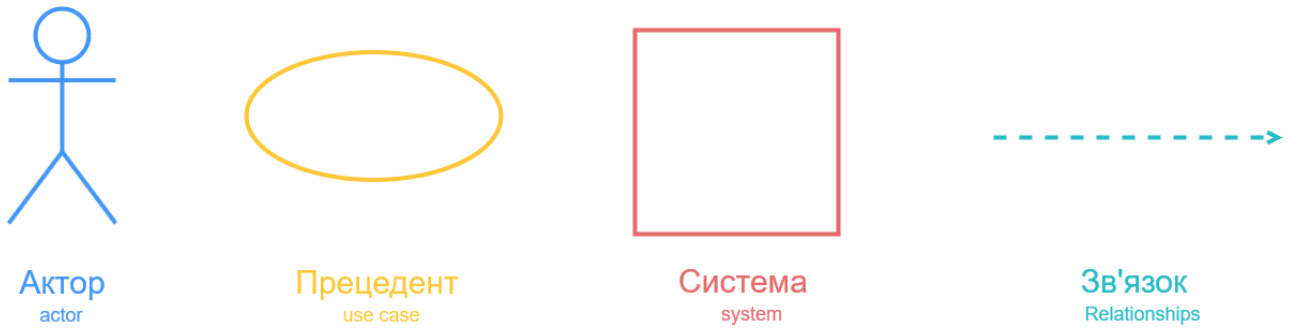

Рис. 3.2. Основні позначення use-case діаграми

Актор у use-case діаграмі є основним елементом, який відіграє певну роль у взаємодії з функціями системи. Актор може представляти собою будь-який об'єкт, який взаємодіє з системою, такий як користувач, клієнт, працівник і т.д. Одна з особливостей актора полягає в тому, що він дуже схожий на користувача і їх часто плутають. Але різниця полягає в тому, що користувач може виконувати різні ролі, у той час, як актор – лише одну. Наприклад: професор може бути викладачем, а також дослідником. Тобто це один користувач, але у діаграмі його можуть позначати два різних актори.

Актор запускає варіанти використання та несе відповідальність за введення вхідних даних в систему. В той же час, актор очікує певних результатів від системи, тобто вихідних даних. Використання акторів у use-case діаграмі дозволяє уникнути непорозумінь між розробниками та користувачами системи, а також сприяє кращому розумінню взаємодії між різними елементами системи.

Прецедент(use-case) - це специфікація функціональної поведінки системи з точки зору її користувачів або зовнішніх систем. Він описує дії, які можуть виконуватися системою з точки зору її користувачів, а також реакції системи на ці дії. Кожен прецедент пов'язаний з одним або декількома акторами, які взаємодіють з системою. Прецедент може описувати як одну конкретну дію, так і цілу послідовність дій. Основна мета прецедента - описати, яку корисну функціональну можливість надає система своїм користувачам.

Система. Об'єкт "система" може бути використаний для визначення меж використання та зображується у вигляді прямокутника. Хоча це не є обов'язковим елементом, воно може бути корисним для візуалізації великих проектів. Наприклад, ви можете спочатку створити всі варіанти використання, а потім використовувати об'єкт "система", щоб визначити область, яку охоплює ваш проект. Ви також можете використовувати цей об'єкт, щоб показати різні області, які охоплюються різними випусками.

Зв'язки. У use-case діаграмі важливо, щоб кожен актор був пов'язаний з принаймні одним прецедентом, але не обов'язково, щоб кожен прецедент був пов'язаний з актором. Часто для зображення зв'язків між ними використовується суцільна лінія. Це допомагає краще візуалізувати взаємодію між акторами та прецедентами в системі.

Зв'язки на діаграмі варіантів використання можна розділити на п'ять типів:

1. Асоціація між актором і варіантом використання, що показує, як актор взаємодіє з системою.
2. Узагальнення актора, що показує загальну роль або категорію актора.
3. Розширення зв'язку між двома варіантами використання, що показує, як один варіант може розширити функціональність іншого.
4. Включення зв'язку між двома варіантами використання, що показує, як один варіант може включати інший.
5. Узагальнення варіанту використання, що показує загальний функціонал або категорію варіанту використання.

Побудова use-case діаграми

Перед початком розробки проєкту Habit-Rabbit, було вирішено створити діаграму прецендентів (Рис. 3.3), щоб узагальнити бачення майбутнього вигляду системи та упевнитись, що всі члени команди розуміють поставлену задачу однаково.

Для побудови діаграми було використано сервіс *drawio.com,* який надає весь необхідний функціонал для побудови різних типів діаграм та фловчартів.

Згідно з діаграмою, веб-застосунок передбачає взаємодію з двома типами користувачів:

- Гість: людина, яка не авторизована в системі.
- Користувач: людина, яка пройшла процес авторизації та автентифікації й успішно увійшла в систему.

Для гостя передбачено лише два варіанти використання системи:

- Реєстрація
- Вхід в систему

Для авторизованого ж користувача відкривається увесь функціонал програми:

- Перегляд персональної інформації, який розширюється можливістю редагування та видаленя профілю
- Створення нового челенджу (звички для набування), який також можна редагувати або видалити
- Перегляд завдань на будь-який конкретний день
    - Можливість відмітити прогрес для кожного завдання
    - Можливість видалити прогрес завдання
- Перегляд статистики проходження викликів та виконання щоденних завдань
- Можливість отримувати винагороду за виконані завдання та завершені челенджі

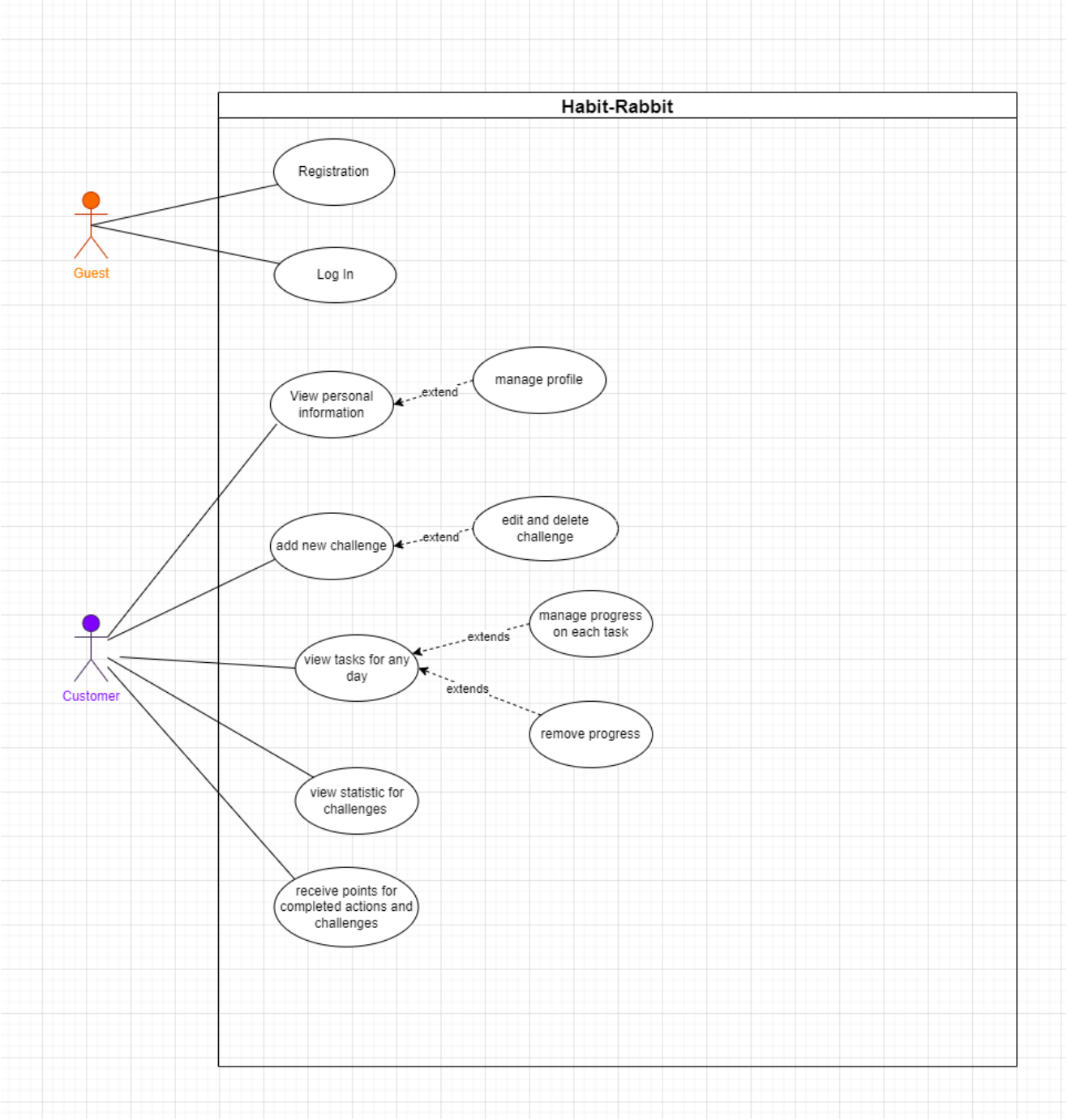

Рис. 3.3. Use-case діаграма проєкту Habit-Rabbit

2.1.2. Проектування бази даних

Бази даних є фундаментом усіх сучасних інформаційних систем. Це організований набір логічно пов'язаних даних. Інформація перетворюється на корисні знання, структуровані та підтримувані відповідно до потреб користувача. Окрім зберігання самих даних, база даних також зберігає зв'язки між елементами даних.

У ширшому розумінні база даних - це інтегрований набір інформації про систему, а також процедури її підтримки та використання. На відміну від електронних таблиць, до сховища даних можуть мати доступ одразу кілька користувачів і додатків.

Наразі існує багато різних типів баз даних, кожен з яких має свої переваги та недоліки. Кожен тип бази даних створює специфічне середовище для зберігання даних і взаємозв'язків між ними. У проєкті було реалізовано реляційний тип.

Реляційні бази даних зберігають дані як табличні структури у вигляді рядків і стовпців з фокусом на узгодженість даних. Цей тип баз даних фокусується на зв'язках між даними, і є найпоширенішим типом баз даних.

Можна виділити 4 основних кроки проектування реляційної бази даних:

1. Визначення мети бази даних (аналіз вимог)
2. Визначення сутностей
3. Визначення атрибутів/полів, що будуть включені до кожної сутності
4. Визначення первинних ключів для кожної сутності
5. Визначення зв'язків між сутностями
6. Вдосконалення структури (нормалізація)
7. Заповнення таблиць початковими даними

Реалізація проектування бази даних

Після аналізу мети та вимог проєкту, було визначено перелік сутностей для бази даних: User, RefreshToken, Challenge, Subtask, DailyTask, ChallengeType, Unit, Visibility.

EntityFramework пропонує декілька підходів до створення бази даних. Зокрема, можна створити базу даних сторонніми засобами (поза кодом), наприклад, за допомогою SQL, а потім засобами EntityFramework інтегрувати її в програмний код. Такий підхід називається Database-First. Також можна створити базу даних на основі побудованої схеми таблиць та зв’язків, це Model-First. Але ми обрали третій підхід, який називається Code-First. Із назви зрозуміло, що спершу сутності та їх взаємозв’язки прописуються за допомогою коду, а потім EntityFramework на основі цього коду створює базу даних.

Для того, щоб база даних була створена, потрібно визначити похідний клас від DbContext, який представляє контекст бази даних і містить DbSet для кожного типу у моделі. Сутності представлені звичайними класами ООР, де поля сутностей – це властивості, або поля, класів. Зв’язки між таблицями також створюються за допомогою коду.

Існує три способи встановлення зв’язків.

Перший спосіб – це *code convention*. Все, що вимагається від розробника в такому випадку – давати такі назви полям, щоб EntityFramework міг «зрозуміти», який потрібно створити зв’язок, а також запам’ятати декілька правил створення сутностей. Наприклад, поле з назвою Id буде автоматично розпізнано фреймворком, як первинний ключ; а якщо у класі Challenge створити поле UnitId, то створиться зв’язок між таблицями Challenges та Units із вторинним ключем UnitId. Проте, цей спосіб підходить не для всіх випадків і не покриває усі потреби створення бази даних.

Другий спосіб – це *анотація даних*. Тоді, коли code convention недостатньо, можна використовувати атрибути, які записуються у квадратних дужках над полем або класом, для якого вони призначені. Анотації даних можуть бути розділені на дві категорії: атрибути, пов’язані із схемою бази даних, та атрибути, які відповідають за валідацію полів.

Те саме можна зробити, використовуючи *Fluent API configuration*. При конфігуруванні зв'язків з використанням Fluent API ви починаєте з екземпляра EntityTypeConfiguration та використовуєте методи HasRequired, HasOptional або HasMany, щоб вказати тип зв'язку для даної сутності. Для методів HasRequired та HasOptional необхідно передати лямбда-вираз, який представляє властивість навігації. Для методу HasMany потрібно передати лямбда-вираз, який представляє властивість навігації колекції. Зворотню навігацію можна налаштувати за допомогою методів WithRequired, WithOptional та WithMany. Ці методи мають перевантаження, які можуть бути використані для визначення кардинальності з односпрямованою навігацією. Після цього ви можете налаштувати властивості зовнішнього ключа за допомогою методу HasForeignKey. Цей метод приймає лямбда-вираз, який представляє властивість, що буде використовуватися як зовнішній ключ.

Різниця між Fluent API та data annotation полягає у синтаксисі та місці застосування. Якщо анотації прописуються безпосередньо у місці створення поля чи класу, то конфігурації Fluent API можуть знаходитися в окремих методах, файлах, або навіть проектах.

Окрім того, важливо знати, що хоч в теорії і можна поєднати обидва способи конфігурації, проте краще цього уникати. Це одночасно і полегшить читабельність та зрозумілість коду і допоможе не допустити помилок, якщо до якогось поля буде застосовано одразу два підходи. Тому у проєкті було вирішено використовувати лише Fluent API configuration.

Всі сутності можна побачити в проєкті Core у папці Entities(Рис. 3.4).

Додаток А

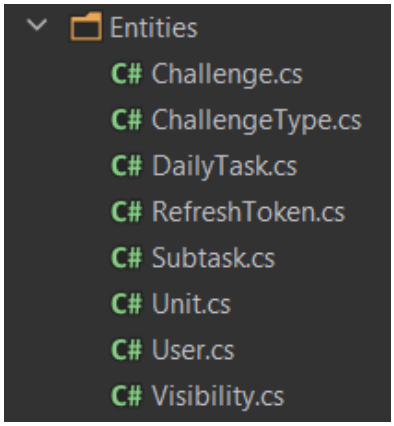

Рис. 3.4. Вміст папки Entities

Після створення сутностей, конфігурацій та dbContext було встановлено налаштування підключення до бази даних у файлі appsettings.json.

Для того, щоб база даних врешті створилася, потрібно було створити міграцію та виконати команду для оновлення бази даних, яка також її створює, якщо такої ще не існує. Міграції Code First - це рекомендований спосіб розвитку схеми бази даних додатку при використанні підходу Code First. Цей спосіб надає набір інструментів, які дозволяють:

1. Створити початкову базу даних, що відповідає EF-моделі.
2. Створювати міграції для відстеження змін, які вносяться до EF-моделі.
3. Підтримувати базу даних у актуальному стані з урахуванням цих змін.

Оскільки я працювала у Rider, який не є продуктом Microsoft, то не могла використовувати NuGet Package Manager Console для створення міграцій та оновлення БД, бо вона там відсутня. Спершу, я використовувала плагін Entity Framework Core UI, який надає інтерфейс для роботи з базою даних та міграціями. Проте, він виявився не найкращим рішенням для роботи із декількома проєктами, коли сутності знаходяться в одному, dbContext в іншому, а третій проєкт є точкою входу в програму. Тому найкращим способом для мене стало використання звичайного терміналу, який вбудований у Rider. За допомогою команди `dotnet ef` можна отримати доступ до взаємодії із базою даних, а проблема кількох проектів була вирішена за допомогою прапорців `-p` та `-s`, які дозволяють вказати відповідно проєкт з контекстом бази даних та проєкт запуску. Таким чином, щоб застосувати міграції до БД у проєкті потрібно запустити в терміналі таку команду:

```
dotnet ef database update -p Infrastructure -s API
```

У процесі розробки модель бази даних неодноразово змінювалась, редагувалась та доповнювалась. І тепер схема БД має такий вигляд (Рис. 3.5):

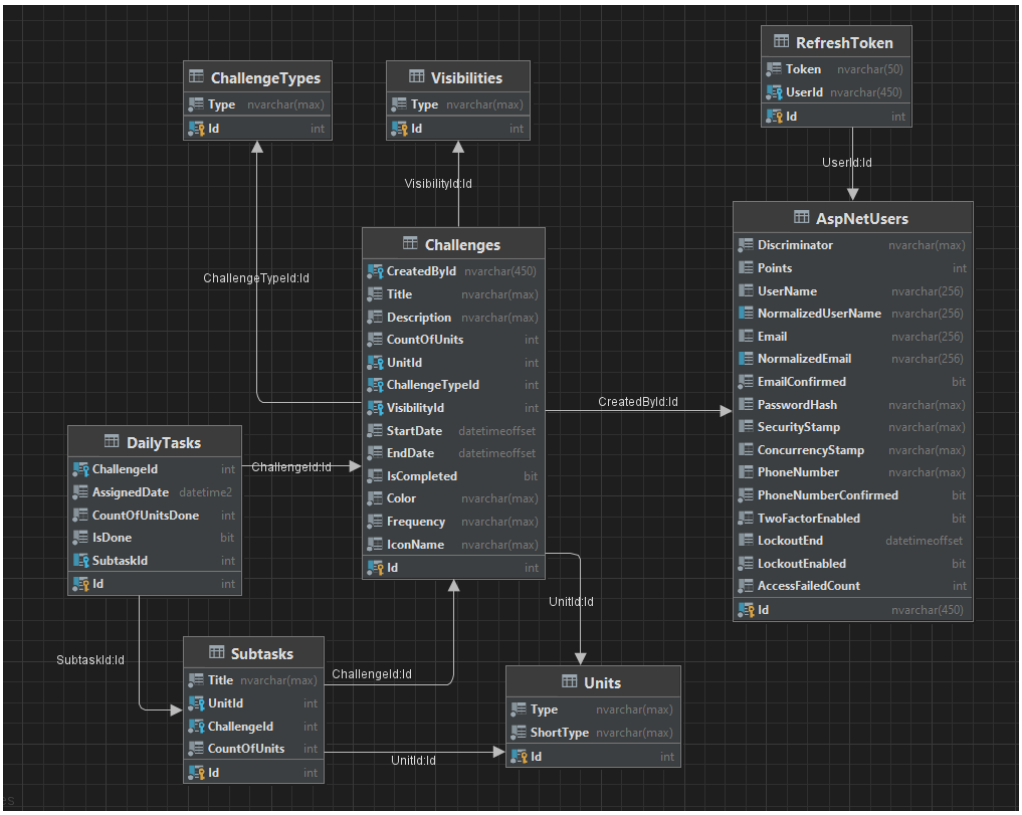

Рис. 3.5. Схема бази даних проєкту Habit-Rabbit

Для того, щоб спростити процес розробки та тестування, таблиці заповнюються початковими даними у класі SeedData. А така сутність як Units заповнюється лише в такий спосіб, оскільки програма не передбачає створення нових одиниць вимірювання користувачами. Тут також використовується бібліотека для генерації фейкових даних - Bogus. Саме за допомогою неї створено декілька тестових користувачів.

2.2. Архітектура рішення

2.2.1. Чиста архітектура

Чиста архітектура - це програмна архітектура, яка забезпечує контроль над усім кодом програми. Основна мета чистої архітектури - створення логіки, яка залишиться стабільною та не потребуватиме змін у майбутньому. Цей код повинен бути написаний без будь-яких прямих залежностей. Іншими словами, якщо ви захочете змінити фреймворк розробки або користувацький інтерфейс (UI) системи, ядро системи не повинно змінюватися. Це також означає, що всі зовнішні залежності повністю замінні.

Чиста архітектура включає в себе чотири рівні: доменний, прикладний, інфраструктурний та рівень інтерфейсу або представлення (Рис. 4.1). Ядро системи знаходиться в доменному та прикладному рівнях, які є центром проектування. Це означає, що ядро не залежить від доступу до даних та інфраструктури. Щоб досягти цієї мети, ми використовуємо інтерфейси та абстракції в ядрі системи, а реалізація знаходиться за межами ядра.

В Clean Architecture всі залежності в програмі є внутрішніми і ядро системи не має жодних залежностей від інших рівнів. Це означає, що у майбутньому, якщо нам знадобиться змінити UI або фреймворк, ми зможемо зробити це легко, оскільки всі інші залежності системи не залежать від ядра.

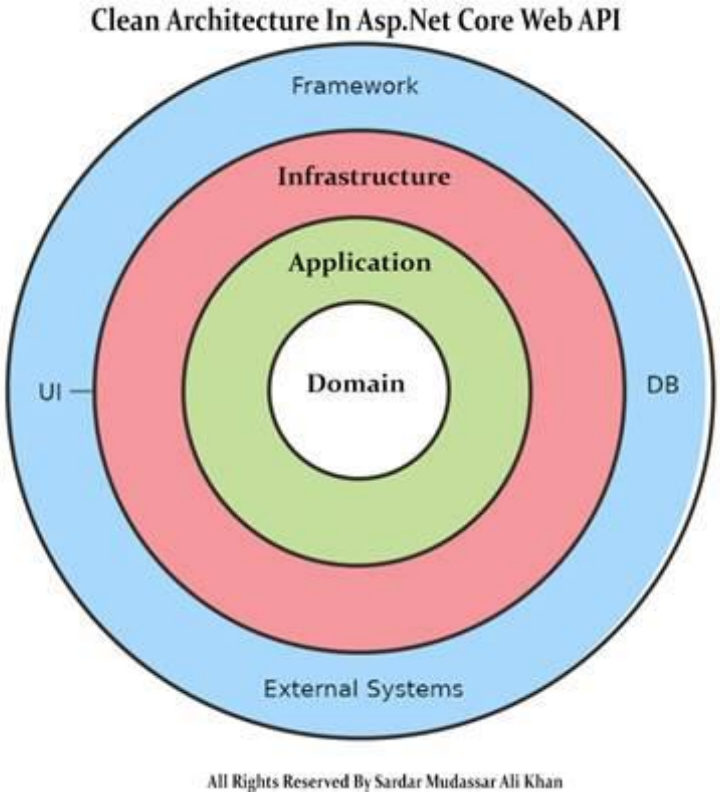

Рис. 4.1. Схема організації рівнів у Clean Architecture

Доменний рівень

Доменний рівень в чистій архітектурі містить логіку підприємства, наприклад, сутності та їх специфікації. Цей рівень лежить в центрі архітектури, де ми маємо сутності додатку, які є класами моделі додатку або класами моделі бази даних, використовуючи підхід "code first" в розробці додатків з використанням ядра Asp.net, ці сутності використовуються для створення таблиць в базі даних.

Прикладний рівень

Рівень додатку містить бізнес-логіку. Вся бізнес-логіка буде написана в цьому шарі. Саме в цьому шарі зберігаються інтерфейси сервісів, окремо від їх реалізації, для вільного зв'язку та розділення проблем.

Інфраструктурний рівень

У шарі інфраструктури ми маємо об'єкти моделі, ми будемо підтримувати всі міграції бази даних і контекстні об'єкти бази даних у цьому шарі. У цьому шарі ми маємо репозиторії всіх об'єктів моделі предметної області.

Рівень представлення

У випадку з API рівень представлення представляє нам дані об'єктів з бази даних за допомогою HTTP-запиту у вигляді JSON-об'єктів. Але у випадку інтерфейсних додатків ми представляємо дані за допомогою інтерфейсу користувача, використовуючи API. [9]

2.2.2. Переваги використання чистої архітектури

1. Покращена підтримуваність

Однією з головних переваг використання CleanArchitecture є покращення підтримуваності. Завдяки розділенню проблем різних компонентів і застосуванню правил залежностей, стає набагато легше розуміти і модифікувати код. Залежність від абстракцій дозволяє гнучко проектувати бізнес-логіку без необхідності знати деталі реалізації.

2. Модульність та розподіл обов'язків

CleanArchitecture допомагає створити чітке розділення проблем в коді. Кожен рівень має певне призначення і відокремлений від інших, що полегшує розуміння та модифікацію окремих компонентів без впливу на решту системи. Така модульність також полегшує повторне використання компонентів в інших проектах.

3. Можливість тестування

CleanArchitecture також полегшує тестування та налагодження коду. Оскільки внутрішнє коло не залежить від зовнішніх шарів, легше писати модульні тести, які фокусуються саме на бізнес-правилах. Це може допомогти виявити помилки на ранніх стадіях процесу розробки та зменшити загальний обсяг тестування.

4. Вільне з'єднання компонентів

CleanArchitecture також сприяє слабкому зв'язку між різними компонентами системи. Це означає, що легше замінити зовнішні залежності або внести інші зміни, не впливаючи на основну бізнес-логіку. Це може бути особливо корисно, коли мова йде про модернізацію або заміну технології.

5. Підвищена гнучкість

Ще однією ключовою перевагою CleanArchitecture є підвищена гнучкість. Розділяючи проблеми різних компонентів, легше модифікувати та адаптувати код до

мінливих вимог. Це може бути особливо корисно у швидкоплинних середовищах, де вимоги постійно змінюються.

6. Підвищення продуктивності команди

CleanArchitecture може допомогти підвищити продуктивність команди. Завдяки встановленню чіткого розподілу обов'язків та чітко визначених меж, членам команди легше зрозуміти свої ролі та обов'язки. Це може покращити комунікацію та співпрацю, що призведе до більш ефективної та результативної роботи.

2.2.3. Реалізація чистої архітектури у проєкті

Для реалізації чистої архітектури було вирішено розділити проєкт на рівні: Core, Infrastructure та API. Таким чином, Core – це ядро системи, а також рівень бізнес-логіки. Було вирішено об’єднати їх в один проєкт. Infrastructure – це інфраструктурний рівень, який відповідає за контекст бази даних та міграції. Та API – рівень представлення. Це проєкт ASP.NET Core WebAPI, який дозволяє «спілкуватися» з клієнтом за допомогою HTTP-запитів.

Рівень Core

На цьому рівні розміщено сутності та бізнес-логіку проекту (Рис. 4.2). Цей шар “не знає” про існування жодного з інших рівнів, отже від них не залежить.

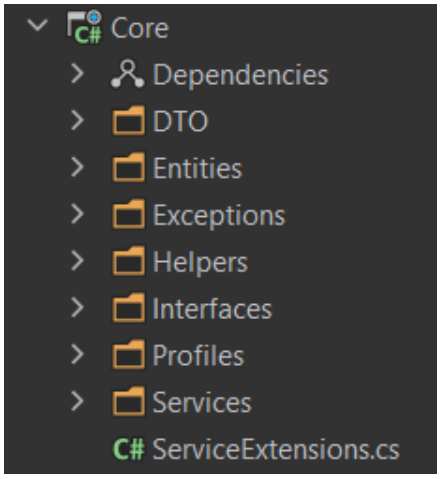

Рис. 4.2. Структура рівня Core

Окрім сутностей(Entities) тут також є DTO(Data Transfer Object) - моделі даних, які необхідні для надсилання та отримання з API. Для того, щоб перетворювати DTO в сутності та навпаки, у папці Profiles знаходяться профілі, які конфігурують ці перетворення за допомогою бібліотеки Automapper.

У директорії Services знаходяться такі класи: AuthenticationService, ChallengeService, DailyTaskService, JwtService, UnitService, UserService. Кожен із сервісів відображає бізнес-логіку роботи із певною сутністю. Задля дотримання принципів SOLID робота із сервісами ведеться через інтерфейси, які розміщені у папці Interfaces. SOLID — це абревіатура, складена з перших літер п'яти базових принципів об'єктно-орієнтованого програмування та дизайну і запропонована Робертом Мартіном. Останній з цих принципів – це принцип інверсії залежностей (Dependency Inversion Principle). Він говорить про те, що високорівневі модулі не повинні залежати від низькорівневих. І ті, і ті мають залежати від абстракцій. Такими абстракціями у нас і є інтерфейси, які потім будуть використовуватися на рівні API замість самих сервісів.

Саме сервіси і є основою додатку, бо тут зосереджена уся бізнес-логіка та функціонал. Рівень презентації використовує ці сервіси в HTTP-контроллерах, а самі сервіси обробляють запити, містять логіку роботи додатку та звертаються до бази даних, щоб читати, записувати, редагувати або видаляти дані.

Рівень Infrastructure

Infrastructure або рівень доступу до даних (Рис. 4.3) “знає” про рівень Core. Але, водночас, він нічого не знає про рівень API, а також про те, яка база даних буде використовуватись.

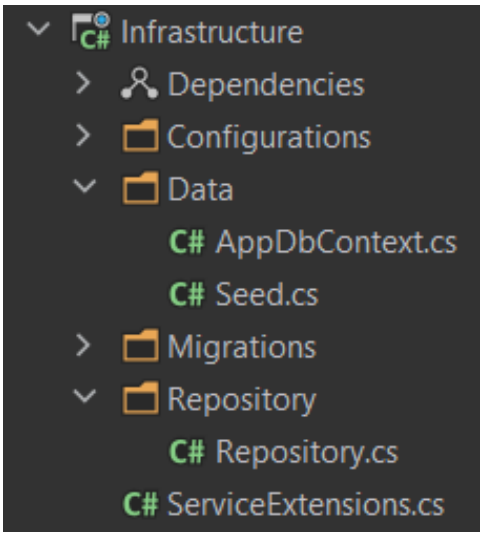

Рис. 4.3. Структура рівня Infrastructure

На цьому рівні знаходиться DBContext, клас SeedData, який заповнює базу початковими даними, міграції бази даних, конфігурації для того, щоб між сутностями з рівня Core були створені правильні зв’язки при ініціалізації бази даних та створенні таблиць сутностей. А також тут знаходиться базовий клас Repository, який реалізує патерн Репозиторій у проєкті.

Рівень API

API - рівень, завдяки якому клієнт може отримувати дані, а також надсилати їх на сервер (Рис. 4.4). По ієрархії цей рівень знаходиться “вище” попередніх, а тому має посилання і на рівень Core, і на Infrastructure.

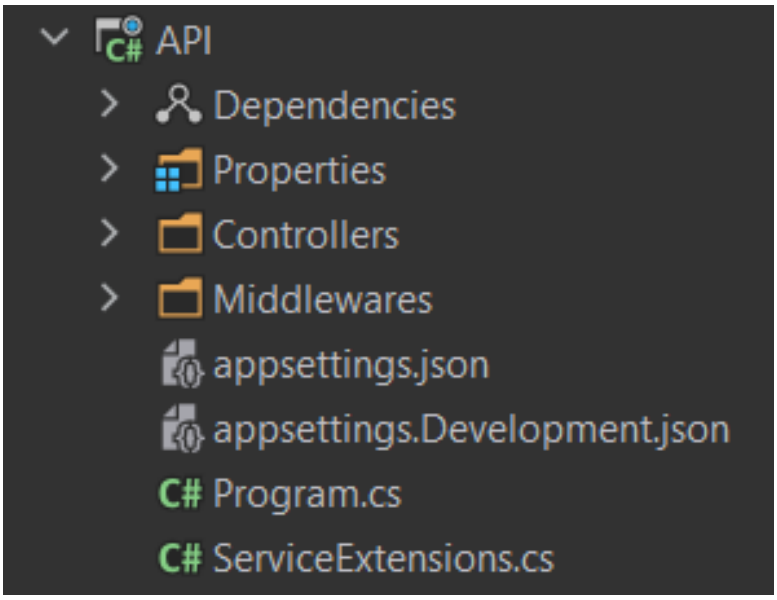

Рис. 4.4. Структура рівня API

На відміну від попередніх рівнів, які були реалізовані як проєкти ClassLibrary, цей рівень представлений ASP.Net Core WebAPI і є проєктом запуску, за що відповідає клас Program. Тут лежать *api*-контролери у папці Controllers, а також у файлі *appsettings.json* знаходиться підключення до бази даних і налаштування *JWT*.

У таблиці 4.1 наведено перелік створених контролерів та їх призначення.

Таблиця 4.1

Перелік контролерів

Назва контролера	***Методи***
AuthenticationController	Містить методи для входу та реєстрації користувача в системі, оновлення access-токену за допомогою refresh-токену та для виходу із системи.
ChallengeController	Містить методи для створення, редагування та видалення челенджу, а також для перегляду всіх челенджів користувача або одного за Id челенджу.
DailyTaskController	Містить методи для перегляду всіх завдань на сьогодні, або за конкретною датою. Також тут є методи для додавання та видалення прогресу
UnitController	Призначений для перегляду всіх доступних одиниць вимірювання прогресу.
UserController	Містить методи для отримання інформації про поточного користувача за допомогою розшифрування токену, який передається у хедері запиту, а також для оновлення та видалення профілю.

На рівні представлення також налаштовується Swagger. В результаті ми отримуємо інтерфейс, який дозволяє зручно виконувати запити та тестувати API.

2.2.4. Патерн Репозиторій

Як уже було згадано вище, у проекті використовується архітектурний патерн Репозиторій. Шаблон Репозиторій є одним з більш популярних шаблонів на даний момент. Він дотримується принципів SOLID і, якщо використовується правильно, є простим та зрозумілим.У цього патерну є дві основні мети: перша - це абстракція рівня даних, а друга - це спосіб централізованої обробки об'єктів домену.

Патерн Репозиторій здобув досить велику популярність з моменту його першого запровадження як частини Domain-Driven Design у 2004 році. За своєю суттю, він

надає абстракцію даних, щоб програма могла працювати з простою абстракцією, яка має інтерфейс, що наближається до колекції. Додавання, видалення, оновлення та вибір елементів з цієї колекції виконується за допомогою послідовності простих методів, без потреби мати справу з питаннями баз даних, такими як підключення, команди, курсори або читачі. Використання цього шаблону може допомогти досягти слабкої зв'язності і зберігати об'єкти домену незалежними від збереження даних. [10]

Для застосування цього патерну в проєкті було використано бібліотеку Ardalis.Specification, яка доступна у NuGet Package менеджері. Сама реалізація полягала у створенні класу Repository на рівні Infrastructure (Додаток Б). Цей клас наслідується від базового класу Репозиторію, який є Generic-типом. Це означає, що при створенні об’єкта класу, туди можна передати будь-яку сутність, з якою необхідно працювати і цей об’єкт успішно забезпечить виконання методів, пов’язаних із доступом до бази даних.

Зокрема, у класі Repository реалізовано такі методи:

`GetAllAsync()` – повертає всі записи вказаної таблиці, наявні у базі даних.

`GetByIdAsync<TKey>(TKey id)` – виконує пошук за Id та повертає один елемент.

`AddAsync(TEntity entity)` – додає один запис до таблиці.

`AddRangeAsync(IList<TEntity> entities)` - додає декілька записів до таблиці у базі даних.

`UpdateAsync(TEntity entityToUpdate)` - оновлює вказаний елемент.

`DeleteAsync(TEntity entityToDelete)` –видаляє переданий запис із бази даних.

`DeleteRange(IEnumerable<TEntity> entitiesToDelete)` –видаляє перелік елементів.

`SaveChangesAsync()` –зберігає зміни в базі даних; без цього методу зміни не будуть застосовані.

`Query(params Expression<Func<TEntity, object>>[] includes)` – метод для написання власних запитів, наприклад, якщо потрібна фільтрація чи сортування даних.

Висновки до розділу 2

У цьому розділі було розглянуто створення use-case діаграми проєкту, проектування бази даних та реалізацію чистої архітектури у побудові програми.

Діаграма прецедентів проєкту Habit-Rabbit надає загальне уявлення про функціонал системи та взаємодію з різними типами користувачів. Згідно з діаграмою, гостям системи доступні лише обмежені можливості, такі як реєстрація і вхід в систему. Натомість, авторизовані користувачі мають повний доступ до функцій системи, таких як перегляд і редагування персональної інформації, створення, редагування та видалення челенджів, перегляд та відмітка прогресу завдань, перегляд статистики та отримання винагороди за досягнення. Використання сервісу drawio.com дозволило команді ефективно побудувати діаграму прецедентів та забезпечити однакове розуміння задачі серед усіх членів команди. Ця діаграма є важливим інструментом для опису функціональності системи та забезпечення правильної співпраці між командою розробників.

База даних є фундаментом сучасних інформаційних систем і є організованим набором логічно пов'язаних даних. Вона забезпечує збереження даних і зв'язків між ними, а також структурує та підтримує дані відповідно до потреб користувачів.

У нашому проєкті було обрано реляційний тип бази даних, який зберігає дані у вигляді табличних структур. Реляційні бази даних фокусуються на зв'язках між даними і є найпоширенішим типом баз даних.

Створення реляційної бази даних включає кілька кроків, таких як визначення мети бази даних, визначення сутностей, атрибутів та зв'язків між ними, а також нормалізацію структури. В проєкті було визначено перелік сутностей та їх взаємозв'язків, які були реалізовані за допомогою коду.

Для створення бази даних використовувався підхід Code-First, де спочатку визначалися сутності та їх зв'язки у коді, а потім на основі цього коду база даних створювалася за допомогою EntityFramework. Було використано Fluent API configuration для встановлення зв'язків між сутностями.

Для забезпечення взаємодії з базою даних було налаштовано підключення у файлі appsettings.json. Міграції Code First використовувалися для відстеження та

застосування змін до бази даних. Застосування міграцій було здійснено через термінал з використанням команди `dotnet ef`.

Процес розробки та тестування спрощувався завдяки заповненню бази даних тестовими даними, що дозволило перевірити функціональність та відповідність бази даних поставленим вимогам.

Чиста архітектура є потужним інструментом для створення стабільних, легко змінюваних та підтримуваних програмних рішень. Її реалізація передбачає розділення коду на чотири рівні: доменний, прикладний, інфраструктурний та рівень інтерфейсу. Ядро системи знаходиться в доменному та прикладному рівнях і не залежить від зовнішніх компонентів. Чиста архітектура має кілька переваг, таких як покращена підтримуваність, модульність, можливість тестування, вільне з'єднання компонентів, підвищена гнучкість та підвищення продуктивності команди.

Для реалізації чистої архітектури у проєкті було використано розподіл на три рівні: Core, Infrastructure та API. Рівень Core містить сутності та бізнес-логіку проекту. Рівень Infrastructure відповідає за доступ до даних та міграції. Рівень API представлений як ASP.NET Core WebAPI і забезпечує спілкування з клієнтом за допомогою HTTP-запитів.

Застосування архітектурного патерну Репозиторій в проекті дозволило досягти слабкої зв'язності і централізованої обробки об'єктів домену. Цей шаблон надає абстракцію рівня даних, що дозволяє програмі працювати з простою абстракцією, що наближається до колекції.

Використання бібліотеки Ardalis.Specification у проекті спростило реалізацію патерну Репозиторій, надаючи готові інструменти для створення репозиторіїв. Клас Repository на рівні Infrastructure успішно використовується для роботи з будь-якою сутністю, з якою необхідно працювати.

У класі Repository були реалізовані методи, які забезпечують основні операції з базою даних, такі як отримання всіх записів, пошук за ідентифікатором, додавання, оновлення та видалення записів. Також були додані додаткові методи для виконання власних запитів, фільтрації чи сортування даних.

РОЗДІЛ 3

ПРОГРАМНЕ ТА ТЕХНІЧНЕ ЗАБЕЗПЕЧЕННЯ

3.1. Засоби розробки

3.1.1. Опис програмного забезпечення

З метою ефективної розробки програмного забезпечення, наша команда розділила проект на кілька етапів, включаючи дизайн, клієнтську та серверну частини. Кожен з цих етапів вимагав використання певного, специфічного для конкретних задач програмного середовища.

Для розробки серверної частини ми використовували JetBrains Rider, який дозволяє ефективно писати код та взаємодіяти з базами даних. Для розробки інтерфейсу було використано WebStorm, який забезпечує багато функцій для розробки фронтенду, включаючи підтримку мов програмування та автоматичне завантаження залежностей. Для проектування та малювання дизайну інтерфейсу ми використовували Figma, що дозволило нам ефективно створювати та спільно працювати над дизайном.

Використання цих програмних засобів дозволило нам зосередитися на кожному етапі розробки та максимально оптимізувати процес.

JetBrains Rider

JetBrains Rider - це інтегроване середовище розробки на основі платформи IntelliJ та ReSharper, спрямоване на .NET-розробку.

Широкі можливості для проектів .NET

Rider підтримує багато типів проєктів .NET, включаючи .NET Framework, .NET Core та проекти на основі Mono. Це дозволяє розробляти десктопні програми, сервіси та бібліотеки .NET, ігри Unity, додатки Xamarin, веб-додатки ASP.NET та ASP.NET Core.

Багатофункціональний та продуктивний

Rider має вбудовані можливості перевірки коду в реальному часі, сотні контекстних дій та рефакторингів, що забезпечує ReSharper, а також можливості IDE платформи IntelliJ. Незважаючи на широкий набір функцій, Rider працює швидко та ефективно.

Крос-платформеність

Rider можна використовувати на різних платформах: Windows, macOS та Linux. Він може запускати та налагоджувати декілька середовищ виконання .NET, забезпечуючи крос-платформеність вашого проекту.

Окрім наведених переваг, Rider надає зручний інтерфейс для взаємодії з базами даних безпосередньо із IDE. [2]

WebStorm

WebStorm - це інтегроване середовище розробки для JavaScript та суміжних технологій. Як і інші IDE JetBrains, воно робить процес розробки приємнішим, автоматизуючи рутинну роботу та допомагаючи легко справлятися зі складними завданнями.

Найрозумніший редактор

WebStorm глибоко розуміє структуру вашого проекту і може допомогти вам з кожним аспектом написання коду. Він автоматично доповнить ваш код, виявить і запропонує виправлення помилок і надмірностей, а також допоможе вам безпечно рефакторити код.

Вбудовані інструменти для розробників

Однією з найкращих переваг роботи в IDE є те, що всі необхідні інструменти доступні в одному місці. Використовуйте WebStorm для налагодження та тестування клієнтських і Node.js додатків, а також для роботи з керуванням версіями. Скористайтеся перевагами лінтерів, інструментів збірки, терміналу та HTTP-клієнта, які інтегровані в IDE.

Ефективна командна робота

Швидше залучайте нових членів команди з усіма ключовими функціями. Кодуйте разом у реальному часі та спілкуйтеся з іншими прямо з IDE. Діліться конфігурацією проекту, включаючи налаштування стилів коду, та ефективно працюйте з Git та GitHub. [3]

Figma

Figma є інструментом розробки інтерфейсів та прототипування, який доступний як онлайн-сервіс. Компанія Figma розробляє цей векторний сервіс для спільної роботи

над дизайном інтерфейсів. Figma пропонує дві форми використання: через веб-браузер або як клієнтський додаток для роботи на десктопі користувача.

Швидко та безпечно в хмарі

Figma пропонує безпечне зберігання та роботу з вашими проектами у хмарі. Відтепер ви можете забути про проблеми з версіями та хмарну синхронізацію. Ваші файли зберігаються в Інтернеті та завжди доступні вам з будь-якого пристрою.

Ефективна співпраця

Figma була створена спеціально для ефективної співпраці. Обмін файлами є легким та простим, а співробітники можуть працювати в одному проекті одночасно, додавати коментарі та вносити правки. Функції спільної роботи відображають правильний контекст та дозволяють отримувати швидкий та якісний зворотний зв'язок.

Деталі не залишаються непоміченими

Завдяки широкому спектру інструментів та можливостям програми, ви можете зосередитися на найдрібніших деталях свого проекту та зробити його досконалим.

GitHub

GitHub - це веб-сервіс для контролю версій, який базується на Git. Git - це розподілена система керування версіями, що дозволяє зберігати, відслідковувати та керувати версіями файлів проекту.

GitHub забезпечує зручний спосіб зберігання та спільного використання коду та інших файлів проекту. Розробники можуть створювати репозиторії, додавати та зберігати файли, використовувати розгалуження (branches) для розробки функціоналу в окремих гілках, тестувати та зливати зміни в основну гілку проекту, а також відстежувати проблеми та пропонувати внесення змін через pull requests.

GitHub також надає багато інших корисних функцій, таких як інструменти для автоматичної збірки та тестування проекту, інтеграцію з іншими сервісами, статистику використання та багато іншого.

Для використання GitHub розробникам необхідно створити обліковий запис та встановити Git на свій комп'ютер. GitHub забезпечує можливість зберігати публічні репозиторії безкоштовно, але за користування приватними репозиторіями стягує плату в залежності від кількості користувачів та розміру зберігання.

ClickUp

ClickUp - це хмарний інструмент, який надає можливості для спільної роботи та управління проектами. Його функціонал включає інструменти для комунікації та спільної роботи, призначення та статуси завдань, сповіщення та панель інструментів для завдань. Користувачі можуть призначати коментарі та завдання для конкретних членів команди або груп членів команди, а також можуть позначати їх як вирішені або в процесі виконання.

ClickUp також має Agile-інформаційну, що дозволяє користувачам керувати проектами ефективніше. Потік активності відображає завдання в режимі реального часу, що допомагає користувачам бути в курсі всього, що відбувається з їхнім проектом.

Крім того, ClickUp дозволяє налаштувати сповіщення, щоб користувачі отримували їх тільки для певних елементів, що полегшує спільну роботу в команді. Функція згадок сповіщає користувачів, коли інший член команди згадує їх в обговоренні. Коментарі можна редагувати після публікації, що дозволяє користувачам виправляти помилки та додавати додаткову інформацію. Крім того, ClickUp інтегрується з Slack та GitHub, що дозволяє користувачам працювати з цими платформами зручно та ефективно.

Swagger

Для проведення тестування API було вирішено використовувати Swagger. Swagger надає інструменти для розробки, документування та тестування REST API, що дозволяє забезпечити якість функціонування серверної частини веб-додатка. Будучи заснованим на специфікації OpenAPI, Swagger забезпечує стандартизацію процесу написання API, що спрощує розробку та тестування серверної частини. Використання Swagger дозволяє нам зосередитися на важливих аспектах, таких як безпека, продуктивність та масштабованість, тим самим забезпечуючи нашій команді швидку і якісну розробку веб-додатка.

Swagger допомагає забезпечити стандартизацію та спрощення використання API, а також має багато додаткових переваг:

- дружній користувацький інтерфейс, який відображає схему роботи з API;

- зрозумілу документацію для розробників, менеджерів проектів та клієнтів;
- можливість читати специфікації як людиною, так і машиною;
- інтерактивну документацію, що легко тестується;
- підтримку створення бібліотек API більш ніж 40 мовами;
- підтримку форматів JSON та YAML для полегшення редагування;
- допомогу в автоматизації процесів, пов'язаних з API.

3.1.2. Аналіз стеку технологій

ASP.NET Core

ASP.NET - це веб-фреймворк з відкритим вихідним кодом, створений корпорацією Майкрософт для створення сучасних веб-додатків і сервісів за допомогою .NET.

ASP.NET є крос-платформеним і працює на Windows, Linux, macOS та Docker.

Платформа .NET

NET - це платформа для розробників, що складається з інструментів, мов програмування та бібліотек для створення різних типів додатків.

Базова платформа надає компоненти, які підходять для всіх типів додатків. Додаткові фреймворки, такі як ASP.NET, розширюють .NET компонентами для створення конкретних типів додатків.

Ось деякі речі, що входять до складу платформи .NET:

- Мови програмування C#, F# та Visual Basic
- Базові бібліотеки для роботи з рядками, датами, файлами/інтерфейсами вводу-виводу тощо
- Редактори та інструменти для Windows, Linux, macOS та Docker

.NET надає стандартний набір бібліотек базових класів та API, які є спільними для всіх .NET-додатків.

Кожна модель програми може також використовувати додаткові API, які є специфічними для операційних систем, на яких вона працює, або можливостей, які вона надає. Наприклад, ASP.NET - це кросплатформенний веб-фреймворк, який надає додаткові API для створення веб-додатків, що працюють під Linux або Windows.

ASP.NET

ASP.NET розширює платформу .NET інструментами та бібліотеками, спеціально призначеними для створення веб-додатків.

Ось деякі речі, які ASP.NET додає до платформи .NET:

- Базовий фреймворк для обробки веб-запитів на C# або F#
- Синтаксис шаблонів веб-сторінок, відомий як Razor, для створення динамічних веб-сторінок за допомогою C#
- Бібліотеки для поширених веб-паттернів, таких як Model View Controller (MVC)
- Система автентифікації, що включає бібліотеки, базу даних та шаблони сторінок для обробки входів, включаючи багатофакторну автентифікацію та зовнішню автентифікацію за допомогою Google, Twitter тощо.
- Розширення редактора для підсвічування синтаксису, завершення коду та інших функцій, спеціально призначених для розробки веб-сторінок

Оскільки ASP.NET розширює .NET, ви можете використовувати велику екосистему пакетів і бібліотек, доступних для всіх розробників .NET. Ви також можете створювати власні бібліотеки, які можна використовувати в будь-яких додатках, написаних на платформі .NET.

ASP.NET дозволяє створювати багато типів веб-додатків, включаючи веб-сторінки, REST API, мікросервіси та хаби, які передають контент в режимі реального часу підключеним клієнтам.

ASP.NET Core

ASP.NET Core - це версія ASP.NET з відкритим вихідним кодом, яка працює на macOS, Linux та Windows. Вперше ASP.NET Core був випущений у 2016 році і є переробленою версією попередньої версії ASP.NETб яка працювала виключно для Windows. [4]

REST API з використанням .NET та C#

Створення сервісів, які можуть працювати з браузерами та мобільними пристроями, є легкою задачею завдяки ASP.NET. Цей фреймворк дозволяє використовувати одні й

ті ж шаблони та інструменти для розробки веб-сторінок та сервісів в одному проекті. Це дозволяє зменшити час розробки та забезпечити більш ефективну співпрацю між розробниками. Крім того, використання ASP.NET забезпечує високу масштабованість проекту та його стабільну роботу під великим навантаженням.

Проста серіалізація

ASP.NET розроблено з метою надання підтримки для сучасних веб-додатків. Одним з його переваг є можливість автоматичної серіалізації класів в правильно відформатований JSON. Це означає, що немає необхідності в додатковій конфігурації, оскільки це відбувається автоматично в кінцевих точках. Звісно, ви можете налаштувати серіалізацію для кінцевих точок з унікальними вимогами. Таким чином, ASP.NET надає просту та зручну можливість серіалізації, яка дозволяє легко обмінюватись даними між клієнтами та сервером.

Аутентифікація та авторизація

ASP.NET надає потужні інструменти для забезпечення безпеки вашого додатка, зокрема аутентифікації та авторизації. Вбудована підтримка стандартних веб-токенів JSON (JWT) дозволяє створювати безпечні ендпоінти API, що забезпечують захист від несанкціонованого доступу. Застосування політик авторизації дозволяє гнучко визначати правила контролю доступу, забезпечуючи максимальний рівень безпеки вашого додатка. Усі ці інструменти доступні в коді, що дозволяє легко налаштовувати та змінювати захист вашого додатка.

Вбудована маршрутизація

ASP.NET дозволяє зручно та ефективно визначати маршрутизацію у вашому коді за допомогою атрибутів. Це дозволяє зосередитись на логіці додатку та роботі з даними, не витрачаючи час на ручну конфігурацію маршрутів.

Інформація з шляху запиту, рядка запиту та тіла запиту автоматично прив'язується до параметрів методу. Це забезпечує зручну та просту обробку даних в коді, що дозволяє швидко та ефективно відповідати на запити користувачів.

Завдяки цим можливостям, ви можете забезпечити високу продуктивність вашого додатку та швидко реагувати на зміни в запитах користувачів. [5]

Entity Framework

Entity Framework - це ORM-фреймворк з відкритим вихідним кодом для .NET-додатків, що підтримується Microsoft. Вона дозволяє розробникам працювати з даними, використовуючи об'єкти специфічних для домену класів, не зосереджуючись на базових таблицях і стовпцях бази даних, де ці дані зберігаються. Завдяки Entity Framework розробники можуть працювати з даними на вищому рівні абстракції, а також створювати і підтримувати додатки, орієнтовані на дані, з меншим обсягом коду порівняно з традиційними додатками.

Офіційне визначення: "Entity Framework - це об'єктно-реляційний маппер (ORM), який дозволяє .NET розробникам працювати з базою даних, використовуючи об'єкти .NET. Це усуває необхідність в більшості коду для доступу до даних, який зазвичай доводиться писати розробникам".

На малюнку нижче (Рис. 3.1) зображено, як Entity Framework може бути інтегрований у програму.

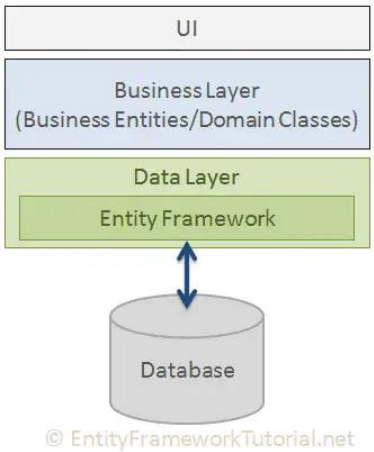

Рис. 3.1. Використання Entity Framework у проєкті

Він забезпечує зручний спосіб взаємодії з базою даних у вигляді об'єктів. На цьому малюнку показано, що Entity Framework може бути використаний для звернення до бази даних з допомогою об'єктів-моделей, які відображають таблиці в базі даних. Ці об'єкти-моделі можуть бути створені вручну або згенеровані автоматично з бази даних. Далі, програма може виконувати дії з базою даних, такі як додавання, видалення або зміна даних, звертаючись до цих об'єктів-моделей. Окремо зображено, що Entity Framework може використовувати мову SQL для взаємодії з базою даних,

або ж можна використовувати мову LINQ (Language-Integrated Query), яка дозволяє виконувати запити до бази даних в коді на мові програмування C# або VB.NET. [6]

Переваги Entity Framework

- Кросплатформеність: EF Core - це кросплатформенний фреймворк, який може працювати на Windows, Linux та Mac.
- Моделювання: EF (Entity Framework) створює EDM (Entity Data Model) на основі сутностей POCO (Plain Old CLR Object) з властивостями отримання/встановлення різних типів даних. Вона використовує цю модель при запиті або збереженні даних сутностей в основній базі даних.
- Запити: EF дозволяє використовувати запити LINQ (C#/VB.NET) для отримання даних з основної бази даних. Постачальник бази даних перекладе ці LINQ-запити на мову запитів для конкретної бази даних (наприклад, SQL для реляційної бази даних). EF також дозволяє виконувати необроблені SQL-запити безпосередньо до бази даних.
- Відстеження змін: EF відстежує зміни, що відбулися в екземплярах ваших сутностей (значеннях властивостей), які необхідно відправити в базу даних.
- Збереження: EF виконує команди INSERT, UPDATE і DELETE до бази даних на основі змін, що відбулися з вашими сутностями, коли ви викликаєте метод SaveChanges(). EF також надає асинхронний метод SaveChangesAsync().
- Паралелізм: За замовчуванням EF використовує оптимістичний паралелізм для захисту від перезапису змін, зроблених іншим користувачем з моменту отримання даних з бази даних.
- Транзакції: EF виконує автоматичне керування транзакціями під час запиту або збереження даних. Він також надає можливості для налаштування управління транзакціями.
- Кешування: EF включає перший рівень кешування з коробки. Таким чином, повторний запит поверне дані з кешу замість того, щоб звертатися до бази даних.

- Вбудовані конвенції: EF дотримується угод щодо шаблону конфігураційного програмування і включає набір правил за замовчуванням, які автоматично налаштовують модель EF.
- Конфігурації: EF дозволяє конфігурувати модель EF, використовуючи атрибути анотацій даних або API Fluent для заміни стандартних конвенцій.
- Міграції: EF надає набір команд міграції, які можна виконати на консолі NuGet Package Manager або в інтерфейсі командного рядка для створення або керування схемою баз даних.
- Підтримка зв’язків між таблицями: EF дозволяє визначати зв'язки між таблицями в базі даних, такі як один-до-одного, один-до-багатьох та багато-до-багатьох. Це дозволяє легко отримувати пов'язані дані з бази даних.
- Lazy loading: EF підтримує ліниве завантаження даних, що означає, що пов'язані дані будуть завантажуватись з бази даних тільки тоді, коли вони будуть дійсно запрошені в програмі.
- Eager loading: EF дозволяє використовувати зведені запити (Eager Loading), щоб завантажити пов'язані дані з бази даних за один запит, замість того, щоб завантажувати їх окремо при кожному запиті.
- Підтримка множинних постачальників баз даних: EF підтримує підключення до різних постачальників баз даних, таких як SQL Server, MySQL, PostgreSQL та інших.
- Підтримка Code First: EF дозволяє використовувати Code First підхід до розробки баз даних, що означає, що ви можете спочатку створити моделі даних у вашій програмі і потім автоматично створити базу даних з цих моделей.

Automapper

AutoMapper - це маппер об'єкт-об'єкт. Об'єкт-об'єктне відображення працює шляхом перетворення вхідного об'єкта одного типу у вихідний об'єкт іншого типу. AutoMapper цікавий тим, що він надає деякі цікаві конвенції, які полегшують брудну роботу, пов'язану з визначенням того, як відобразити тип A у тип B. Якщо тип B

відповідає встановленій конвенції AutoMapper, то для відображення двох типів потрібна майже нульова конфігурація.

Для чого використовувати Automapper?

AutoMapper забезпечує не лише просту конфігурацію типів і тестування відображень, але також допомагає зменшити кількість дублювання коду та збільшити читабельність. Крім того, об'єкт-об'єктне відображення дозволяє забезпечити більш гнучке управління даними та зниження залежностей між різними шарами архітектури програми. Зокрема, це дозволяє змінювати структуру бази даних без впливу на інші частини програми та забезпечує можливість збільшувати продуктивність за рахунок зменшення кількості запитів до бази даних. [7]

Microsoft SQL Server

Microsoft SQL Server - одна з основних систем управління реляційними базами даних на ринку, яка обслуговує широкий спектр програмних додатків для бізнес-аналітики та аналізу в корпоративному середовищі.

Заснована на мові Transact-SQL, вона включає в себе набір стандартних розширень для програмування, а її застосування доступне для використання як в локальних, так і в хмарних середовищах.

Microsoft SQL Server ідеально підходить для зберігання всієї необхідної інформації в реляційних базах даних, а також для управління цими даними без ускладнень, завдяки своєму візуальному інтерфейсу та опціям й інструментам, які він має. Це надзвичайно важливо, особливо для веб-сайтів, які мають можливість реєстрації користувачів для входу в систему.

Серед основних функцій, які вирізняють Microsoft SQL Server, є різноманітні інструменти для управління та аналізу даних, а також бізнес-аналітика, за допомогою якої можна отримати інформацію про ваш бізнес і клієнтів за допомогою машинного навчання.

Завдяки інтеграції з Azure AI Microsoft SQL Server дозволяє легко інтегрувати дані в додатки та використовувати широкий набір когнітивних сервісів для застосування

штучного інтелекту в будь-якому масштабі даних, як у локальному, так і в хмарному середовищі.

Оскільки він базується на відкритому вихідному коді, до нього дуже легко отримати доступ, і переважна більшість програмістів, що працюють у сфері веб-розробки, використовували Microsoft SQL Server у деяких своїх проектах, а також, оскільки він дуже поширений, то має велику спільноту, яка пропонує підтримку іншим користувачам.

3.2. Опис програмної реалізації

3.2.1. Автентифікація та авторизація

Хоча автентифікація та авторизація можуть здаватися схожими і їх часто вважають взаємозамінними поняттями, у світі керування ідентифікацією та доступом це різні процеси безпеки. Автентифікація перевіряє особу користувача або служби, а авторизація визначає їхні права доступу. Хоча ці два терміни звучать однаково, вони відіграють різні, але однаково важливі ролі у захисті програм і даних. Розуміння різниці має вирішальне значення. У сукупності вони визначають безпеку системи. Неможливо мати безпечне рішення, якщо не налаштувати як автентифікацію, так і авторизацію правильно.

Автентифікація

- Аутентифікація — це процес перевірки облікових даних, наданих користувачем, із тими, що зберігаються в системі, щоб підтвердити, що користувач є тим, за кого себе видає. Якщо облікові дані збігаються, ви надаєте доступ. Якщо ні, ви відхиляєте запит. Аутентифікацію ще називають AuthN.
- Вона використовується як сервером, так і клієнтом. Сервер використовує аутентифікацію, коли хтось хоче отримати доступ до інформації, і сервер повинен знати, хто намагається отримати цей доступ. Клієнт використовує її, коли хоче знати, що це той самий сервер, за який він себе видає.
- Аутентифікація на сервері здійснюється здебільшого за допомогою імені користувача та пароля. Інші способи аутентифікації на сервері також можуть

здійснюватися за допомогою карток, сканування сітківки ока, розпізнавання голосу та відбитків пальців.

- Аутентифікація не визначає, які завдання в рамках процесу може виконувати одна особа, які файли вона може переглядати, читати чи оновлювати. Здебільшого вона визначає, ким насправді є особа чи система.

Відомі методи аутентифікації:

1. Аутентифікація на основі пароля

Це найпростіший спосіб аутентифікації. Для цього потрібен пароль для конкретного імені користувача. Якщо пароль збігається з іменем користувача та обидві дані збігаються з базою даних системи, користувача буде успішно автентифіковано.

2. Аутентифікація без пароля

У цій техніці користувачеві не потрібен пароль; замість цього він отримує OTP (one-time password) або посилання на свій зареєстрований номер мобільного телефону або номер телефону. Це також можна сказати, що автентифікація на основі OTP.

3. 2FA/MFA

2FA/MFA або 2-факторна автентифікація/багатофакторна автентифікація є вищим рівнем аутентифікації. Для аутентифікації користувача потрібен додатковий PIN-код або питання безпеки.

4. Єдиний вхід

Єдиний вхід або SSO — це спосіб увімкнути доступ до кількох програм за допомогою одного набору облікових даних. Це дозволяє користувачеві ввійти один раз, і він автоматично ввійде в усі інші веб-програми з того самого централізованого каталогу.

5. Соціальна автентифікація

Соціальна автентифікація не вимагає додаткового захисту; замість цього вона перевіряє користувача за наявними обліковими даними для доступної соціальної мережі.

Авторизація

- Авторизація — це процес надання комусь дозволу на виконання певних дій. Це означає, що це спосіб перевірити, чи має користувач дозвіл на використання ресурсу, чи ні.
- Вона визначає, до яких даних та інформації може отримати доступ один користувач. ЇЇ також називають AuthZ.
- Авторизація зазвичай працює з автентифікацією, щоб система могла знати, хто має доступ до інформації.
- Авторизація не завжди потрібна для доступу до інформації, доступної в Інтернеті. Деякі дані, доступні в Інтернеті, можуть бути доступні без будь-якого дозволу.

Методи авторизації:

1. Контроль доступу на основі ролей

RBAC або техніка керування доступом на основі ролей надається користувачам відповідно до їхньої ролі чи профілю в організації. Він може бути реалізований як система-система, так і користувач-система.

2. Веб-токен JSON

Веб-токен JSON або JWT — це відкритий стандарт, який використовується для безпечної передачі даних між сторонами у формі об’єкта JSON. Користувачі перевіряються та авторизуються за допомогою пари закритий/відкритий ключ.

3. SAML

SAML розшифровується як Security Assertion Markup Language. Це відкритий стандарт, який надає облікові дані для авторизації постачальникам послуг. Обмін цими обліковими даними здійснюється через XML-документи з цифровим підписом.

4. Авторизація OpenID

Це допомагає клієнтам перевірити особу кінцевих користувачів на основі аутентифікації.

5. OAuth

OAuth — це протокол авторизації, який дозволяє API автентифікувати та отримувати доступ до запитаних ресурсів. [11][12]

Веб-токен JSON

JSON Web Token (JWT) - це відкритий стандарт (RFC 7519), який визначає компактний і автономний спосіб безпечної передачі інформації між сторонами у вигляді об'єкта JSON. Ця інформація може бути перевірена і їй можна довіряти, оскільки вона підписана цифровим підписом. JWT можуть бути підписані за допомогою секретного (за допомогою алгоритму HMAC) або відкритого/закритого ключа з використанням RSA або ECDSA.

Хоча JWT можуть бути зашифровані, щоб також забезпечити таємницю між сторонами, ми зосередимося на підписаних токенах. Підписані токени дозволяють перевірити цілісність тверджень, що містяться в ньому, в той час як зашифровані токени приховують ці твердження від інших сторін. Коли токени підписуються за допомогою пар відкритих/закритих ключів, підпис також засвідчує, що тільки сторона, яка володіє закритим ключем, є тією, хто підписав токен.

Для чого використовувати JWT-токен?

Ось два сценарії, де веб-токени JSON можуть бути корисними:

1. Авторизація: Це найпоширеніший сценарій використання JWT. Після того, як користувач увійшов в систему, кожен наступний запит буде включати JWT, дозволяючи користувачеві отримати доступ до маршрутів, сервісів і ресурсів, які дозволені з цим токеном. Єдиний вхід - це функція, яка широко використовує JWT в наш час, через її невеликі накладні витрати і можливість легко використовувати в різних доменах.

2. Обмін інформацією: Веб-токени JSON - це хороший спосіб безпечної передачі інформації між сторонами. Оскільки JWT можна підписувати, наприклад, за допомогою пар відкритих/закритих ключів, ви можете бути впевнені, що відправники є тими, за кого себе видають. Крім того, оскільки підпис обчислюється за допомогою заголовка і корисних даних, ви також можете перевірити, що вміст не був підроблений.

Структура JWT

JWT складається з трьох основних частин, розділених між собою крапками:

- заголовок (header),

- навантаження (payload),
- підпис (signature).

Заголовок і навантаження формуються окремо, а потім на їхній основі обчислюється підпис.

Таким чином, JWT зазвичай виглядає так:

```
xxxxx.yyyyy.zzzzz
```

Header

Зазвичай заголовок складається з двох частин: типу токена, тобто JWT, і використовуваного алгоритму підпису, наприклад, HMAC SHA256 або RSA.

Наприклад:

```
{
  "alg": "HS256",
  "typ": "JWT"
}
```

Потім цей JSON кодується з Base64Url, щоб сформувати першу частину JWT.

Payload

Payload - це дані, які можна передати в токені, але у стандарті JWT передбачено кілька зарезервованих полів:

- iss (issuer) - видавець токена;
- sub (subject) - "тема", для якої призначений токен;
- aud (audience) - аудиторія, до якої адресований токен;
- exp (expire time) - термін дії токену;
- nbf (not before) - термін, до якого токен є недійсним;
- iat (issued at) - час створення токену;
- jti (JWT id) - ідентифікатор токену.

Хоча всі ці поля не є обов'язковими, використання їх для інших цілей може призвести до колізій.

Важливо пам’ятати, що для підписаних токенів ця інформація, хоча і захищена від підробки, може бути прочитана будь-ким. Не розміщуйте секретну інформацію в корисному навантаженні або елементах заголовку JWT, якщо вона не зашифрована.

Signature

Підпис використовується для перевірки того, що повідомлення не було змінено по дорозі, а у випадку токенів, підписаних приватним ключем, він також може підтвердити, що відправник JWT є тим, за кого він себе видає.

Підпис обчислюється на основі заголовка і навантаження. Таким чином, якщо хтось спробує змінити дані в токені, він не зможе змінити підпис, не знаючи приватного ключа. Під час аутентифікації приватним ключем може виступати пароль користувача (або хеш від пароля).

Спочатку header і payload приводяться до формату JSON, а потім переводяться в base64. Потім, два ці рядки з'єднуються через крапку і хешуються зазначеним у header алгоритмом. Результат роботи алгоритму і є підпис.

Створення токена

Тепер залишилося тільки сформувати сам токен, для цього потрібно через крапку з'єднати header, payload в base64 і підпис.

Результатом є три рядки Base64-URL, розділені крапками, які можна легко передавати в середовищах HTML і HTTP, при цьому вони більш компактні в порівнянні зі стандартами на основі XML, такими як SAML.

Нижче показано приклад JWT, в якому закодовано попередній заголовок і payload, і який підписано секретним ключем (Рис. 5.1).

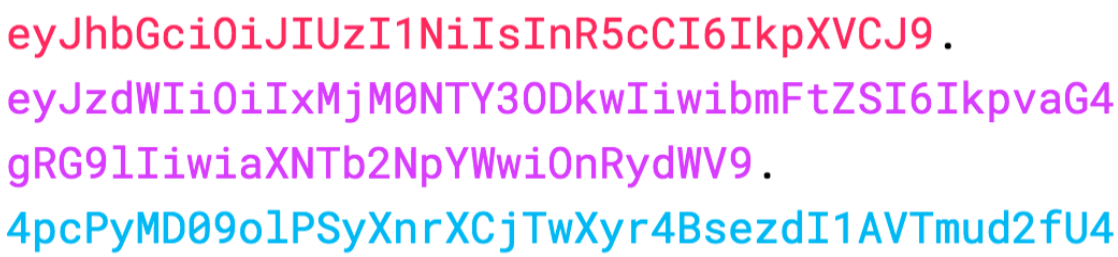

Рис. 5.1. Вигляд JSON веб-токена

Перевірити токен можна на сайті jwt.io, який дозволяє зашифрувати та розшифрувати токен із різною інформацією, секретним ключем та за допомогою різних алгоритмів шифрування. [13]

Токен оновлення (refresh token)

Якщо ми використовуємо токен доступу (access token) протягом тривалого часу, існує ймовірність того, що хакер може вкрасти наш токен і використати його не за призначенням. Тому не дуже безпечно використовувати токен доступу протягом тривалого періоду.

Токени оновлення - це тип токенів, які можна використовувати для отримання нових токенів доступу. Коли термін дії токенів доступу закінчується, ми можемо використовувати токени оновлення, щоб отримати новий токен доступу від контролера автентифікації. Термін дії токенів оновлення зазвичай набагато довший, ніж термін дії токенів доступу.

Зазвичай встановлюється короткий час життя для токенів доступу. Таким чином, навіть якщо хакер використає access token, він отримає доступ до сервісу лише на короткий період часу. Токен оновлення видається разом з токеном доступу із запиту на вхід. Щоразу, коли термін дії токена доступу закінчується, ми можемо отримати новий токен доступу за допомогою токена оновлення.

Щоразу, коли користувач входить у додаток, використовуючи дійсні облікові дані, оновлюється refresh token та термін дії токену в таблиці користувачів у базі даних. Після закінчення терміну дії токену доступу, якщо користувач знову спробує отримати захищений ресурс з додатку, він отримає помилку 401 несанкціонованого доступу. Тоді користувач може спробувати оновити токен, використовуючи поточний токен доступу та токен оновлення. У методі оновлення додаток підтвердить токен, час дії якого вийшов, і токен оновлення. Якщо обидва токени дійсні, програма видасть користувачеві новий токен доступу і токен оновлення. Відповідний користувач може використовувати цей новий токен для доступу до захищених ресурсів у додатку.

Якщо щось пішло не так, токен оновлення може бути відкликаний, а це означає, що коли програма спробує використати його для отримання нового токена доступу, цей запит буде відхилено, і користувачеві доведеться ще раз ввести облікові дані та пройти автентифікацію.

Реалізація автентифікації та авторизації у проєкті

В проекті Habit-Rabbit реалізовано аутентифікацію на основі пароля й авторизацію за допомогою JWT. Аутентифікацію реалізовано на серверній частині проекту. Для цього реалізовано клас AuthenticationService та контролер AuthenticationController (рис. 2.2.8).

Додаток В

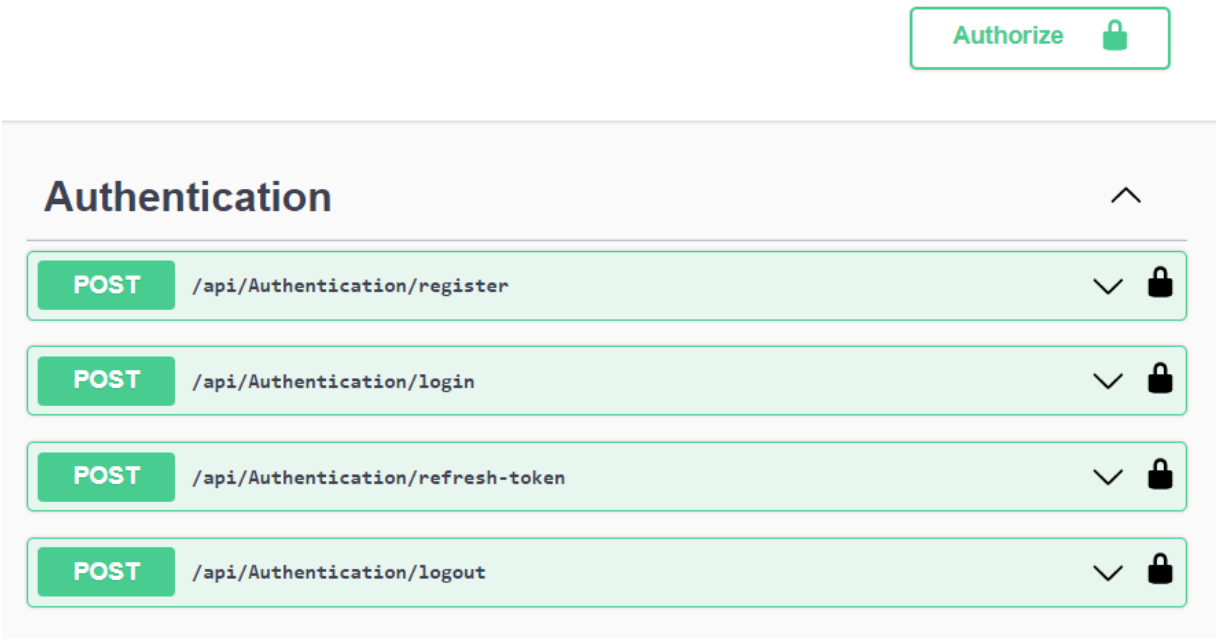

Рис. 5.2. Вигляд доступних запитів AuthenticationController у Swagger UI

На стороні UI користувач вводить облікові дані для входу, які потім передаються на сервер і порівнюються з наявними у базі даних. Після успішної перевірки відбувається авторизація користувача. Для цього на рівні бізнес-логіки генеруються access- та refresh-токени, які потім передаються як http-відповідь на запит для входу користувача в систему. Токени зберігаються у локальному сховищі браузера користувача - local storage, а також refresh-токен зберігається в базі даних. При надсиланні авторизованих запитів з клієнта на сервер, в хедері завжди передається access-токен. Таким чином сервер знає, що користувач авторизований, отже має право надсилати цей запит, а також, що це за користувач (у токені зашифровано id та email).

На стороні клієнта авторизація також представлена компонентом PrivateRoute, в якому перевіряється статус авторизації і, у відповідності до нього, надається доступ до вмісту сторінок. Якщо користувач увійшов в систему, то він може перейти на будь-

яку сторінку, окрім сторінок входу та реєстрації. Якщо ж користувач не пройшов автентифікацію, будь-які спроби зайти на захищені сторінки переадресовуватимуть його на сторінку входу в систему.

3.2.2. Опис бізнес-процесів

Кожного разу, коли користувач хоче набути нову звичку, він ставить перед собою свого роду челендж, результатом якого і буде сформована звичка. Наша задача, спростити це завдання і зробити його максимально комфортним. Щоб це реалізувати, на серверній частині проекту було створено декілька сервісів, які допоможуть юзерам створювати челенджі та проходити їх регулярно, із дня в день, відмічаючи свій прогрес в додатку.

Challenge

Основною функцією для цього є створення такого челенджу (Рис. 6.1.). Тут все дуже просто: користувач вносить назву, опис звички, яку хоче набути, дату початку і дату кінця, частоту повторень (наприклад, кожного понеділка, вівторка і п’ятниці), а також кількісну величину для вимірювання прогресу та одиниці вимірювання (наприклад, 10 сторінок або 6 разів). Додатково, він може обрати іконку для відображення цього челенджу та колір, а також його видимість - бачитиме прогрес лише він, чи, можливо, ще його друзі.

Subtask

Проте, що ж буде, якщо звичка, яку хоче набути користувач - це певний комплекс дій? До прикладу, хтось захоче бігати щодня, готуючись до марафону, але в процес бігу ще має бути включена розминка, правильний перекус і якісь додаткові вправи. Як відслідкувати, чи виконано все із цього списку? Звісно, можна створити для кожного з пунктів окремий челендж. Але це не надто зручно і неможливо буде відстежити прогрес всіх дій в сукупності, а лише кожної окремо. Щоб вирішити цю задачу було створено сутність Subtasks, що означає підзадачі. Відтепер, користувач може в основний челендж додати список завдань, без виконання яких, челендж не можна вважати пройденим. Підзадача по структурі дуже схожа на звичайний челендж, але дещо спрощений. Щоб створити підзадачу достатньо вказати її назву, та величину для

вимірювання прогресу з одиницями вимірювання. Окрім того, для сутностей subtask додаються посилання на челендж, до якого вони належать.

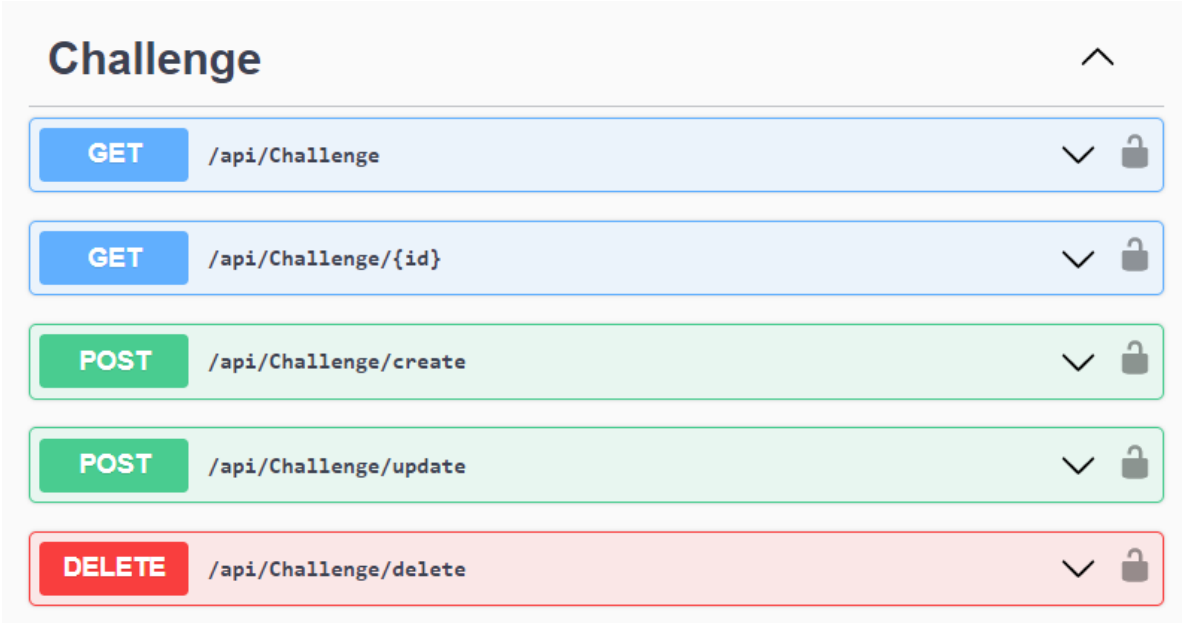

Рис. 6.1. Вигляд доступних запитів для роботи із челенджами у Swagger UI

DailyTask

Під час створення нового челенджу, в базу даних додаються також об’єкти сутності DailyTask. Вони створені для того, щоб користувач кожного дня бачив, які завдання у нього є на сьогодні, та міг відмічати свій прогрес щодня у новому об’єкті, при цьому зберігається прогрес за попередні дні. Оскільки при створенні челенджу одразу створюються щоденні завдання для кожного дня челенджу, користувач також може переглядати dailyTask-и на інші дні, попередні, або майбутні. При цьому, прогрес можна додати тільки для попередніх днів або сьогодні, а для наступних днів цей функціонал заблокований (Рис. 6.2).

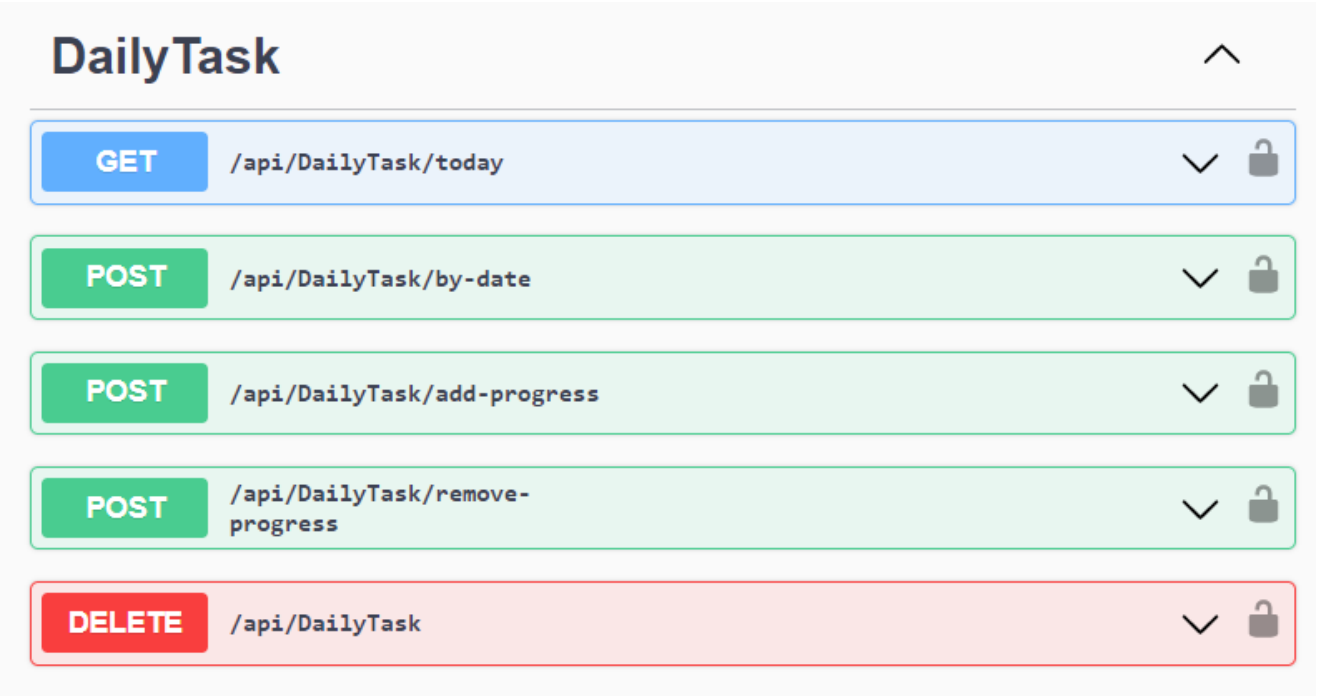

Рис. 6.2. Доступні запити для роботи із щоденними завданнями у Swagger UI

Unit

Для того, щоб користувач міг створити челендж і відслідковувати прогрес, потрібно задати в чому цей прогрес буде вимірюватись і скільки таких одиниць вимірювання потрібно для досягнення мети.

Для цього створено ендпоінт на отримання всіх записів таблиці Units (Рис. 6.3).

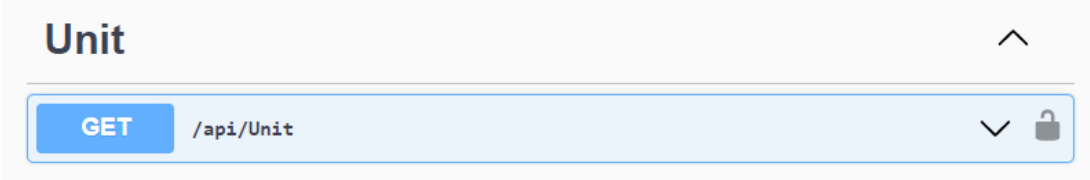

Рис. 6.3. API для отримання доступних одиниць вимірювання

З рисунків видно, що реалізовано також можливість переглядати всі челенджі та завдання на поточний день та за датою, знаходити інформацію про челендж за Id, редагувати його та видаляти, а ще можна додавати, або видаляти прогрес для кожного завдання.

Окрім того, щоб процес набуття звички був приємнішим та цікавішим, було додано елемент гейміфікації у вигляді накопичення балів-морквинок за створення нового челенджу, виконання щодених завдань та успішне завершення челенджів. А також, для зручності моніторингу прогресу додано блок статистики у вигляді графіків.

Висновки до розділу 3

Використання спеціалізованих програмних середовищ, таких як JetBrains Rider для розробки серверної частини, WebStorm для розробки інтерфейсу, Figma для проектування дизайну, а також використання GitHub для контролю версій і ClickUp для спільної роботи та управління проектами, дозволило нашій команді ефективно працювати на кожному етапі розробки, забезпечуючи оптимальність процесу та підвищуючи якість результуючого програмного забезпечення. Використання Swagger для тестування API забезпечило надійність та якість функціонування серверної частини веб-додатка.

У проекті було використано ASP.NET Core, що дозволило створити крос-платформені веб-додатки та сервіси з використанням мови програмування C#. ASP.NET Core забезпечує просту серіалізацію, вбудовану маршрутизацію та потужні інструменти для аутентифікації та авторизації, що сприяє ефективній розробці та високій продуктивності додатків. Крім того, використання Entity Framework та AutoMapper дозволило зручно та ефективно працювати з базою даних, а Microsoft SQL Server забезпечив надійне зберігання та управління даними. Застосування цих технологій сприяло розробці стабільного та функціонального програмного забезпечення.

У проекті Habit-Rabbit реалізовано автентифікацію та авторизацію з використанням JWT. Аутентифікація проводиться на серверній частині проекту, де клас AuthenticationService та контролер AuthenticationController відповідають за перевірку облікових даних користувача. Після успішної аутентифікації генеруються access- та refresh-токени, які передаються користувачу як http-відповідь для використання в подальших запитах.

На стороні клієнта авторизація реалізована за допомогою компонента PrivateRoute, який перевіряє статус авторизації користувача і надає доступ до захищених сторінок. У разі відсутності авторизації користувач буде перенаправлений на сторінку входу в систему.

Використання токенів оновлення дозволяє підтримувати безпеку системи, оскільки їх можна використовувати для отримання нових токенів доступу, замість тривалого використання одного токену доступу.

Загалом, реалізація автентифікації та авторизації у проекті Habit-Rabbit забезпечує безпеку доступу до системи та контроль над правами користувачів.

Бізнес-процеси, описані у третьому розділі, спрощують створення та управління челенджами для формування нових звичок. Вони дозволяють користувачам ставити перед собою цілі, відслідковувати прогрес та досягати їх.

Створено сервіси, які допомагають користувачам створювати челенджі, включаючи визначення назви, опису, тривалості та параметрів вимірювання прогресу. Також реалізована можливість додавання підзадач та відстеження прогресу виконання. Проект також включає можливість редагування, видалення та перегляду челенджів, завдань та прогресу. Користувачі можуть взаємодіяти з системою, вносячи зміни та відмічаючи свій прогрес у досягненні звичок.

В цілому, цей проект надає зручний та ефективний спосіб створювати челенджі та досягати поставлених цілей, сприяючи формуванню нових звичок у користувачів.

ВИСНОВОК

Звички грають велику роль у нашому житті, впливаючи на нашу повсякденність і якість життя в цілому. Вони можуть бути позитивними або негативними, і формування здорових звичок може вимагати зусиль. Проект Habit-Rabbit є веб-додатком, який надає інструменти для створення і відстеження звичок. Це зручний спосіб автоматизувати процес формування нових звичок та позбутися негативних. Habit-Rabbit дозволяє користувачам створювати трекери звичок та відстежувати їх прогрес на комп'ютері або ноутбуці, надаючи зручну альтернативу паперовому плануванню або використанню мобільних додатків. Це допомагає зосередитися на важливих справах і досягненні особистих цілей без надмірного використання телефону.

У результаті виконання кваліфікаційної роботи було розроблено серверну частину веб-додатку для допомоги у формуванні та позбутті звичок з елементами гейміфікації Habit-Rabbit.

Для досягнення поставленої мети було здійснено аналіз існуючих рішень і на основі цього аналізу розроблено технічне завдання для створення застосунку. Було вирішено, що веб-застосунок Habit-Rabbit має вирізнятися простим, зрозумілим та привабливим дизайном, помірною гейміфікацією процесу набування звички, функціоналом для відслідковування широкого спектру звичок та можливістю спостерігати за своїм прогресом за допомогою візуалізованої аналітики.

Основу цільової аудиторії складають люди, які переважну більшість часу проводять, працюючи за комп’ютером. Саме для цього проєкт реалізовано саме у вигляді веб-застосунку, що зручно для тих, хто хоче сконцентруватися на роботі чи навчанні, не відволікаючись на сторонні девайси. Веб-застосунок Habit-Rabbit надає користувачам весь необхідний функціонал, щоб успішно формувати нові та позбуватися старих звичок. Для цього можна створювати челенджі з детальним описом, що полегшує процес відстеження прогресу, адже можна виконувати завдання частинами і одразу вносити дані про виконання у систему.

Для організації командної роботи було розподілено ролі та відповідальність кожного члена команди. А також було використано такі сервіси, як GitHub та ClickUp, які значно спрощують одночасну розробку продукту кількома людьми.

Використання сервісу drawio.com дозволило команді ефективно побудувати діаграму прецедентів та забезпечити однакове розуміння задачі серед усіх членів команди. Use-case діаграма проєкту Habit-Rabbit надає загальне уявлення про функціонал системи та взаємодію з різними типами користувачів. Вона показує, що гостям системи доступні обмежені можливості, такі як реєстрація і вхід в систему, тоді як авторизованим користувачам надається повний доступ до всіх функцій системи.

У проєкті Habit-Rabbit було використано реляційну базу даних, оскільки вона забезпечує збереження даних і зв'язків між ними. Для реалізації бази даних було обрано підхід Code-First, де визначалися сутності та їх зв'язки у коді, а потім на основі цього коду база даних створювалася. Застосування Fluent API configuration дозволило встановити зв'язки між сутностями.

Забезпечення взаємодії з базою даних відбувалося через налаштування підключення у файлі appsettings.json. Використання міграцій Code First спрощувало процес розробки та тестування, дозволяючи відстежувати та застосовувати зміни до бази даних.

Для побудови архітектури рішення було вирішено використати підхід Clean Architecture, що надає гнучкість проєкту із можливістю легко змінити клієнтський фреймворк чи базу даних у будь-який момент. Для цього проєкт було розділено на три рівні: Core, Infrastructure та API. Додатково, було використано патерн Репозиторій, який дозволяє абстрагувати доступ до даних від решти системи та полегшує роботу з базою даних. Такі важливі аспекти безпеки, як автентифікація та авторизація, втілили за допомогою пароля та JSON веб-токенів.

В роботі також було наведено стек технологій, використаний для написання коду програми, проведено аналіз переваг та недоліків і обгрунтовано вибір кожної з технологій. Для розробки серверної частини даного рішення було обрано ASP.NET Core платформу від Microsoft разом з MS SQL системою керування базами даних. Додатково, для зручної трансформації даних між сутностями бази даних та об’єктами,

з якими працює API застосовано Automapper, також Entity Framework Core використовувався для створення шару доступу до даних.

Використання спеціалізованих програмних середовищ, таких як JetBrains Rider для розробки серверної частини, WebStorm для розробки інтерфейсу, Figma для проектування дизайну, а також використання GitHub для контролю версій і ClickUp для спільної роботи та управління проектами, дозволило нашій команді ефективно працювати на кожному етапі розробки, забезпечуючи оптимальність процесу та підвищуючи якість результуючого програмного забезпечення. Використання Swagger для тестування API забезпечило надійність та якість функціонування серверної частини веб-додатка.

Реалізація бізнес-процесів у проекті Habit-Rabbit забезпечує користувачам зручний та ефективний спосіб створювати челенджі та досягати поставлених цілей шляхом формування нових звичок.

У проекті створено сервіси, які допомагають користувачам створювати челенджі з визначенням назви, опису, тривалості та параметрів вимірювання прогресу. Користувачі мають можливість додавати підзадачі до кожного челенджу та відстежувати прогрес їх виконання.

Крім того, в проекті реалізована можливість редагування, видалення та перегляду челенджів, завдань та прогресу. Це дозволяє користувачам взаємодіяти з системою, змінюючи та відмічаючи свій прогрес у досягненні звичок.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Опис застосунку Habitica URL: https://habitica.com/static/features
   (дата звернення: 12.03.2023)
2. Article: Rider URL: https://www.jetbrains.com/rider/
   (дата звернення: 20.03.2023)
3. Article: WebStorm URL: https://www.jetbrains.com/webstorm/
   (дата звернення: 20.03.2023)
4. Article: What is ASP.NET?
   URL: https://dotnet.microsoft.com/en-us/learn/aspnet/what-is-aspnet
   (дата звернення: 01.04.2023)
5. Article: APIs with ASP.NET Core
   URL: https://dotnet.microsoft.com/en-us/apps/aspnet/apis
   (дата звернення: 01.04.2023)
6. Article: What is EntityFramework
   URL: https://www.entityframeworktutorial.net/what-is-entityframework.aspx
   (дата звернення: 05.04.2023)
7. Article: Automapper. Get started.
   URL: https://docs.automapper.org/en/stable/Getting-started.html
   (дата звернення: 06.04.2023)
8. Стаття: Як будувати UML-діаграми. - Юлія Каграманова
   URL: https://dou.ua/forums/topic/40575/
   (дата звернення: 15.04.2023)
9. Article: Clean Architecture in ASP.Net Core Web API URL:
   https://www.c-sharpcorner.com/article/clean-architecture-in-asp-net-core-web-api/
   (дата звернення: 15.04.2023)
10. Article: Repository Pattern
    URL: https://deviq.com/design-patterns/repository-pattern
    (дата звернення: 25.04.2023)

11. Article: Authentication vs Authorization – What's the Difference?
URL: https://www.freecodecamp.org/news/whats-the-difference-between-authentication-and-authorisation/
(дата звернення: 26.04.2023)
12. Article: Difference between Authentication and Authorization
URL: https://www.javatpoint.com/authentication-vs-authorization
(дата звернення: 28.04.2023)
13. Article: Introduction to JSON Web Tokens URL: https://jwt.io/introduction/
(дата звернення: 01.05.2023)

ДОДАТОК А

Сутності веб-застосунку Habit-Rabbit

Лістинг програми

Аркушів 3

Острог 2023

Challenge.cs

```
public class Challenge
{
    public int Id { get; set; }
    public string CreatedById { get; set; }
    public User CreatedBy { get; set; }
    public string Title { get; set; }
    public string Description { get; set; }
    public int CountOfUnits { get; set; }
    public int UnitId { get; set; }
    public Unit Unit { get; set; }
    public string Frequency { get; set; }
    public string IconName { get; set; }
    public string Color { get; set; }
    public int ChallengeTypeId { get; set; }
    public ChallengeType ChallengeType { get; set; }
    public int VisibilityId { get; set; }
    public Visibility Visibility { get; set; }
    public DateTimeOffset StartDate { get; set; }
    public DateTimeOffset EndDate { get; set; }
    public bool IsCompleted { get; set; }

    public ICollection<DailyTask> DailyTasks { get; set; }
    public ICollection<Subtask> Subtasks { get; set; }
    public ICollection<User> Users { get; set; }
}
```

ChallengeType.cs

```
public class ChallengeType
{
    public int Id { get; set; }
    public string Type { get; set; }

    public ICollection<Challenge> Challenges { get; set; }
}
```

DailyTask.cs

```
public class DailyTask
{
    public int Id { get; set; }
    public int ChallengeId { get; set; }
    public Challenge Challenge { get; set; }
    public int? SubtaskId { get; set; }
    public Subtask? Subtask { get; set; }
    public DateTimeOffset AssignedDate { get; set; }
    public int CountOfUnitsDone { get; set; }
    public bool IsDone { get; set; }
}
```

Subtask.cs

```
public class Subtask
{
    public int Id { get; set; }
    public string Title { get; set; }
    public int UnitId { get; set; }
    public Unit Unit { get; set; }
    public int ChallengeId { get; set; }
    public Challenge Challenge { get; set; }
    public int CountOfUnits { get; set; }

    public ICollection<DailyTask> DailyTasks { get; set; }
}
```

RefreshToken.cs

```
namespace Core.Entities;

public class RefreshToken
{
    public int Id { get; set; }
    public string Token { get; set; }
    public string UserId { get; set; }
    public User User { get; set; }
}
```

Unit.cs

```
public class Unit
{
    public int Id { get; set; }
    public string Type { get; set; }
    public string ShortType { get; set; }

    public ICollection<Challenge> Challenges { get; set; }
    public ICollection<Subtask> Subtasks { get; set; }
}
```

User.cs

```
using Microsoft.AspNetCore.Identity;

namespace Core.Entities;

public class User : IdentityUser
{
    public int Points { get; set; }
    public ICollection<Challenge> AuthoredChallenges { get; set; }
    public ICollection<Challenge> Challenges { get; set; }
    public ICollection<RefreshToken> RefreshTokens { get; set; }

}
```

Visibility.cs

```
public class Visibility
{
    public int Id { get; set; }
    public string Type { get; set; }

    public ICollection<Challenge> Challenges { get; set; }
}
```

ДОДАТОК Б

Реалізація патерну Репозиторій

Лістинг програми

Аркушів 3

Острог 2023

IRepository.cs

```
using System.Linq.Expressions;

namespace Core.Interfaces;

public interface IRepository<TEntity> where TEntity : class
{
    Task<IEnumerable<TEntity>> GetAllAsync();
    Task<TEntity> GetByIdAsync<TKey>(TKey id);
    Task<TEntity> AddAsync(TEntity entity);
    Task<IList<TEntity>> AddRangeAsync(IList<TEntity> entities);
    Task DeleteAsync(TEntity entityToDelete);
    Task DeleteRange(IEnumerable<TEntity> entitiesToDelete);
    Task UpdateAsync(TEntity entityToUpdate);
    Task<int> SaveChangesAsync();
    IQueryable<TEntity> Query(params Expression<Func<TEntity, object>>[]
includes);
}
```

Repository.cs

```
using System.Linq.Expressions;
using Ardalis.Specification.EntityFrameworkCore;
using Core.Interfaces;
using Infrastructure.Data;
using Microsoft.EntityFrameworkCore;

namespace Infrastructure.Repository;

public class Repository<TEntity> : RepositoryBase<TEntity>,
IRepository<TEntity> where TEntity : class
{
    private AppDbContext _context;
    private DbSet<TEntity> _dbSet;

    public Repository(AppDbContext context) : base(context)
    {
        _context = context;
        _dbSet = _context.Set<TEntity>();
```

```
}

public async Task DeleteAsync(TEntity entityToDelete)
{
    await Task.Run(() =>
    {
        if (_context.Entry(entityToDelete).State == EntityState.Detached)
        {
            _dbSet.Attach(entityToDelete);
        }
        _dbSet.Remove(entityToDelete);
    });
}

public Task DeleteRange(IEnumerable<TEntity> entitiesToDelete)
{
    _dbSet.RemoveRange(entitiesToDelete);
    return Task.CompletedTask;
}
public async Task<IEnumerable<TEntity>> GetAllAsync()
{
    return await _dbSet.ToListAsync();
}

public async Task<TEntity> GetByIdAsync<TKey>(TKey id)
{
    return await _dbSet.FindAsync(id);
}

public async Task<TEntity> AddAsync(TEntity entity)
{
    return (await _dbSet.AddAsync(entity)).Entity;
}

public async Task<IList<TEntity>> AddRangeAsync(IList<TEntity> entities)
{
    await _dbSet.AddRangeAsync(entities);
    await _context.SaveChangesAsync();
```

```
        return entities.ToList();
    }

    public async Task UpdateAsync(TEntity entityToUpdate)
    {
        await Task.Run(() =>
        {
            _dbSet.Attach(entityToUpdate);
            _context.Entry(entityToUpdate).State = EntityState.Modified;
        });
    }

    public async Task<int> SaveChangesAsync()
    {
        return await _context.SaveChangesAsync();
    }

    public IQueryable<TEntity> Query(params Expression<Func<TEntity,
object>>[] includes)
    {
        var query = includes
            .Aggregate<Expression<Func<TEntity, object>>,
IQueryable<TEntity>>(_dbSet, (current, include) => current.Include(include));

        return query;
    }
}
```

ДОДАТОК В

Реалізація автентифікації та авторизації

Лістинг програми

Аркушів 6

Острог 2023

IAuthenticationService.cs

```
using Core.DTO.UserDTO;

namespace Core.Interfaces.Services;

public interface IAuthenticationService
{
    Task<UserAuthorizationDTO> RegisterAsync(UserRegistrationDTO data);
    Task<UserAuthorizationDTO> LoginAsync(UserLoginDTO data);
    Task<UserAuthorizationDTO> RefreshTokenAsync(UserAuthorizationDTO
userTokensDTO);
    Task LogoutAsync(UserLogoutDTO userLogoutDTO);
}
```

AuthenticationService.cs

```
using System.Net;
using System.Text;
using AutoMapper;
using Core.DTO.UserDTO;
using Core.Entities;
using Core.Exceptions;
using Core.Interfaces;
using Core.Interfaces.Services;
using Microsoft.AspNetCore.Identity;
namespace Core.Services;

public class AuthenticationService : IAuthenticationService
{
    private readonly UserManager<User> _userManager;
    private readonly IMapper _mapper;
    private readonly IJwtService _jwtService;
    private readonly IRepository<RefreshToken> _refreshTokenRepository;

    public AuthenticationService(
        UserManager<User> userManager,
        IMapper mapper,
        IJwtService jwtService,
        IRepository<RefreshToken> refreshTokenRepository)
```

```
    {
        _userManager = userManager;
        _mapper = mapper;
        _jwtService = jwtService;
        _refreshTokenRepository = refreshTokenRepository;
    }

    public async Task<UserAuthorizationDTO> RegisterAsync(UserRegistrationDTO
userData)
    {
        var user = _mapper.Map<User>(userData);
        var result = await _userManager.CreateAsync(user, userData.Password);
        if (!result.Succeeded)
        {
            var messageBuilder = new StringBuilder();
            foreach (var error in result.Errors)
            {
                messageBuilder.AppendLine(error.Description);
            }
            throw new HttpException(messageBuilder.ToString(),
HttpStatusCode.BadRequest);
        }
        var loginDTO = new UserLoginDTO
        {
            Email = user.Email,
            Password = userData.Password
        };
        return await LoginAsync(loginDTO);
    }

    public async Task<UserAuthorizationDTO> LoginAsync(UserLoginDTO data)
    {
        var user = await _userManager.FindByEmailAsync(data.Email) ??
                   await _userManager.FindByNameAsync(data.Email);
        if (user == null || !await _userManager.CheckPasswordAsync(user,
data.Password))
        {
            throw new HttpException("Incorrect login or password",
HttpStatusCode.Unauthorized);
```

```
        }
        return await GenerateUserTokens(user);
    }

    public async Task LogoutAsync(UserLogoutDTO userLogoutDTO)
    {
        var refreshToken = _refreshTokenRepository.Query().SingleOrDefault(t
=> t.Token == userLogoutDTO.RefreshToken);
        if (refreshToken == null)
        {
            throw new HttpException("Invalid Token",
HttpStatusCode.NotFound);
        }

        await _refreshTokenRepository.DeleteAsync(refreshToken);
        await _refreshTokenRepository.SaveChangesAsync();
    }

    private async Task<UserAuthorizationDTO> GenerateUserTokens(User user)
    {
        var claims = _jwtService.SetClaims(user);
        var token = _jwtService.CreateToken(claims);
        var refreshToken = await CreateRefreshToken(user.Id);
        user.RefreshTokens.Add(refreshToken);
        var tokens = new UserAuthorizationDTO
        {
            AccessToken = token,
            RefreshToken = refreshToken.Token
        };
        return tokens;
    }

    private async Task<RefreshToken> CreateRefreshToken(string userId)
    {
        var refreshToken = _jwtService.GenerateRefreshToken();
        var refreshTokenEntity = new RefreshToken
        {
            Token = refreshToken,
            UserId = userId
```

```
        };
        await _refreshTokenRepository.AddAsync(refreshTokenEntity);
        await _refreshTokenRepository.SaveChangesAsync();
        return refreshTokenEntity;
    }

    public async Task<UserAuthorizationDTO>
RefreshTokenAsync(UserAuthorizationDTO userTokensDTO)
    {
        var refreshToken = GetRefreshToken(userTokensDTO.RefreshToken);
        var claims =
_jwtService.GetClaimsFromExpiredToken(userTokensDTO.AccessToken);
        var newToken = _jwtService.CreateToken(claims);
        var newRefreshToken = _jwtService.GenerateRefreshToken();
        refreshToken.Token = newRefreshToken;
        await _refreshTokenRepository.UpdateAsync(refreshToken);
        await _refreshTokenRepository.SaveChangesAsync();
        var tokens = new UserAuthorizationDTO
        {
            AccessToken = newToken,
            RefreshToken = refreshToken.Token
        };
        return tokens;
    }

    private RefreshToken GetRefreshToken(string token)
    {
        var refreshToken = _refreshTokenRepository
            .Query()
            .FirstOrDefault(t => t.Token == token);

        if (refreshToken == null)
        {
            throw new HttpException("Invalid Token",
HttpStatusCode.NotFound);
        }
        return refreshToken;
    }
}
```

AuthenticationController.cs

```
using Microsoft.AspNetCore.Mvc;
using Core.DTO.UserDTO;
using Core.Interfaces.Services;
using Microsoft.AspNetCore.Authorization;

namespace API.Controllers;

[Route("api/[controller]")]
[ApiController]
public class AuthenticationController : Controller
{
    private readonly IAuthenticationService _authenticationService;

    public AuthenticationController(
        IAuthenticationService authenticationService)
    {
        _authenticationService = authenticationService;
    }

    [HttpPost("register")]
    public async Task<ActionResult> Register([FromBody] UserRegistrationDTO
data)
    {
        var tokens = await _authenticationService.RegisterAsync(data);
        return Ok(tokens);
    }

    [HttpPost("login")]
    public async Task<ActionResult> Login([FromBody] UserLoginDTO data)
    {
        var tokens = await _authenticationService.LoginAsync(data);
        return Ok(tokens);
    }

    [HttpPost("refresh-token")]
    public async Task<IActionResult> RefreshTokenAsync([FromBody]
UserAuthorizationDTO userTokensDTO)
```

```
    {
        var tokens = await
_authenticationService.RefreshTokenAsync(userTokensDTO);
        return Ok(tokens);
    }

    [Authorize]
    [HttpPost("logout")]
    public async Task<IActionResult> LogoutAsync([FromBody] UserLogoutDTO
userLogoutDTO)
    {
        await _authenticationService.LogoutAsync(userLogoutDTO);
        return Ok();
    }
}
```