НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
“КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ
імені ІГОРЯ СІКОРСЬКОГО”

Факультет інформатики та обчислювальної техніки

Кафедра обчислювальної техніки

До захисту допущено:
Завідувач кафедри
_____ Сергій СТІРЕНКО
(підпис)
“__” _____2022 р.

Дипломний проєкт

на здобуття ступеня бакалавра
за освітньо-професійною програмою “Комп’ютерні системи та мережі”
спеціальності 123 “Комп’ютерна інженерія”

на тему: *Багатокористувацька система ведення словників*

Виконав : студент __4__ курсу, групи _ІО-83_
(шифр групи)
_____Мітячкін Денис Євгенійович_____
(прізвище, ім’я, по батькові) (підпис)

Керівник _____асист. Молчанова А.А._____
(посада, науковий ступінь, вчене звання, прізвище та ініціали) (підпис)

Консультант (нормоконтроль) професор, д.т.н., Сімоненко В. П._____
(назва розділу) (посада, вчене звання, науковий ступінь, прізвище та ініціали) (підпис)

Рецензент _____
(посада, науковий ступінь, вчене звання, прізвище та ініціали) (підпис)

Засвідчую, що у цьому дипломному проєкті немає запозичень з праць інших авторів без відповідних посилань.
Студент _____
(підпис)

Київ – 2022 р.

НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
“КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ
імені ІГОРЯ СІКОРСЬКОГО”

Факультет інформатики та обчислювальної техніки

Кафедра обчислювальної техніки

Рівень вищої освіти – перший (бакалавр)

Освітньо-професійна програма

“Комп’ютерні системи та мережі”

спеціальності 123 “Комп’ютерна Інженерія”

ЗАТВЕРДЖУЮ
Завідувач кафедри
_____ Сергій СТІРЕНКО
(підпис)
“__” _____2021 р.

ЗАВДАННЯ
на бакалаврський дипломний проєкт студента

Мітячкіна Денис Євгенійовича

1. Тема проєкту *Багатокористувацька система ведення словників*
керівник проєкту _____ асист. Молчанова А. А.,
(прізвище, ім’я, по батькові, науковий ступінь, вчене звання)
затверджені наказом по університету від 11 травня 2021 року №1139-с

2. Термін здачі студентом закінченого проєкту *30 червня 2022 р.*

3. Вихідні дані до проєкту *технічна документація, теоретичні дані.*

4. Зміст розрахунково-пояснювальної записки (перелік питань, які розробляються)

Розділ 1. Актуальність та огляд існуючих рішень.

Розділ 2 Вибір і обґрунтування засобів реалізації програми.

Розділ 3. Розробка додатку для поліпшення харчового раціону людини.

Розділ 4. Тестування розробленого додатку.

5. Перелік графічного матеріалу (з точним позначенням обов'язкових креслень) структурна схема – діаграма структури системи, функціональна схема – діаграма програмних компонентів, принципова схема – алгоритм роботи програми.

6. Консультанта проєкту, з вказівкою розділів проєкту, які до них вносяться

Розділ	Консультант	Підпис, дата	
		Завдання видав	Завдання прийняв
Нормоконтрол	Сімоненко В. П.		

7. Дата видачі завдання «12» грудня 2022 р.

Календарний план

№ п/п	Найменування етапів дипломного проєкту	Терміни виконання етапів проєкту	Примітки
1.	*Затвердження теми проєкту*	*19.12.2022-21.12.2022*	
2.	*Вивчення та аналіз завдання*	*20.12.2022-20.12.2022*	
3.	*Розробка архітектури та загальної структури системи*	*20.04.2022-25.04.2022*	
4.	*Розробка структур окремих підсистем*	*25.04.2022-28.04.2022*	
5.	*Програмна реалізація системи*	*28.04.2022-15.05.2022*	
6.	*Оформлення пояснювальної*	*15.05.2022-26.05.2022*	
7.	*Захист програмного продукту*	*05.06.2022*	
8.	*Передзахист*	*23.05.2022*	
9.	*Захист*	*14.06.2022*	

Студент-дипломник _____ Денис МІТЯЧКІН
(підпис)

Керівник проєкту _____ Анастасія МОЛЧАНОВА
(підпис)

АНОТАЦІЯ

У бакалаврському дипломному проекті спроектовано і реалізовано багатокористувацьку систему ведення словників з можливістю поширення власних словників іншим користувачам.

Програма дозволяє користувачам, авторизуватись, взаємодіяти з словниками, налаштовувати акаунт, здійснювати пошук інших користувачів, імпортувати словники з інших програм.

Застосунок був розроблений у вигляді веб-застосунку на мові JavaScrypt з використанням Node.js для серверної частини. Користувацький інтерфейс розроблено з застосуванням технології React.js.

ANNOTATION

The bachelor's degree project designed and implemented a multi-user dictionary system with the ability to distribute their own dictionaries to other users.

The program allows users to sign up, interact with dictionaries, set up an account, search for other users, import dictionaries from other programs.

The application was developed as a web application in JavaScrypt using Node.js for the server part. The user interface was developed using React.js technology.

ОПИС АЛЬБОМУ
ДО ДИПЛОМНОГО ПРОЄКТУ

на тему: «Багатокористувацька система ведення словників»

Київ – 2022

справки	Формат	*Значення*	*Найменування*	Кіл. листів	№ екземпля	*Додаток*
			Документація загальна			
			Знову розроблена			
	А4	***ІАЛЦ.467200.002 ТЗ***	Багатокористувацька	3		
			система ведення			
			словників			
			Технічне завдання			
	А4	***ІАЛЦ.467200.003 ПЗ***	Багатокористувацька	68		
			система ведення			
			словників			
			Пояснювальна записка			
	А4	***ІАЛЦ.467200.004 Д1***	Багатокористувацька	1		
			система ведення			
			словників			
			Структурна схема			
			системи			
	А4	***ІАЛЦ.4672008.005 Д2***	Багатокористувацька	1		
			система ведення			
			словників			
			Функціональна схема -			
	А4	***ІАЛЦ.467200.006 Д3***	Багатокористувацька	1		
			система ведення			
			словників			
			Алгоритм дій програмного			
			забезпечення			
	А4	***ІАЛЦ.467200.007 Д4***	Багатокористувацька	27		
			система ведення			
			словників			
			Текст програмного коду			

					# *ІАЛЦ.467200.001 ОА*	
Зм	*Лист*	*№ докум.*	*Підп*	*Дата*		
Розроб		Мітячкін Д. Є			*Багатокористувацька систеа ведення словників* Опис альбому	Літ. / Аркуш 1 / Аркушів 1
Перев		Молчанова А.А..				
					НТУУ “КПІ” ФІОТ ІО-83	

ТЕХНІЧНЕ ЗАВДАННЯ
ДО ДИПЛОМНОГО ПРОЄКТУ

на тему: «Багатокористувацька система ведення словників»

Київ – 2022

ЗМІСТ

1. НАЙМЕНУВАННЯ ТА ОБЛАСТЬ ЗАСТОСУВАННЯ

Технічне завдання поширюється на розробку програмного забезпечення потреб користувачів та бізнесу.

Область застосування: використання з метою оптимізації робочого, навчального процесів користувачів з надмірним потоком інформації.

2. ПІДСТАВИ ДЛЯ РОЗРОБКИ

Підставою для розробки системи є завдання виконання роботи кваліфікаційно-освітнього рівня «бакалавр комп’ютерної інженерії», який був затверджений факультетом “Інформатики та обчислювальної техніки” кафедрою обчислювальної техніки Національного технічного Університету України «Київський Політехнічний інститут ім. Ігоря Сікорського».

3. МЕТА І ПРИЗНАЧЕННЯ РОЗРОБКИ

Метою розробки системи є часткове вирішення проблем оптимізації робочого процесу учнів, вирішення проблеми поширення структурованої інформації серед інших користувачів.

4. ДЖЕРЕЛА РОЗРОБКИ

Джерелом розробки системи є технічні документації, науково технічні література, публікації у мережі Інтернет, наукові статті та розроблені програми з застосуванням мови JavaScrypt.

5. ТЕХНІЧНІ ВИМОГИ

5.1. Вимоги до розробленого продукту

Розроблена система має виконувати такі вимоги:

- Простий і інтуїтивно-зрозумілий інтерфейс системи.
- Надати можливість користувачам передавати особисті дані для показу статистики на основі придбаних продуктів.
- Надати можливість користувачам власноруч вносити продукти та формувати статистику за вибраним періодом дат.
- Надати можливість користувачам власноруч обирати дієту та шукати потрібні рецепти.
- Надати вичерпну та зрозумілу документацію для розробленого продукту.

5.2. Вимоги до програмного забезпечення

- Операційна система Android або IOS.
- Встановлений клієнт EXPO на мобільному пристрої.

5.3. Вимоги до апаратної частини

- Не менше 3 Гб оперативної пам’яті.
- Обсяг вільного простору на жорсткому диску не менше 1Гб.

6. ЕТАПИ РОЗРОБКИ

Назва етапів виконання	Термін виконання
Затвердження теми роботи	*19.12.2022-21.12.2022*
Вивчення та аналіз завдання	*20.12.2022-20.12.2022*
Розробка архітектури та загальної структури системи	*20.04.2022-25.04.2022*
Розробка структур окремих частин системи	*25.04.2022-28.04.2022*
Програмна реалізація системи	*28.04.2022-15.05.2022*
Виправлення помилок	15.04.2022-20.05.2022
Оформлення пояснювальної записки	20.05.2022

ПОЯСНЮВАЛЬНА ЗАПИСКА
ДО ДИПЛОМНОГО ПРОЄКТУ

на тему: «Багатокористувацька система ведення словників»

Київ – 2022

ЗМІСТ

ПЕРЕЛІК СКОРОЧЕНЬ

API	Прикладний програмний інтерфейс (з англійської Application Programming Interface)
SQL	Мова структурованих запитів (англ. Structured Query Language)
IDE	Інтегроване середовище розробки (англ. Integrated development environmen)
ORM	Об'єктно-реляційна проєкція (англ. Object-relational mapping)

ВСТУП

У сучасному світі людям доводиться мати справу з величезними обсягами інформації. Її обсяги з кожним десятиліттям стають все більшими і все частіше з’являється необхідність цю інформацію обробляти та структурувати.

Зокрема, є таке, що людям потрібно зберігати текстову інформацію, щоб було зручно її вивчати, запам'ятовувати, повторювати, робити нагадування і так далі. Для цього потрібні засоби, зручні інструменти, використовуючи які, користувач зможе без зайвих дій отримати структуровану та відфільтровану інформацію. Одним з найзручніших засобів є система словників. Вони зручні для використання, вони можуть мати ілюстрації, посилання на детальну інформацію конкретних авторів, їх можна сортувати, що значно полегшить пошук і, що досить важливо, ними можна ділитись з іншими користувачами.

Відповідно, метою є розробка багатокористувацької системи ведення словників, яка може вирішити проблему з величезними обсягами неструктурованої інформації.

Вирішено зробити веб-сервіс що дасть можливість створювати власні словники з даними користувача.

В результаті, отримано сервіс, що є доступним з браузерів які є у більшості сучасних пристроїв. Це дозволяє відразу почати користуватись сервісом а дані можна використати відразу з усіх можливих пристроїв. Застосунок виконаний у сучасному оформленні та зі зручним сучасним інтерфейсом.

РОЗДІЛ 1. АКТУАЛЬНІСТЬ ТА ОГЛЯД НАЯВНИХ РІШЕНЬ

1.1 Актуальність багатокористувацьких систем ведення словників

У наш час, досить актуальною стала тема зберігання важливих (або не дуже важливих) даних у місцях, до яких без будь-яких можливих проблем можна вернутися з метою відновити цю інформацію у пам’яті або ж використати її з іншою метою. Стали все частіше з’являтись програми-записники де можна у вигляді тесту записати усе що необхідно. І такі програми зазвичай є швидкими, займають мало місця і є легкими у ознайомленні. Зараз такі програми є у кожному смартфоні і з’явилось чимало вебзастосунків зі схожим призначенням. Але програми такого виду мали певний недолік. Попри всі свої переваги, вони не могли дати користувачеві можливість структурувати свої записи належним чином а також були розраховані лише на одну особу. Програми-записники були розраховані на те що користувач буде нотувати і потім в своїх цілях використовувати дані записи але ніяк не більше. Такого виду програми було досить тяжко використовувати для людей що наприклад вивчають іноземні мови або ж ведуть словники дат народження чи рецептів і так далі оскільки дані перемішувались і обробляти їх належним чином ставало більш незручно ніж самотужки їх записувати на аркуші паперу. Взявши до уваги дану проблему розробники почали все частіше вкладати ресурси у розвиток систем ведення словників. Прикладом такої системи є програма Cambridge Dictionary (рис. 1.1)

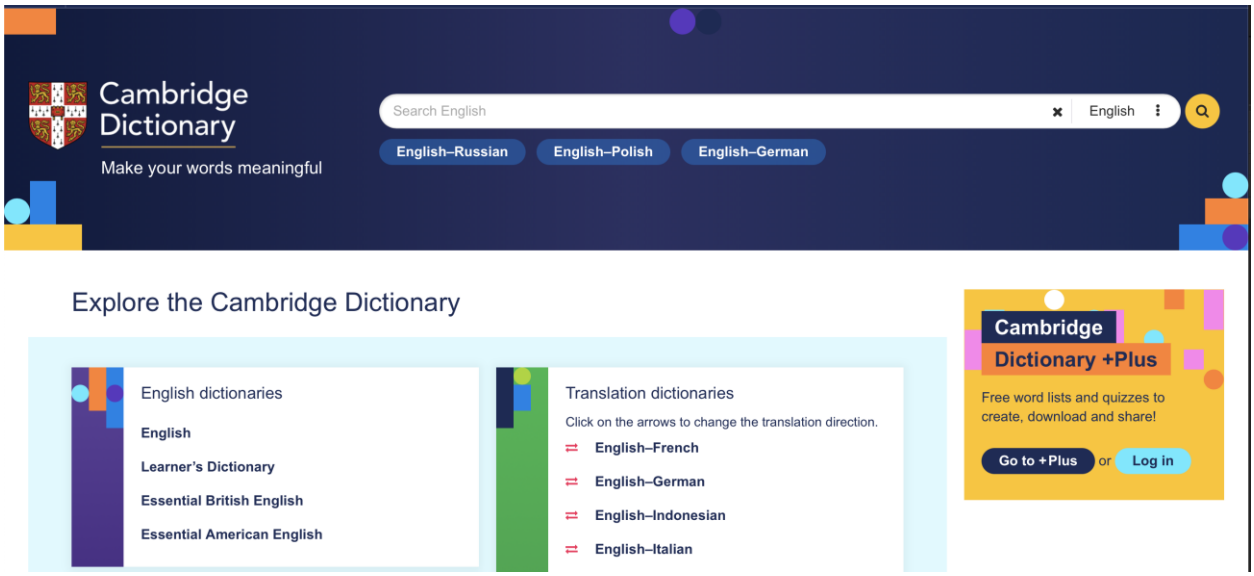


Рисунок 1.1 – Приклад одного з найпопулярніших англомовних словників у світі Cambridge Dictionary

Система ведення словників – це узагальнена назва для програм що використовують у собі функціональність що дає можливість зберігати дані структуруючи їх у вигляді таблиці, списку чи усім відомого словника що має визначення та значення. Цей метод зберігання даних досить популярний у сучасних програмах що призначені для вивчення іншомовних слів а також програм для запису задач і створення структурованих списків, іншими словами – записників.

Такого виду системи були чимось схожими на записники але мали досить важливу особливість що вирізняла їх серед даної категорії. Це структурованість. Системи ведення словників дозволяють відділити усю необхідну інформацію у певну категорію та маніпулювати нею у межах словника. Завдяки такому підходу став більш швидким та зручним пошук необхідних записів серед купи інших а також з’явилась можливість програмно задати дії до певних категорій словників. Як приклад, якщо у записнику користувач записує просто список іншомовних слів і вчить їх просто закриваючи рукою переклад то словник дасть можливість структурувати слова як ключі а переклад як значення. Тоді людина що користується словником може

буде вчити слова бачачи на екрані лише іншомовне слово і лише за необхідності натискати на кнопку перекладу щоб згадати переклад. Також такі програми зазвичай дають можливість випадковим чином видавати такі слова для повторення і мають чимало додаткових функцій що лише покращують процес вивчення. Як приклад, це функція призначення інших користувачів під конкретний запис у словнику, будь то задача чи замітка (рис. 1.2). Саме структурованість та багатофункціональність вирізняє програми словники серед інших категорій.

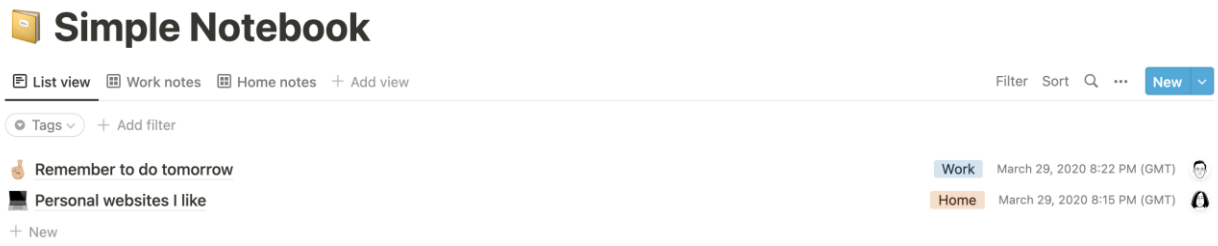


Рисунок 1.2 – Приклад однієї з функцій програми Notion що дає можливість призначити іншу людину під конкретну задачу

Системи бувають двох типів: багатокористувацькі та такі, що розраховані на одного користувача. Застосунки, що розраховані на одну людину обмежують її від інших користувачів забираючи певні функціональні особливості хоч і дають більше конфіденційності. В свою чергу, багатокористувацькі системи дають можливість користувачам взаємодіяти між собою у межах можливостей застосунку. Взаємодіяти, означає додаватись у друзі, ділитись інформацією а також разом маніпулювати даними що використовують. Системи з кількістю користувачів від одного і більше дають можливість розвивати цілі платформи для взаємодії. Підводячи підсумки, робимо висновок, що багатокористувацькі системи більше підходять для сучасних систем ведення словників.

Отже, збираючи до купи усе сказане вище, можна сказати що системи ведення словників глибоко укорінились як основи для більшості застосунків

призначених для роботи зі значною кількістю даних що слід якимось чином обробляти та структурувати. Такі системи мають чітко сформовані напрямки розвитку, а також є актуальними і будуть актуальними у майбутньому.

1.2 Розгляд наявних рішень та порівняння характеристик

Існує чимала кількість програм схожих за функціональністю з тією, що була розроблена під час виконання даної роботи. Проглянувши низку відомих програм зі схожим функціоналом, а також порівнявши деякі з них за схожістю, було обрано декілька варіантів.

1.2.1 Reword

Reword – це зручний застосунок у якому є можливість додавати слова у словник і вчити їх завдяки алгоритму розумного навчання, а саме методом інтервального повторення [1], коли слова видаються періодично (рис. 1.3). Саме завдяки такому підходу, мозок людини поступово тренується та запам’ятовує іншомовні слова.

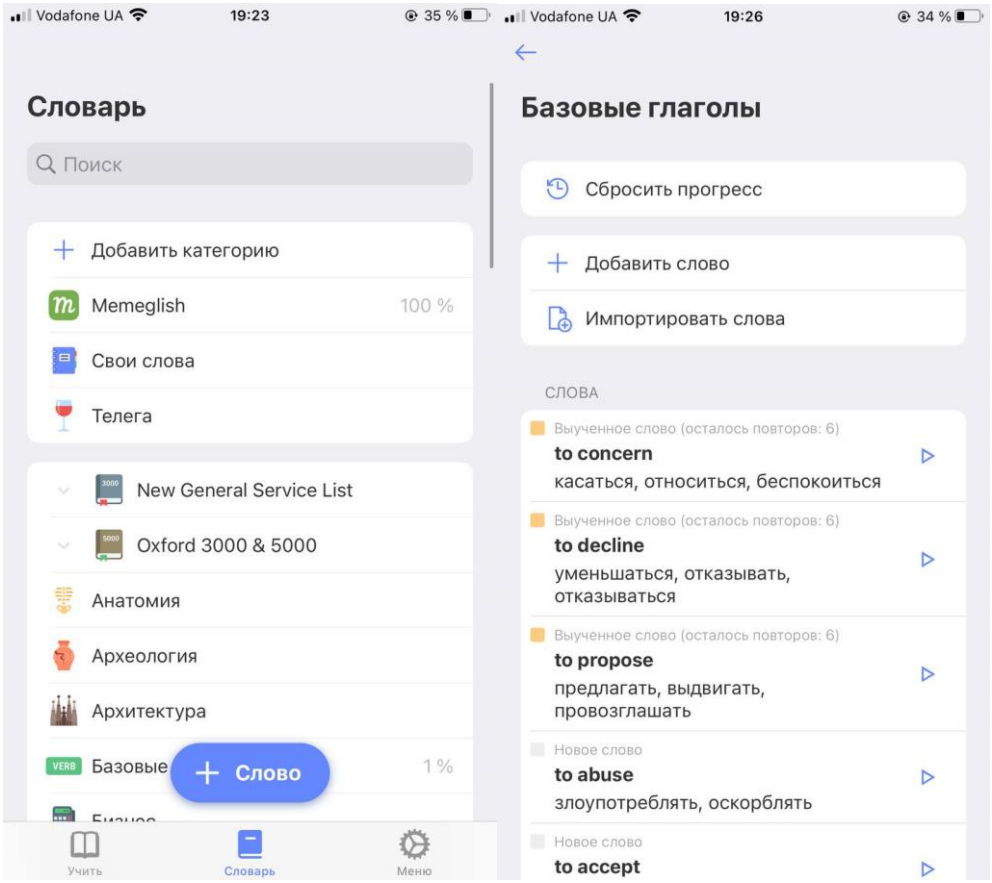


Рисунок 1.3 – Інтерфейс мобільного застосунку для вивчення іншомовних слів у виді словників Reword на IOS

Однією з важливих особливостей є те, що застосунок не дає можливості додавати друзів. Оскільки програма розрахована на персональний розвиток, розробниками даного продукту було прийнято рішення, що функція додавання у друзі не несе необхідності, хоча саме ця функція могла би бути однією з сильних сторін даної програми. Люди що користуються тривалий час застосунком назбирують чимало важливих іншомовних термінів, слів та словосполучень. Таких людей чимало. Досить зручно коли новачок, що зайшов у програму перший раз, за порадою друга, отримує від нього ж цілий словник з важливими та часто використовуваними іншомовними словами. Ця можливість дає користувачам можливість ділитись досвідом. Викладачі можуть давати учням словники для вивчення на наступну контрольні заняття, студентам список термінів для вивчення на семестр і так далі.

Щодо сильних сторін цього застосунку. Однією з найважливіших можна вважити поділ усіх іншомовних слів на категорії. Окрім того що після першого візиту до програми користувач отримує цілий ряд стандартних словників з значною кількістю слів та перекладів, також є можливість додавати власні словники та вчитись по ним також. Користувач отримує налагоджену систему для вивчення слів у вигляді карток з іншомовними словами, а також можливістю переглянути переклад в один клік.

Провівши аналіз використаних технологій для вивчення слів, було виявлено що метод, який використовуються у даному застосунку є одним з найефективніших. Цей метод полягаю у повторенні інформації через певний проміжок часу. У даному застосунку це повторення через короткий проміжок часу на початку вивчення або у випадках коли користувач відповідає неправильно і повторення через довгий при правильних відповідях.

1.2.2 Cambridge Dictionary

Cambridge Dictionary – чи не найвідоміший словник іншомовних слів та виразів у світі (рис. 1.4). Основна ідея вебзастосунку полягає у зручному пошуку необхідних слів та перекладів для кращого розуміння їх у контексті. Зручність полягає у простоті. Є поділ на категорії а саме слова, словосполучення, вирази і тому подібне. Cambridge Dictionary яскравий представник звичайного словника з яким зручно працювати. Для порівняння був обраний оскільки має у собі усе, що потрібно для хорошого словника, але без ускладнень. Він спрямований на те, що користувач заходитиме туди лише для того, щоб розібратися з правильністю вживання тих чи інших фраз та словосполучень. Саме тому відсутні такі елементи як функція, що дозволяє ділитись словами, а також вбудований чат для обговорення з використанням іншомовних слів. Зазначена програма подана конкретно як джерело інформації і з цією функцією справляється краще ніж відомі аналоги.

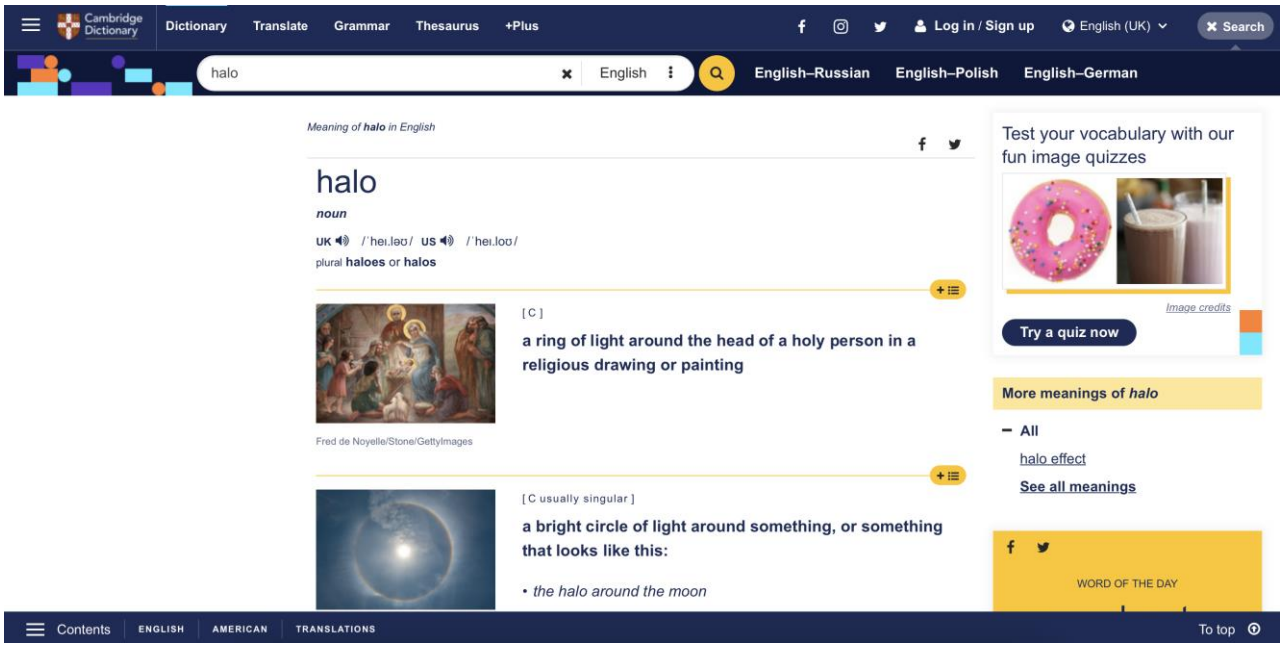


Рисунок 1.4 – Застосунок Cambridge Dictionary.

1.2.3 Vodiy.ua

Vodiy.ua – це українська програма для вивчення правил дорожнього руху (рис. 1.5). Програма досить відома, потрібна, має безліч аналогів по всьому світу. Сформована на основі правил дорожнього руху, що пишуться і упорядковуються безпосередньо законодавством. Досить зручна для порівняння програма оскільки має зрозумілу, просту, не перевантажену структуру і спрямована для вивчення конкретної теми.

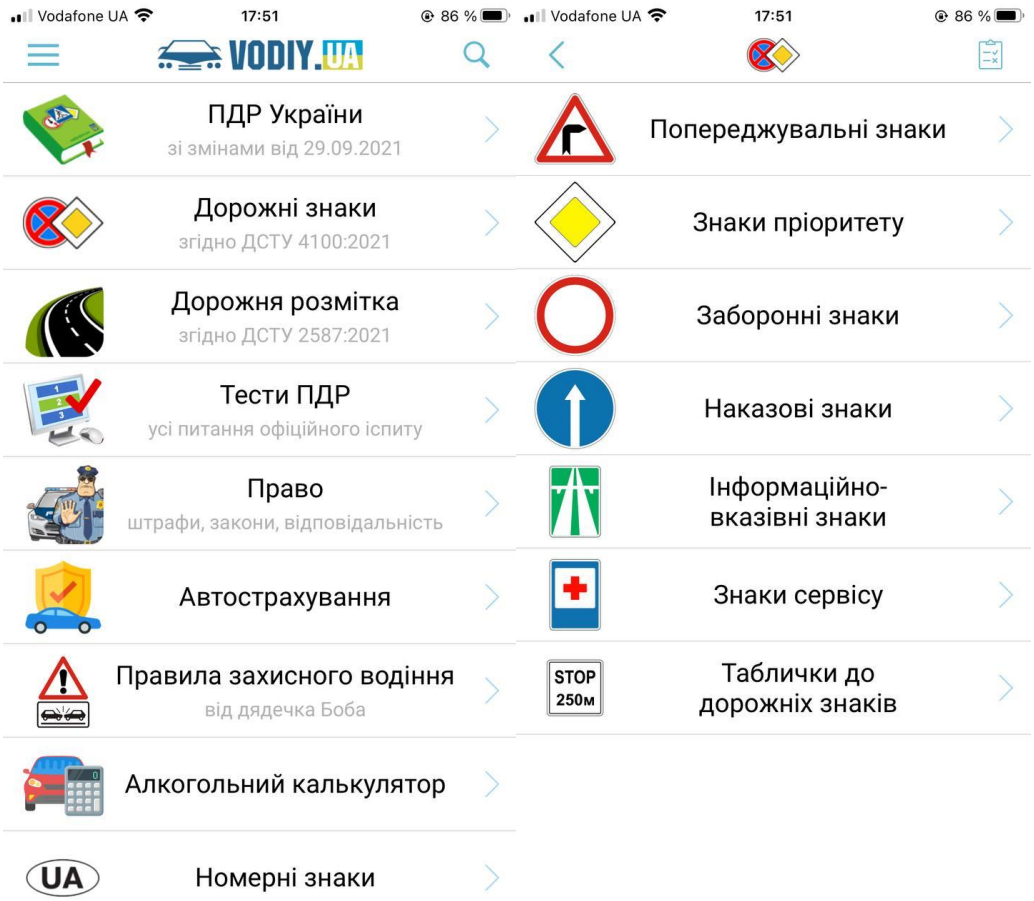


Рисунок 1.5 – Інтерфейс програми Vodiy.ua

Цій програмі, як і багатьом іншим, не вистачає функцій що дадуть можливість взаємодіяти з іншими користувачами. Саме ця вузька спеціалізація поширює думку, що програма є звичайним довідником. Але функції, що дають можливість перевіряти знання отримані від вивчення матеріалів, компенсують ці думки. Якщо з'явиться можливість паралельно виписувати важливе під час вивчення, а потім ділитися у самому застосунку, або ж формувати поступово свої тести і кидати друзям для перевірки власних знань, то програма має почати набирати більшої популярності і серед інших груп людей.

Також, не можна не згадати про чітке розмежування і поділ на категорії усіх видів інформації у даному застосунку. Це є невід'ємною складовою хорошого словника і зазначена програма є такою.

1.2.4 Notion

Notion – одна з відомих програм для записів, ведення словників, ведення бізнесу, планування і так далі (рис. 1.6). Її часто використовують як власне портфоліо, як сховище фотографій, або ж як щоденник. Програма поєднує у собі безліч важливих функцій що робить її універсальною але в свою чергу досить вибагливою до ресурсів програмою. Notion можна віднести до категорії систем ведення словників оскільки тут наявні усі необхідні деталі для цього.

Notion – застосунок з влаштованими додатковими функціями за оплату, що надає функціонал для ведення словників разом з іншими користувачами. Ця функція сильно відрізняє дану систему від ряду інших. Ця функція розглядається як вагома перевага даного продукту оскільки інші програми словники зазвичай нехтують такою особливістю.

Важливим недоліком є саме перевантаженість, оскільки коли говорять про словники, то мають на увазі що такі програми швидкі та доступні. Програми-записники розраховані на те, що користувачі будуть раз по раз відкривати програму щоб записати щось або, як у випадку з словниками-довідниками, відкривати програму щоб щось дізнатись по конкретній темі. Notion змусить чекати після кожного запуску. Також важно сказати що Notion належить конкретно до категорії словників, оскільки має в собі функціональності для низки інших категорій.

Що стосується дизайну, Notion виглядає сучасно. Це і не дивно, оскільки цей проект є комерційним і підтримується командою досвідчених розробників з усього світу.

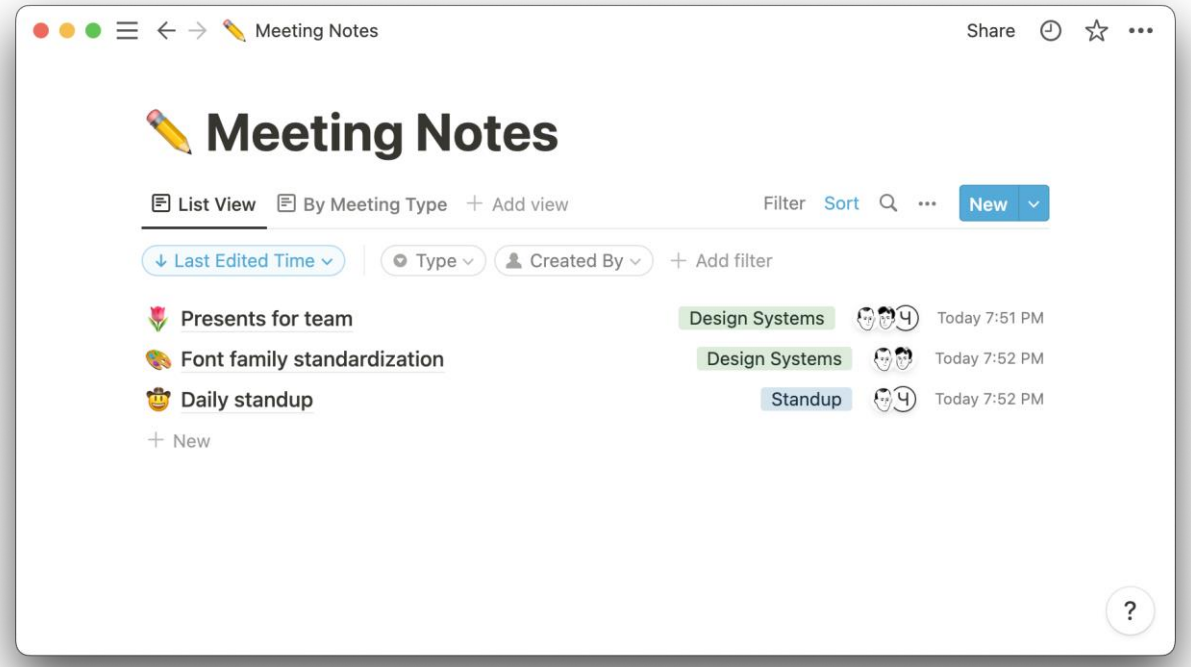


Рисунок 1.6 – Приклад роботи програми Notion

1.3 Порівняння розглянутих систем ведення словників

Отже, в результаті порівняння деяких відомих застосунків схожих за функціоналом з багатокористувацькою системою ведення словників, було сформовано певні висновки.

Перше, і одне із найважливіших, це те що в усіх перелічених програм відсутня функція взаємодії з іншими користувачами. Це означає що вони розраховані на користувачів які будуть мати можливість взаємодіяти з даними у програмі але не матимуть можливість взаємодіяти з іншими користувачами задля взаємної вигоди. Як приклад, словники, що були створені у програмі Reword не можуть бути надані іншим користувачам друзям оскільки це власні записи.

Дві з чотирьох програм мають інформаційну базу що підтримується та доповнюється. Це обґрунтовано тим, що ці програми використовуються для ознайомлення, перегляду або ж вивчення інформації і є джерелами. Дві інших

програми розраховані на те що люди будуть записувати туди власні дані і потім зможуть маніпулювати ними як їм заманеться. Це лише питання спеціалізації програми.

І останній підпункт який лише підтверджує що дані програми використовують систему словників, а також мають за мету відрегулювати та поділити дані по конкретним ознакам. Це функція поділу даних на категорії. Завдяки цьому користувач може без перешкод здійснювати пошук необхідної інформації. Корисно для тих кому потрібно швидко знайти інформацію або тим, хто має маніпулювати великою кількістю інформації.

Таблиця 1.1 Порівняння програм та наявних рішень

№	Назва	Можливість взаємодії з іншими користувачами	Функція поділу даних на категорії	Має інформаційну базу що підтримується та доповнюється
1	Reword	Відсутня	Присутня	Ні
2	Cambridge Dictionary	Відсутня	Присутня	Так
3	Vodiy.ua	Відсутня	Присутня	Так
4	Notion	Присутня	Присутня	Ні
5	Даний Застосунок	Присутня	Присутня	Так

ВИСНОВОК ДО РОЗДІЛУ 1

Отже, було проаналізовано деякі відомі застосунки що використовують так чи інакше систему словників. Відбулося їх порівняння, виявлено слабкі та сильні сторони, а також було проведено короткий аналіз наявних систем та їх актуальності і було сформовано деякі висновки, а саме:

- Система словників є однією із найпопулярніших систем для застосунків що призначені для навчання та менеджменту. Популярність визнана тим, що такі програми структурують дані користувача а також дають багато нових функціональних можливостей.
- Системи словників можуть мати одного або безкінечну кількість користувачів. Система, де користувачів більше ніж один, буде мати назву багатокористувацька система. Цей термін означає, що така система дозволяє користувачам взаємодіяти між собою а також використовувати функціонал програми у своїх цілях разом з іншими зареєстрованими користувачами.
- Системи словників відрізняються від програм з записами своєю структурованістю. Це означає, що словники дають здатність до легкого пошуку а також розділом інформації на категорії. Також мають чимало шляхів для розвитку і будуть актуальними і в майбутньому.
- Обрані для порівняння програми об’єднує здатність до поділу інформації на категорії. Це означає що всі програми що використовують систему словників, використовують основну особливість даного підходу і тому можуть забезпечити легкий пошук та необхідні функції для обробки інформації.
- Більшість з обраних для порівняння програм не мають функції додавання у друзі. Ті, де ця функція реалізована втрачають у швидкості роботи застосунку.

Отже, виходячи з того що було вказано вище, було вирішено зосередитись на розробці функціональності, якої не вистачає ряду інших подібних програмних продуктів.

РОЗДІЛ 2. ВИБІР І ОБҐРУНТУВАННЯ ЗАСОБІВ РЕАЛІЗАЦІЇ ПРОГРАМИ

2.1 Архітектура програми

Під час розробки вебзастосунків прийнято говорити про різного сорту архітектури проектування та шаблони проектування. Один з найефективніших шаблонів проектування, що використовуються при будівництві архітектури вебзастосунків – це архітектура Модель-Представлення-Контролер або скорочено з англійської – MVC (рис. 2.1) [2]. У даному шаблоні проектування важливо насамперед розуміти які проблеми він вирішує. Головною задачею MVC є відокремлення логіки роботи з базою даних, функцій, що обробляють дані і візуальну частину програми [3]. У застосунку за візуальну частину відповідає клієнтська частина що отримує дані, а от контролери та модель знаходяться на серверній частині. Модель може бути описана по різному, у дипломному проекті це шар доступу до даних де відбуваються дії напряму зв’язані з запитами в базу даних. Це означає що у випадках коли слід замінити деякі елементи або всю базу даних, не потрібно буде переписувати увесь застосунок цілком, а лише частину, тобто шар що називають DAL. Шар контролерів поєднує функції, що описані у Моделі та передає оброблені дані у шар представлення де вони відображаються. У контролері прихована уся логіка програми на серверній частині але без явного звертання до бази даних, оскільки це вже задача моделі.

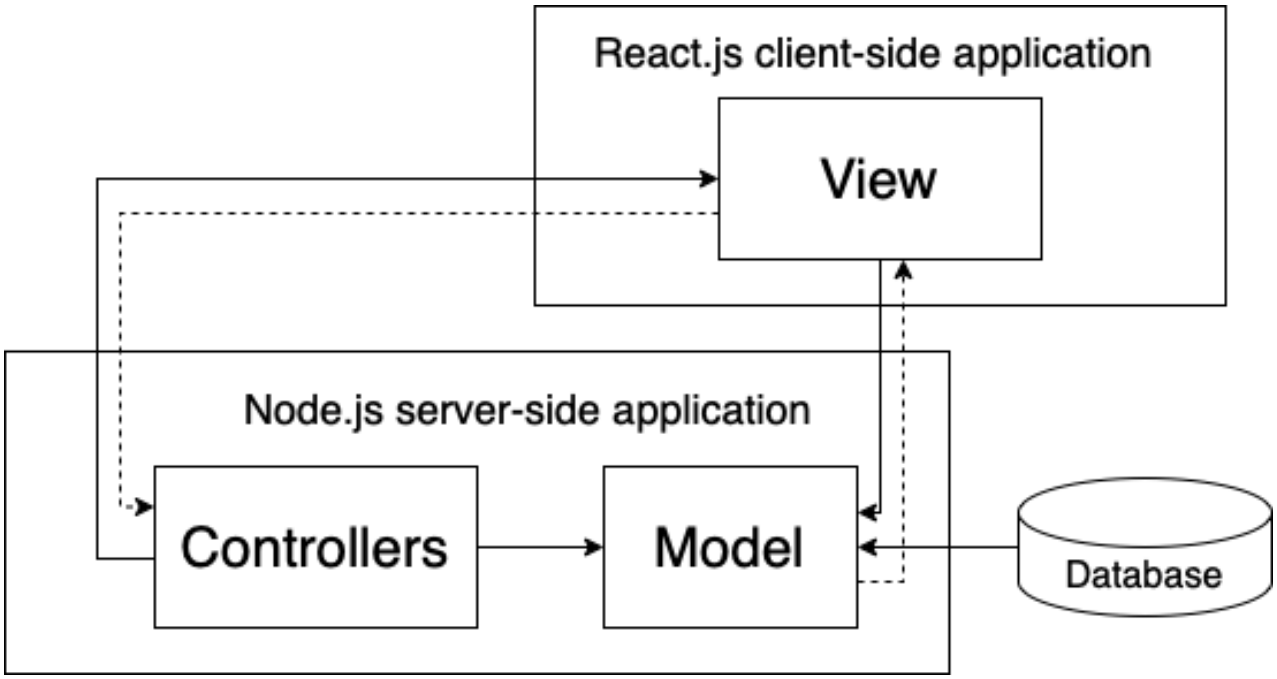


Рисунок 2.1 – Механізм роботи архітектури проектування MVC

2.2 Мова програмування JavaScript

Для написання багатокористувацької системи ведення словників використовувалась мова JavaScript. Це одна з найпопулярніших мов програмування за версією PYPL - рейтингу, створеного на базі кількості пошукових запитів, навчальних програм і матеріалів щодо конкретної мови програмування (рис. 2.2) [4]. Також це мова що займає перше місце у рейтингу від GitHub (платформа, що являється чи не найбільшим сховищем коду з відкритим доступом) уже як понад 6 років [5]. JavaScript все частіше зустрічається у навчальних програмах учнів у школах і є першою мовою програмування для понад як 12% програмістів за результатами незалежного щорічного опитування в Україні.

Worldwide, May 2022 compared to a year ago:

Rank	Change	Language	Share	Trend
1		Python	27.85 %	-2.5 %
2		Java	17.86 %	-0.1 %
3		JavaScript	9.17 %	+0.4 %
4		C#	7.62 %	+0.7 %
5		C/C++	7.0 %	+0.4 %
6		PHP	5.36 %	-1.0 %
7		R	4.34 %	+0.5 %
8	↑↑↑	TypeScript	2.39 %	+0.7 %

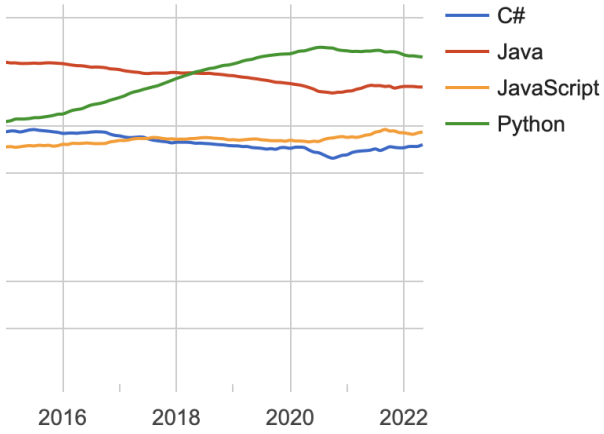


Рисунок 2.2 – Рейтинг популярності мов програмування за версією PYPL станом на травень 2022р.

JavaScript має досить велику низку областей застосування. За останні 7 років, мова зазнала чимало змін [6]. З'явилось чимало нових бібліотек. Зараз можна сказати що вона може використовуватись для написання більшості програм з моменту як так званий 'двигун' v8 перенесли з браузера на сервер і з'явилась бібліотека Node.js. Але якщо підійти до цього питання з точки зору якості результату то слід виділити області де зазначена мова програмування буде більш вживаною. Перше місце, що не дивно, займає веб розробка [7]. JavaScript – це перша мова для розробки вебзастосунків, веб сервісів, сайтів і далі по списку (рис. 2.3). Первинною метою розробників даної мови було створити продукт, що буде здатним виконувати алгоритми що далі будуть називатись скриптами і завдяки цьому на веб сторінках почали б з'являтись нові особливості. З часом JavaScript тільки покращувався. Почали з'являтись бібліотеки, платформи, пакети що тільки покращували можливості даної мови у сфері веб програмування і зазначена мова стала лідером, що, за прогнозами, ще мінімум десятиліття буде тримати дану позицію [8].

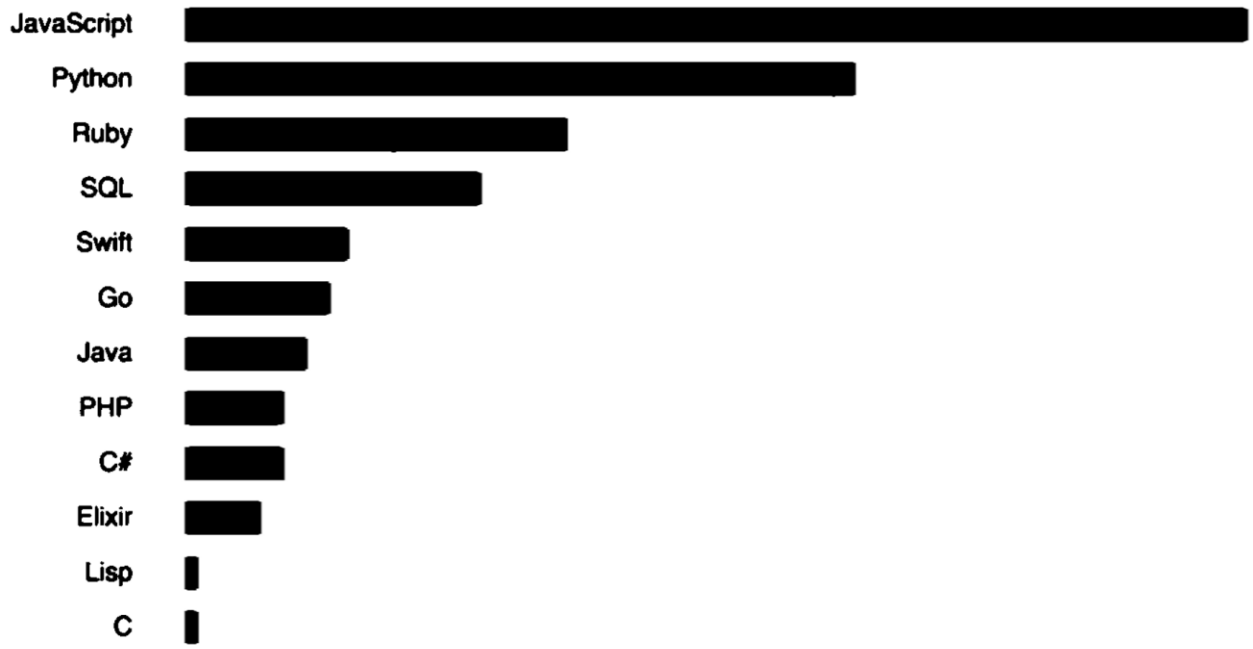


Рисунок 2.3 – Рейтинг популярності мов програмування у сфері веб програмування

Інша, не менш важлива область, це серверна частина застосунків. У наш час стали популярними такі сервіси як: відеохостинги, хмарні сховища, месенджери а також інтернет магазини. Усі вони можуть бути написаними з використанням JavaScript і будуть конкурентоспроможними серед інших мов оскільки, завдяки можливостям фреймворку Node.js, застосунки можуть плавно масштабуватись без явної втрати продуктивності [9].

Також, слід згадати про мобільну розробку та ігрову індустрію де мова JS виступає у ролі альтернативи або ж додаткової для створення конкретних елементів програми. У мобільній розробці значної популярності досягла бібліотека React Native що має за мету об’єднати можливості мобільних програм створених на Objectiv-C та Java, а також облегшити роль розробника і дати можливість писати кросплатформенні застосунки використовуючи лише одну мову JavaScript [10]. Щодо ігор, досить популярними є іо [11] ігри для браузерів що можна грати усім використовуючи для цього лише можливості браузеру.

В інших областях застосування JS має скоріше маргінальне значення ніж впевнене місце у рейтингу.

Щодо особливостей мови, хотілося би зазначити, що JS має нестрогу типізацію [12] що в свою чергу може привести до появи помилок при обробці конкретних даних під час виконання програмного коду. Але такий підхід значно полегшує процес розробки тому не можна явно вказати є це недоліком чи перевагою. У всякому разі завжди може бути використана мова TypeScript що є видозміною JS [13]. Там присутня строга типізація, що покращить надійність коду та вплине на правильність обробки даних [14].

Проте крім вищевказаних переваг, знайдуться і явні недоліки JavaScript. Один з небагатьох це компіляція коду під час його виконання [15]. При масштабуванні кількості коду програми буде відповідно збільшуватись і час виконання програми.

2.3 Бібліотека Node.js

Node.js це одна з найбільш популярних технологій у світі за версією щорічного опитування на платформі Stackoverflow де взяли участь 83052 учасника з усього світу з яких 58 031 це професійні розробники з досвідом (рис. 2.4) [16]. Зазначена технологія це не просто рушій v8 перенесений з браузера Chrome [17], це ціла платформа що призначена для того, щоб виконувати програмний код вебзастосунків написаних на JavaScrypt чи TypeScript. Використовує асинхронну модель виконання з блокуючим вводом та виводом [18].

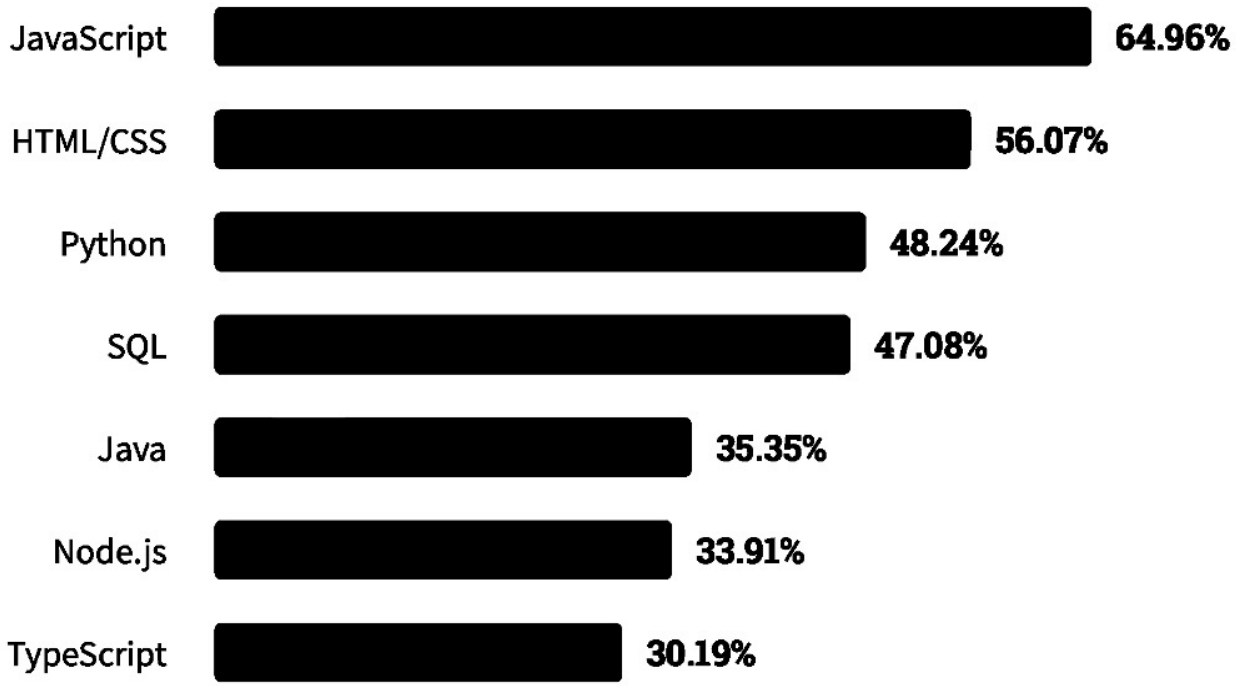


Рисунок 2.4 – Рейтинг найбільш популярних технологій у світі станом на 4 квартал 2021 року за версією порталу Stackoverflow Survey

Також варто описати бібліотеку, що нерідко стає невід'ємною частиною серверу написаного з застосуванням технології Node, а саме Express.js [19]. Головна мета Express це надати функціонал для обробки HTTP-запиту після відправки і до отримання. Завдяки функціям посередникам можна змінювати, видаляти або додавати дані що передаються у запиті і таким чином реалізовувати складні системи як авторизація, перевірки токенів і тому подібні.

Пару слів про роботу Node.js зсередини. Зазначена програмна платформа працює з неблокованим введенням та виводом. Це означає, що при виконанні операцій програма не буде блокуватись і чекати завершення поставленої задачі а буде продовжувати працювати паралельно чекаючи відповіді про завершення тієї задачі. Завдяки такому підходу Node.js стала однією з кращих платформ у швидкості виконання програмного коду серед платформ що застосовуються для програмування веб сервісів [20] оскільки код виконується асинхронно через цикл подій (рис. 2.5) [21].

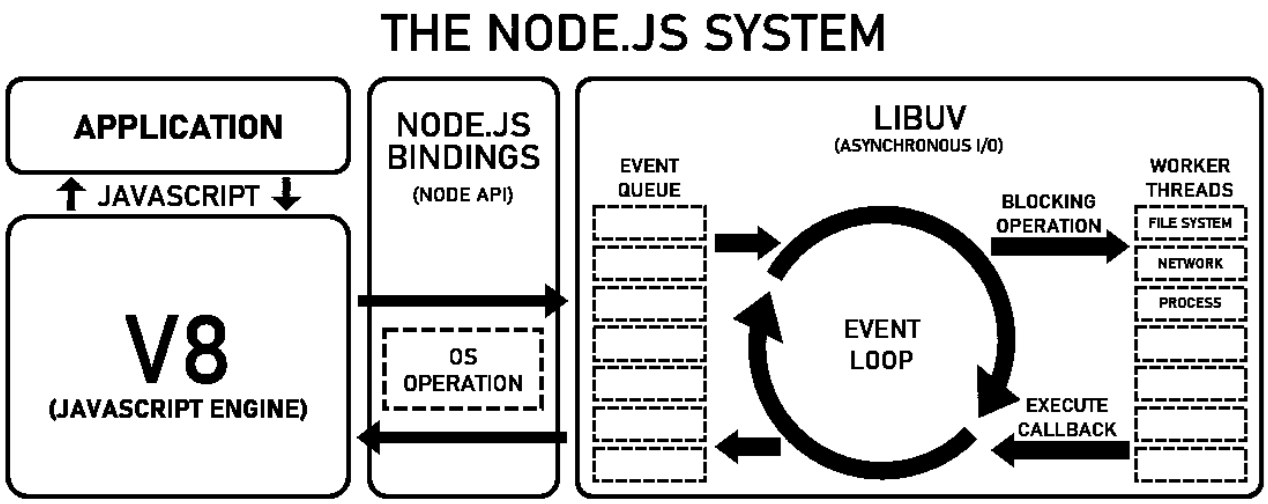


Рисунок 2.5 – Механізм циклу подій у середовищі Node.js

2.4 Бібліотека React.js.

Для розробки клієнтської частини, проаналізувавши популярні варіанти, а також зваживши переваги та недоліки конкурентів було обрано, чи не найкращу у даному сегменті бібліотеку React.js. Наразі це найпопулярніша технологій у категорії ‘Web Frameworks’ (рис. 2.6) [22]. Про це свідчать купа опитувань, що проводяться як звичайними спільнотами програмістів так і відомими платформами як наприклад Stack overflow де взяли участь 67593 учасника з усього світу.

React це відкрита для вільного доступу бібліотека, код якої може бути вдосконаленим третіми зацікавленими особами. Розробкою займається компанія Meta засновником якої є Марк Цукерберг. Бібліотека призначена для розробки інтерфейсів користувача.

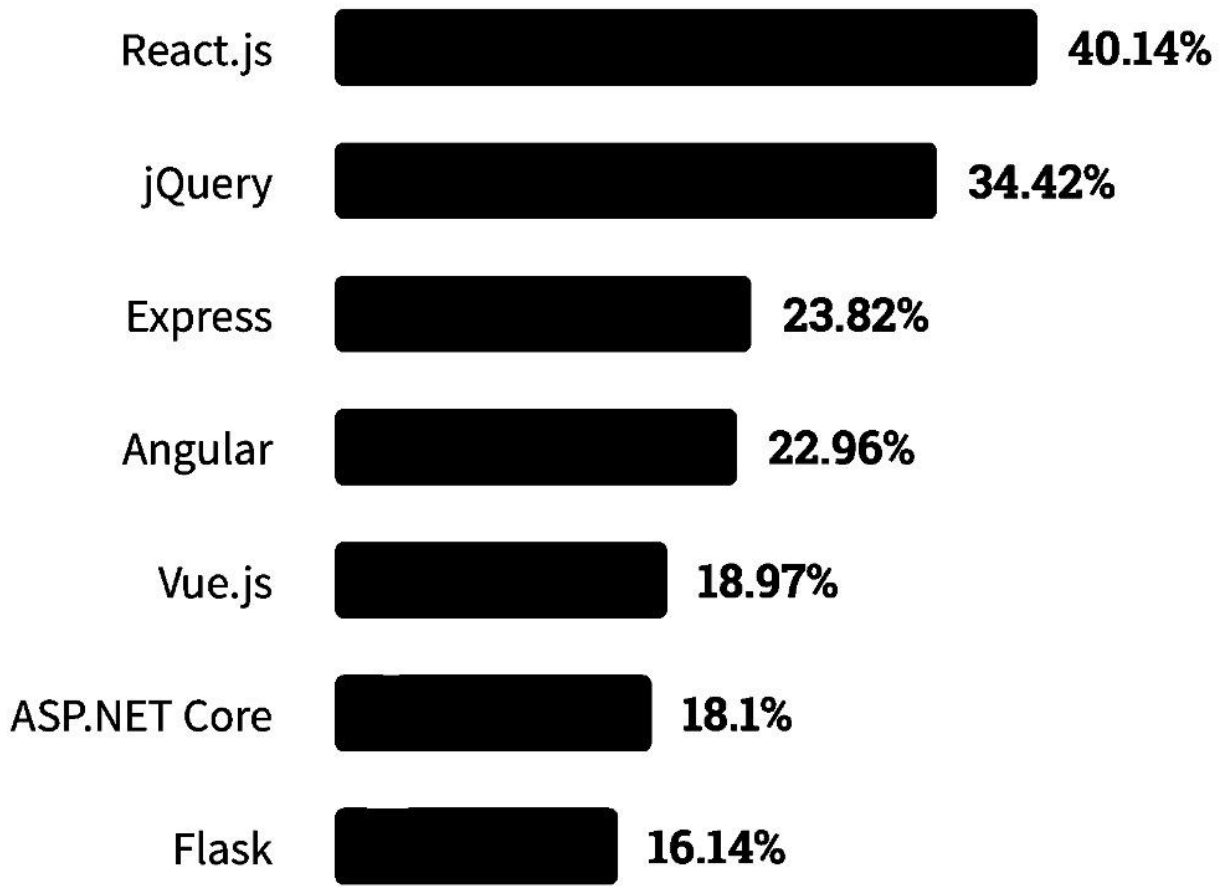


Рисунок 2.6 – Результат опитування на платформі stackoverflow де прийняли участь 67593 учасників

React працює на основі так званого життєвого циклу (рис. 2.7) [23]. Завдяки цій технології програма може обробляти дані від користувача у різних точках даного циклу, чи то оновлення даних для виводу чи безпосередньо їх відображення. Цей підхід дозволяє застосункам, створеним на React працювати швидше ніж аналоги [24]. Також у бібліотеці використовується JSX [25], що дозволяє поєднувати HTML елементи з програмним кодом клієнтської частини і напряму будувати між ними зв’язки. Також слід зазначити, що React відповідає обраному нами шаблону проектування під назвою MVC та виконує у ньому роль представлення, що дозволить без зайвих проблем зкомпонувати клієнтську та серверну частини системи.

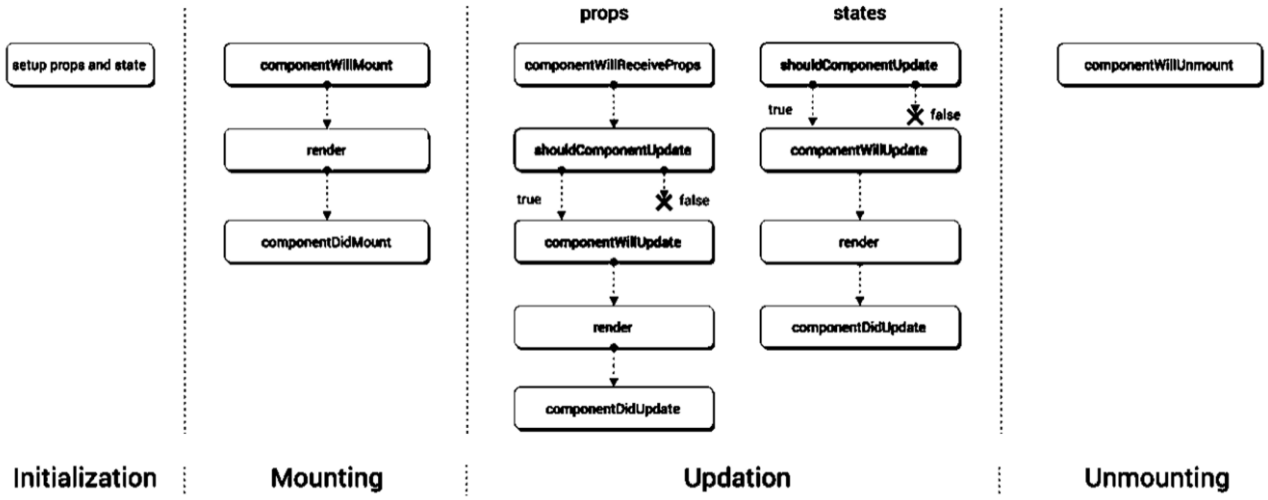


Рисунок 2.7 – Схема життєвого циклу простого застосунку, що написаний з використанням технології React

Важливою складовою клієнтської частини застосунку є робота з серверним API (рис. 2.8). За це відповідає пакет Axios. Він дозволяє робити HTTP запити з користувацькими даними до back-end частини застосунку, а звідти отримувати результат обробки цих даних програмою [26]. Цей пакет являє собою так званий зв’язок між двома основними складовими усього сервісу і дозволяє поєднати їх завдяки методології запит-відповідь.

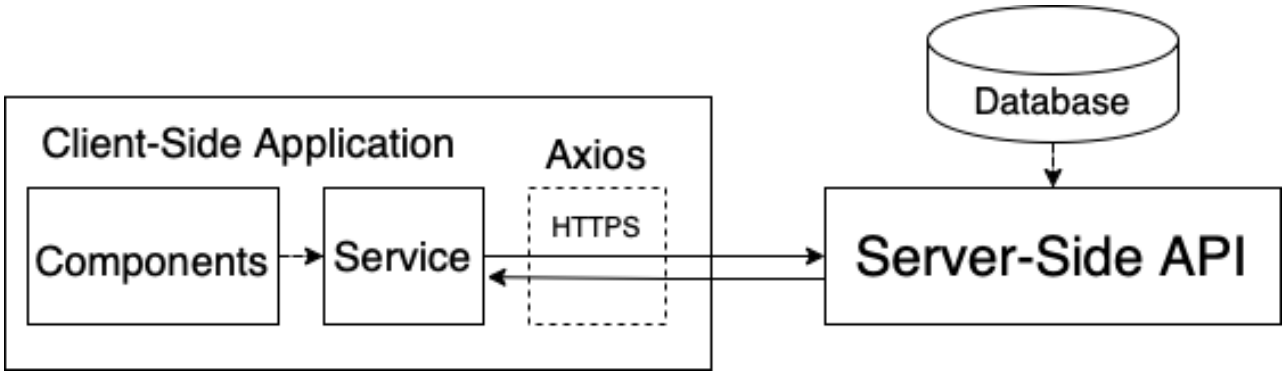


Рисунок 2.8 – Схема роботи з серверними API за допомогою HTTP клієнту Axios

2.5 База даних PostgreSQL

Почнемо з того, що PostgreSQL одна з найпопулярніших реляційних баз даних у сучасному IT [27]. Значну популярній, у значній мірі, він отримав через відкритий програмний код що удосконалювався ще з 1996 року і по сьогодення. У наш час, головними перевагами цієї бази даних є легка масштабованість та висока продуктивність. PostgreSQL наразі використовують такі компанії як Instagram, Skype і подібні.

У контексті даної роботи, PostgreSQL використовується як основна і єдина база даних що дає можливість зберігати усі дані користувачів починаючи від особистих електронних поштових скриньок, паролів та токенів авторизації і закінчуючи таким налаштуваннями як змінене ім’я або картинка профілю. Також завдяки можливостям ORM Sequelize стало можливим додавати нові таблиці, залежності, масштабувати базу даних, а отже здійснювати повне управляння даними без поглиблених знань у сфері проектування баз даних а також знань щодо правильності та надійності підключення до серверної частини. Основна мета ORM Sequelize це організувати взаємозв’язок між серверною частиною застосунку та базою даних без використання конкретних запитів до SQL (рис. 2.9).

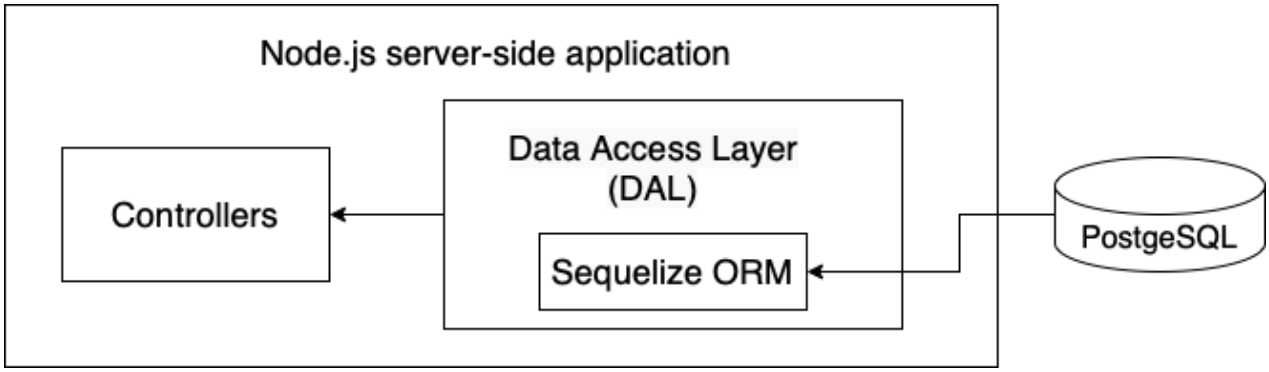


Рисунок 2.9 – Схема взаємодії серверної частини застосунку Node.js з реляційною базою даних PostgreSQL за допомогою ORM Sequelize

2.6 Середовище розробки WebStorm

Для реалізації ідеї створення програми ‘Багатокористувацька система ведення словників’ було обрано чи не найпотужніше серед аналогів середовище для розробки під назвою WebStorm від компанії JetBrains. Головними причинами стали:

- можливість налаштувати IDE та додавати необхідні модулі
- можливість взаємодіяти з базою даних та системою контролю версій Git з вікна для розробки.

Ці можливості значно покращують процес розробки оскільки необхідні дані знаходяться у швидкому доступі на екрані і не потребують зайвих дій, що відволікають, для їх підключення чи якісного використання.

Детальніше про налаштування. WebStorm дає можливість підключатись до будь-якої з баз даних завдяки влаштованим інструментам для роботи з ними (рис. 2.10). Важливо що, база даних не обов’язково має бути на комп’ютері де знаходиться програма. У WebStorm можна підключатися до віддалених баз даних і користуватися ними у повній мірі як і з тими що підключені локально. При написанні даної роботи не раз використовувався цей метод підключення що дало можливість зменшити навантаження на робочий процесор перенісши базу даних на віддалений комп’ютер. Також слід зазначити про можливості самого вікна Database де користувач має повний контроль над базою даних (у дипломному проекті це PostgreSQL). Розробник може додавати, редагувати, видаляти, робити пошук, змінювати структуру, вигляд і багато іншого через інтерфейс програми. Завдяки полю для вводу SQL запитів можна компенсувати елементи роботи з базою даних, яких не вистачає.

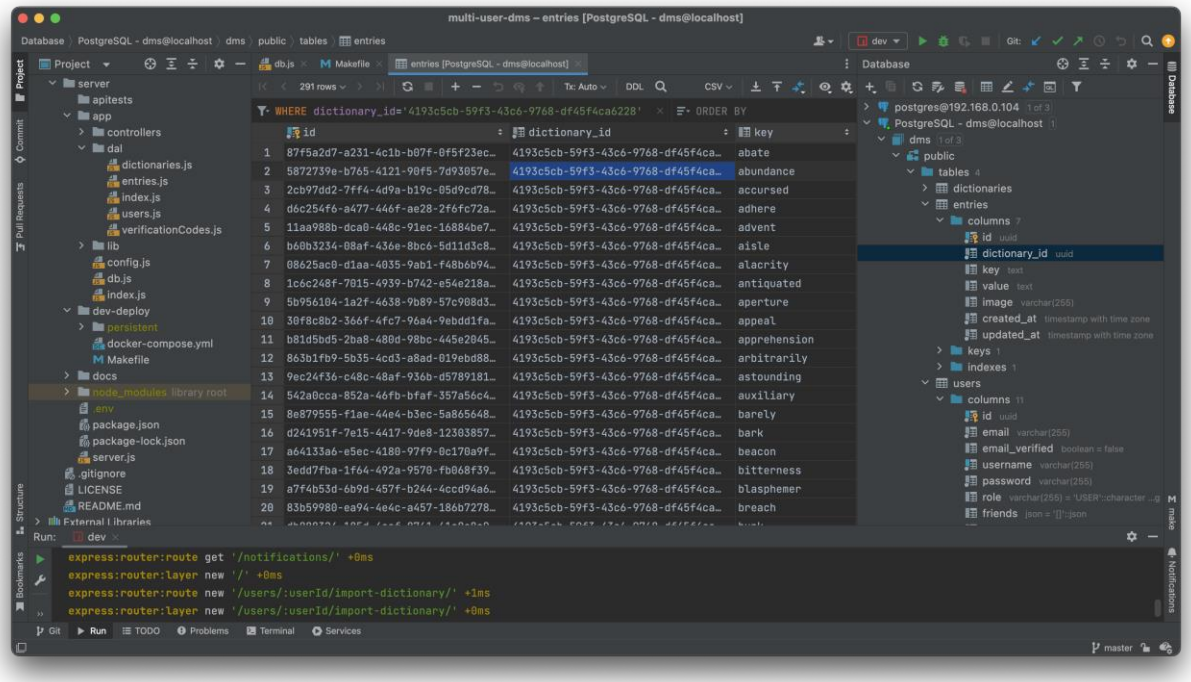


Рисунок 2.10 – Інтерфейс роботи з базами даних у WebStorm

Наступною, мабуть найбільш використовуваною функцією цього IDE є вбудована система контролю версій Git (рис. 2.11). WebStorm. Якщо розробник має підключення до системи Git, то IDE буде постійно показувати інформацію про зміни, що були зроблені після останнього запиту ‘git push’. У випадку якщо зміни були зроблені випадково, є можливість вернути їх у минуле значення. У швидкому доступі кнопки для команди git commit, merge та push. Усі інші часто вживані команди є у випадаючому вікні. Завдяки цьому, розробник не відволікається кожен раз на команди Git у терміналі та може більше зосередитись на програмі що розробляє.

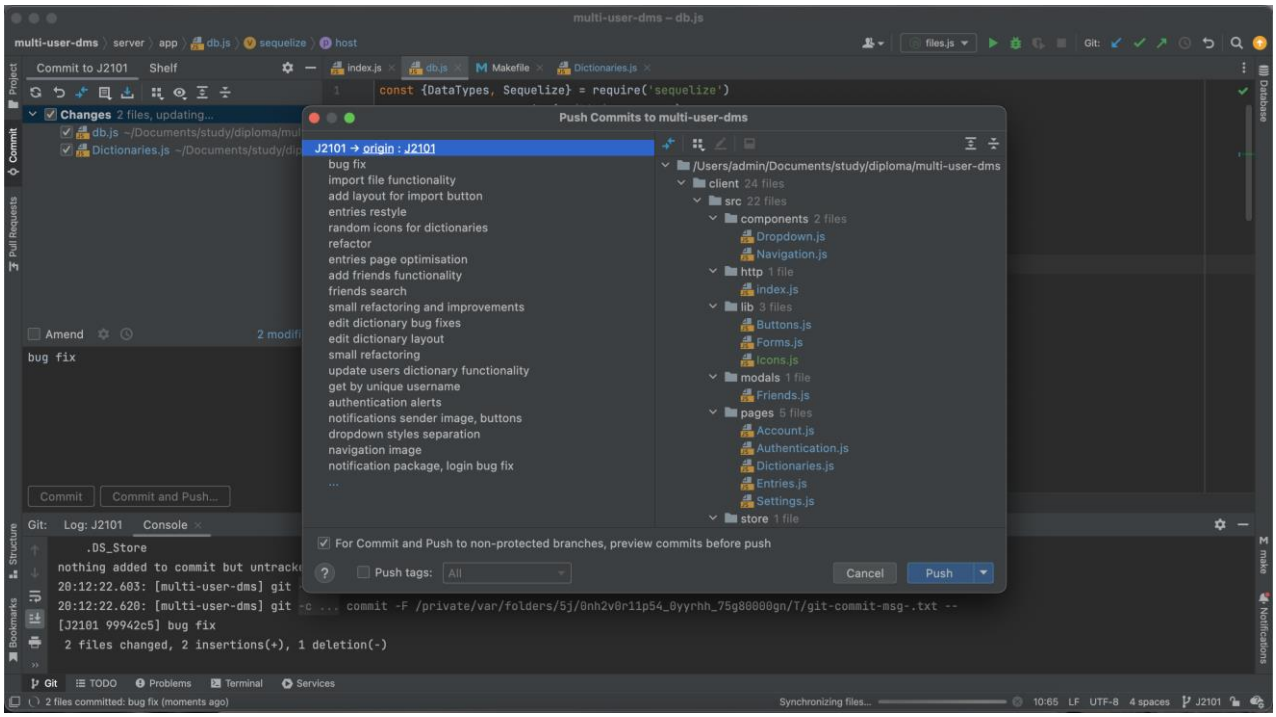


Рисунок 2.11 – Вбудований інтерфейс роботи з системою контролю версій Git у середовищі для розробки WebStorm

Третє, що варто зазначити, це вбудовані механіки для відслідковування помилок прямо під час розробки програмного коду. WebStorm – це програма що виконує компіляцію деяких частин програми прямо під час написання, що дозволяє відслідковувати можливі помилки ще на ранньому етапі розробки. Це значно спрощує роботу над застосунком. Є можливість ‘поглиблюватись’ у функції експортовані з інших файлів а також, використовуючи налагоджувач коду, проходити всі етапи виконання коду з детальною інформацією про дані що зчитуються та записуються. Також легко робити перевірку коду та виправляти помилки слідуючи підказкам в IDE. Ще WebStorm має вбудований інтерфейс конфігурацій запуску (рис. 2.12)

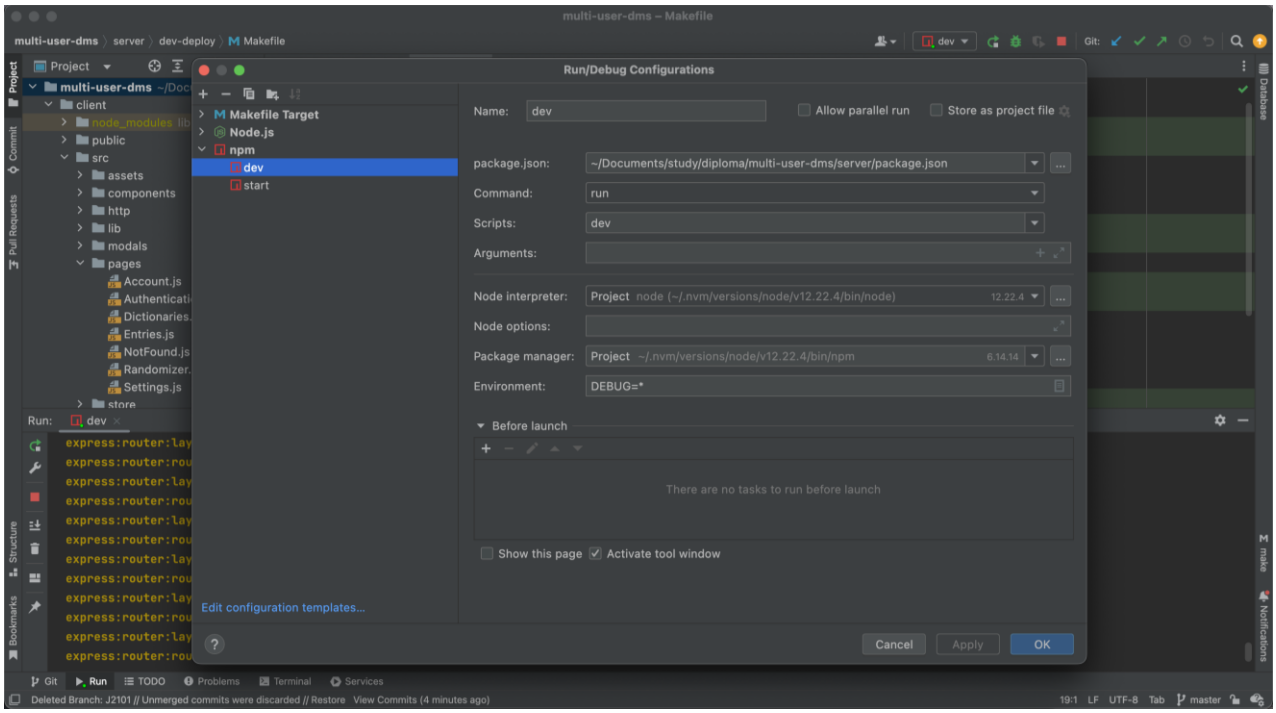


Рисунок 2.12 – Вбудований інтерфейс конфігурацій запуску у середовищі для розробки WebStorm

Не беручи до уваги усі описані вище можливості та функціональності, цей IDE дає можливість встановлювати плагіни для посилення уже наявних можливостей чи додавання нових. А також має влаштований HTTP Client що може замінити на ранніх етапах такий інструмент як Postman, що призначений для відсилання HTTP запитів на клієнтську частину додатку. Чимало важливих функцій, як наприклад ‘Code with me’ чи вікно для перегляду журналів виконання коду з серверу не було описано оскільки в даній роботі не використовувалися, але вони однозначно можуть бути корисними у інших проектах.

ВИСНОВОК ДО РОЗДІЛУ 2

Отже, в результаті детального огляду архітектури та усіх можливих засобів реалізації багатокористувацької системи ведення словників було вирішено, що найкращими для виконання поставленої задачі інструментами та бібліотеками є наступні, зазначені нижче.

Мова програмування JavaScript. Проаналізувавши документацію та програмний код інших додатків, було з'ясовано що зазначена мова має усі необхідні для розробки проекту інструменти а також чи не найбільшу у світі спільноту розробників. Ці фактори є ключовими і впливають насамперед на якість та наповненість даної роботи.

Для розробки серверної частини вебзастосунку вирішено використовувати Node.js. Бібліотека популярна серед відомих компаній IT та підтримується світовою спільнотою. Серверна частина з використанням Node.js являє собою надійну опору для клієнтської частини і гарантує швидкодію та надійність обробки даних та постійний доступу до бази даних.

Клієнтська частина розроблена з використанням бібліотеки від компанії Meta – React.js. Бібліотека призначена для будування користувацького інтерфейсу та взаємодії з серверною частиною методом запитів до API. Серед усіх проаналізованих аналогів React.js привертає увагу своєю швидкодією та багатим інструментарієм для розробки та підтримки проекту. JSX дає можливість поєднувати програмний код з елементами HTML та CSS.

База даних PostgreSQL була обрана виключно через свою обширну документацію, високу продуктивність та легку масштабованість з використанням інструментів СУБД Sequelize.

З міркувань швидкості та безперервності розробки системи було вирішено використовувати один з найбільш функціональних IDE WebStorm. Провівши аналіз можливостей даного середовища було з'ясовано, що там є усі необхідні ресурси для розробки на усіх етапах проектування. З головних, це інструментарії для роботи з Git і PostgeSQL, а також плагіни й конфігурації.

РОЗДІЛ 3. Розробка багатокористувацької системи ведення словників

3.1 Сценарії взаємодії користувача із системою

Було сформовано основані сценарії роботи користувача із застосунком:

- Автентифікація користувача у системі
- Створення, редагування, видалення словника
- Створення та видалення записів у словнику
- Поширення словника друзям та відправка сповіщення
- Імпортування записів з файлу
- Налаштування профілю та акаунту користувача
- Пошук друзів

3.1.1 Автентифікація користувача у системі

Неавторизований користувач не може користуватись функціоналом застосунку, тому він має здійснити автентифікацію. Автентифікацією називають процес, під час якого людина вводить дані, такі як пошта та пароль для входу під своїм обліковим записом. Ці дані використовуються для надання користувачу його особистих даних, таких як словники і записи у них, а також налаштувань.

Коротко про реєстрацію та вхід у систему. Користувач що не має облікового запису в системі повинен зареєструвати новий акаунт. Для цього він вводить пошту та пароль. Після відправки запиту про реєстрацію на сервер, відбуваються перевірка на наявність акаунту з заданою поштою і якщо таких не знаходиться, то створюється новий користувач у базі даних. Далі можна користуватись акаунтом вводячи при вході ті ж самі дані, що були використані під час реєстрації.

У застосунку використовуються технологія JWT [28] для перевірки автентифікації. Це працює так, що користувач, після введення пошти та паролю, отримує від серверу токен у якому зашифрована певна інформація. Цей токен зберігається десь у сховищі в браузері і під час кожного наступного запиту додається до запиту. Сервер отримує цей запит, перевіряє валідність токена та відправляє результат у відповідь. Якщо все виконано правильно, то в результаті користувач отримає очікувані дані від серверу. Важлива особливість JWT токену це те, що він перестає бути дійсним через конкретно-вказаний проміжок часу. Це є одним з елементів захисту від зловмисників, оскільки у випадку, якщо хтось вкраде токен і почне використовувати у своїх цілях, то він не зможе робити це вічно.

3.1.2 Створення, редагування, видалення словника

Оскільки основною ідеєю розроблюваного застосунку є робота з словниками, тому частина що відповідає за їх обробку є найзмістовнішою. Створення словників відбувається шляхом відправки запиту на сервер з усіма необхідними даними. У свою чергу користувацький інтерфейс оновлюється і відображає дані що прийшли у відповідь на запит. Новий словник при створенні, не враховуючи імпортування, містить у собі нуль записів. Для створення нових записів потрібно перейти на сторінку словника та додати їх. У цей момент відбудеться запит на створення нового запису з прив’язкою до обраного словника, тобто у таблиці ‘entries’ в базі даних буде створено запис де ідентифікатор словника буде рівний тому, що обрано при створенні. Під час редагування словника змінюється відображуваний елемент з назвою словника на екрані на поле вводу і після підтвердження змін відбувається запит на оновлення. Під час видалення, відповідно, відбуваються запит на видалення запису з бази даних на сервері і відправляється результат дії. У випадку, якщо запис не видалився, з’являється повідомлення про помилку.

3.1.3 Поширення словника друзям та відправка сповіщення

Один з важливих сценаріїв, це відправка словників друзям. Після того як користувач обирає у словнику опію поділитись та обирає іншого користувача, відправляєтеся запит користувачу з повідомленням та даними словника для його копіювання. В свою чергу, усі інші користувачі постійно відслідковують подію відправки повідомлення з серверу. Якщо така трапляється то за ідентифікатором користувача відбувається відображення повідомлення тому, кому воно було відправлене. Якщо користувач приймає запит, відправляється запит на сервер з даними словника що необхідно скопіювати. Відбувається алгоритм за яким дані копіюються та присвоюються іншому користувачу а повідомлення зникає.

3.1.4 Імпортування записів з файлу

Функція імпортування є інтеграцією з іншими додатками словниками, оскільки дозволяє додати цілий список нових записів у словник з файлу що сформований іншими додатками. Після обрання файлу, відбувається запит на сервер під час якого файл зчитується. Дані з файлу фільтруються та зберігаються у список. Далі для кожного елементу списку створюється окремий запис у базу даних для нового створеного словника. В результаті користувач спостерігає створення словника що має назву ідентичну з назвою файлу.

3.1.5 Налаштування профілю та акаунту користувача

Сторінка налаштувань має низку функцій що дають можливість змінювати дані користувача. Головним сценарієм є зміна картинки користувача та псевдоніма. Під час зміни картинки, після обрання нового файлу з файлової системи, дані картинки зберігаються у локальне сховище де використовуються для миттєвого відображення зображення для попереднього перегляду. У випадку, якщо користувач не збереже зміни, картинка не збережеться і відображення не зміниться. Якщо користувач

збереже зміни, то зображення відправить запитом до серверу та пройде процес зміни назви на унікальний ідентифікатор, збереження у статичну папку на сервері та збереження назви у базі даних в полі користувача. Надалі картинка буде відображатись як фото профілю. З псевдонімом все працює схожим чином, але без збереження файлу.

3.1.6 Пошук друзів

Пошук друзів здійснюється завдяки запиту на сервер з введеним користувачем псевдонімом іншого користувача. Відбувається пошук у базі даних і якщо такий існує, то користувацький інтерфейс відображає знайденого користувача. В усіх інших випадках відображається повідомлення про хибну назву чи не знайденого користувача.

3.2 Формування структури бази даних

Було спроектовано базу даних PostgreSQL що складається з 4 змістовних таблиць (рис. 3.1). Кожна з них використовується для зберігання даних, що належать користувачам. Детальніше про кожну з таблиць:

- users – таблиця для зберігання користувацьких даних. Вона складається з 11 полів. Кожне з них відповідає за конкретну частину функціоналу, що надає застосунок та є невід'ємною частиною програми. Поля email та password використовуються для проходження процесу авторизації користувача. В email зберігається унікальна пошта людини що зареєструвала акаунт. Поле password – це пароль користувача зашифрований бібліотекою bcryptjs. Поле username використовується для пошуку друзів у застосунку і є унікальним. Завдяки цьому полю можна здійснювати пошук у базі даних та знаходити конкретного користувача не використовуючи особисту пошту людини. Поле email_verified використовується для перевірки чи користувач підтвердив регістрацію за своєю поштою. Поля name та image це налаштування користувача що відображатимуться у застосунку. Поле

friends це масив друзів. Поле role відповідає за роль людини та може приймати значення звичайного користувача або адміністратора.

- dictionaries – таблиця для зберігання словників користувача. Містить у собі назву словника, тип та кількість елементів що у ньому знаходяться.
- entries – таблиця для зберігання записів що посилаються на конкретні словники. Це означає що може бути необмежена кількість записів що будуть мати один ідентифікатор словника. Містить у собі назву та значення запису що відображаються у застосунку.
- verification_codes – таблиця для зберігання кодів перевірки. Коди використовуються для підтвердження реєстрації акаунта.

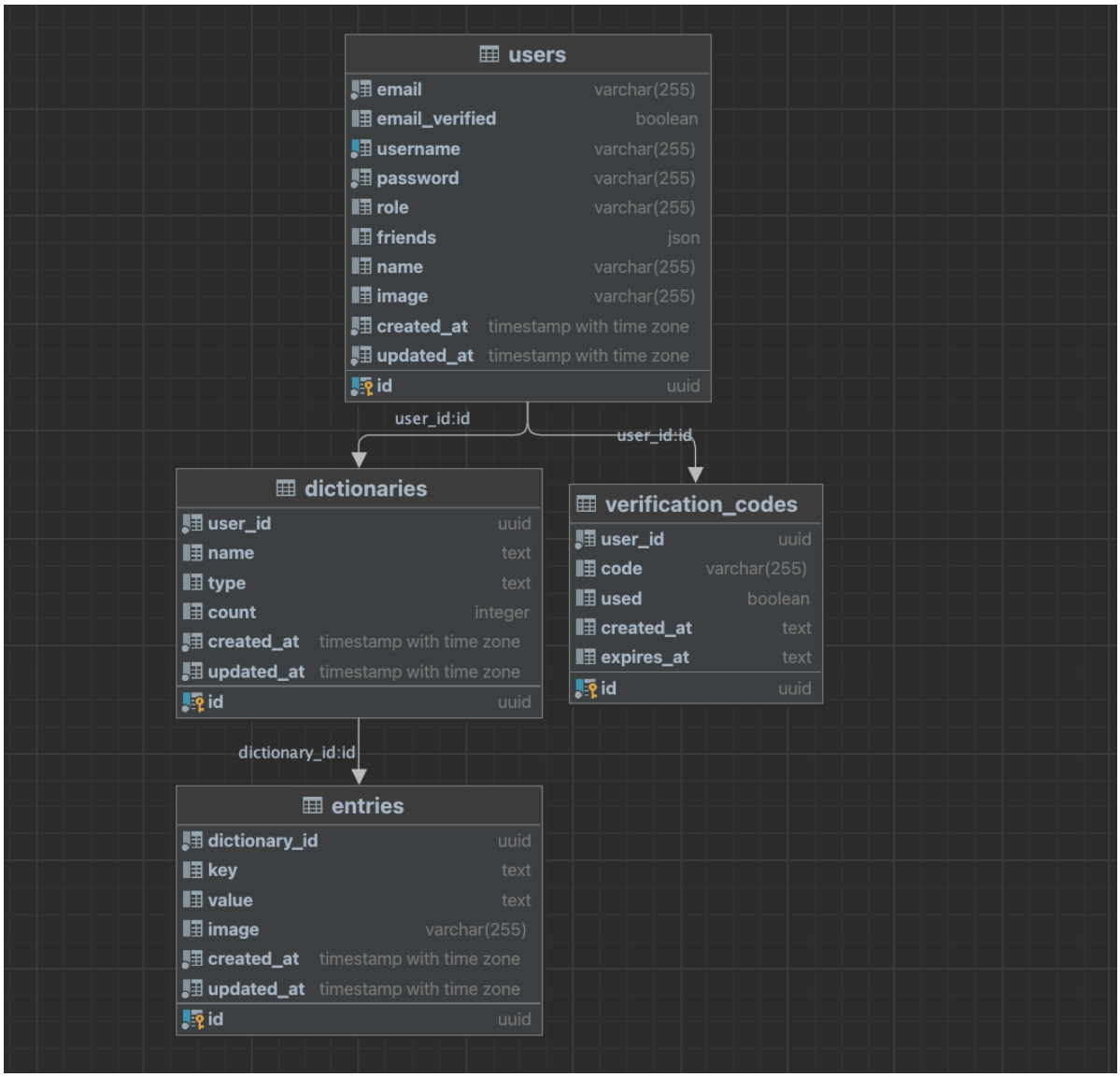


Рисунок 3.1 – Модель даних розробленої бази даних

3.3 Розробка компонентів системи застосунку

Застосунок створений на основі клієнт-сервер взаємодії. Тому слід поділити програму на дві основні складові, а саме серверну та клієнтську частини. Також варто зазначити, що застосунок має у своїй основі архітектуру MVC. Це означає що зв’язок між компонентами системи відбувається у вигляді взаємодії між основними складовими: контролерами, моделлю та представленням. Контролер та модель описані у серверній частині. Представлення розроблено у клієнтській частині.

3.3.1 Серверна частина

Кореневою складовою серверної частини застосунку є файл з маршрутами (рис. 3.2) за якими клієнтська частина отримує необхідні дані. Звертаючись до одного з маршрутів, клієнтська частина отримує відповідь, яка формується у функції, що прив'язана до цієї кінцевої точки. Ці функції називаються контролерами. Їх задача виконати усі необхідні маніпуляції з даними і за необхідності звертатись до бази даних, щоб записати, видалити, змінити або взяти дані сховища.

```
app.get('/status/', status.getStatus)
app.post( path: '/users/', users.create)
app.post( path: '/users/login/', users.login)
app.get('/users/check/', authCheck, users.check)

app.get('/users/', authCheck, users.getUsers)
app.delete( path: '/users/:userId/', authCheck, users.deleteOne)
app.get('/users/:userId/', users.getUser)
app.post( path: '/users/:userId/friends/', users.addFriends)
app.get('/users/:userId/friends/', authCheck, users.getFriends)
app.post( path: '/users/:userId/update-profile/', authCheck, users.updateProfile)
app.post( path: '/users/:userId/username/', authCheck, users.updateUsername)
app.post( path: '/verification-code/', email.sendEmail)
app.post( path: '/users/:userId/verification-codes/', verificationCodes.create)
app.post( path: '/users/:userId/verification-codes/:codeId/', verificationCodes.setAsUsed)

app.post( path: '/users/:userId/dictionaries/:dictionaryId/entries/', entries.createEntry)
app.get('/users/:userId/dictionaries/:dictionaryId/entries/', entries.getEntries)
app.delete( path: '/users/:userId/dictionaries/:dictionaryId/entries/:entryId/', entries.deleteEntry)

app.post( path: '/users/:userId/share-dictionary/', dictionaries.shareDictionary)
app.post( path: '/users/:userId/dictionaries/', dictionaries.createOrUpdateDictionary)
app.get('/users/:userId/dictionaries/', dictionaries.getDictionaries)
app.delete( path: '/users/:userId/dictionaries/:dictionaryId/', dictionaries.deleteDictionary)

app.post( path: '/notifications/', notifications.newNotification)
app.get('/notifications/', notifications.getNotifications)
```

Рисунок 3.2 – Основні API маршрути серверної частини

Також слід зауважити, що між контролерами та кінцевими точками в деяких місцях є проміжні функції. Це функції що здійснюють певні маніпуляції з даними безпосередньо перед виконанням основної функції. Як правило це функція, що перевіряє токен автентифікації на правильність. Але у програмі зустрічаються і функції для збереження файлів (рис. 3.3)

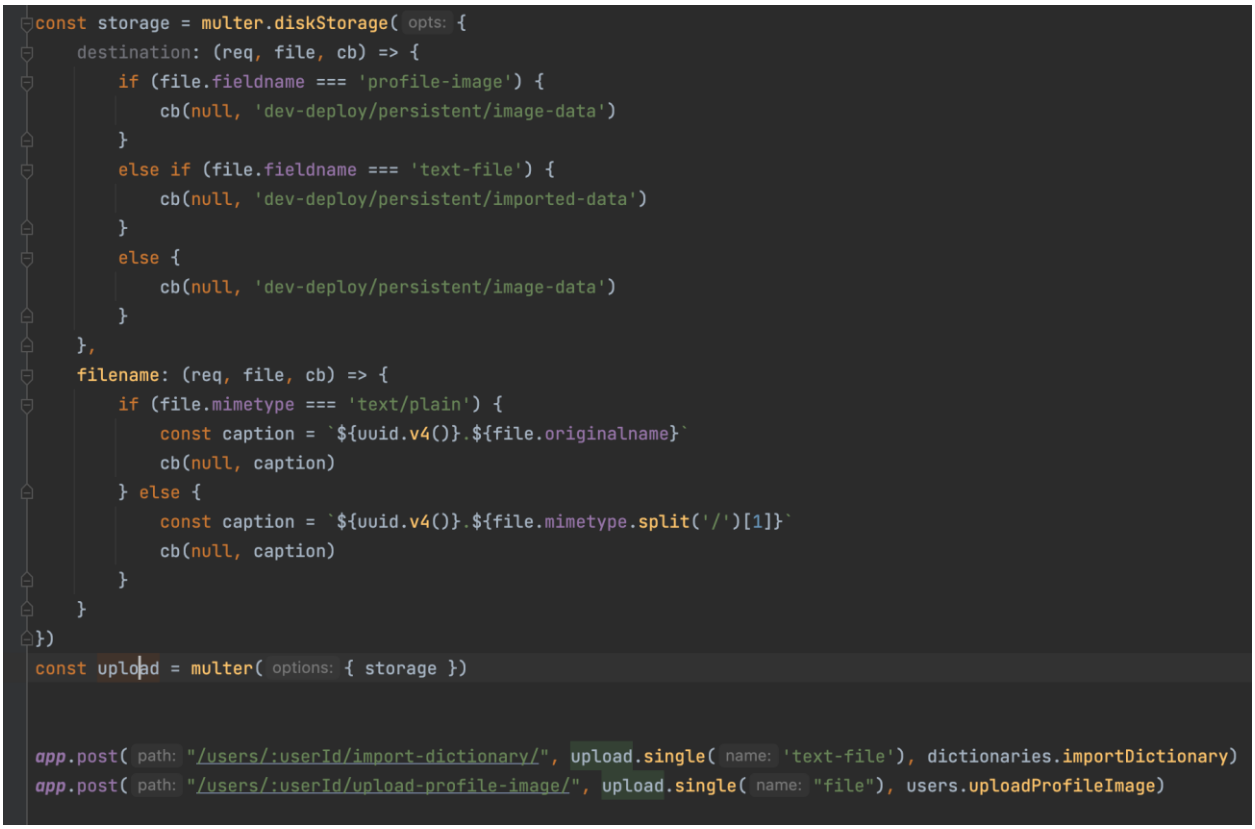


Рисунок 3.3 – Приклад застосування функції збереження файлів

Наступним кроком виконання є функція контролер. Кожна така функція отримує дані запиту і після виконання відправляє відповідь. Така відповідь може бути у вигляді оброблених даних або звичайного підтвердження що виконання пройшло успішно. Важливо розібрати основні функції контролери що формують логіку виконання програми.

Перша, це функція реєстрації (рис. 3.4). Вона працює таким чином, що отримуючи пошту та пароль користувача, перевіряє чи існує користувач з ідентичною поштою у системі, формує йому унікальний псевдонім та ім’я (що можуть бути зміненими у будь який момент після реєстрації), створює стандартні словники та додає нового користувача до бази даних.

Функція входу в систему, в свою чергу, отримує ті ж поля що і функція реєстрації, робить ту ж перевірку на наявність користувача у системі і у випадку, якщо дані введено вірно формує токен за яким користувач може вільно користуватись системою.

```
exports.create = async (req, res) => {
    const {email, password} = req.body

    if(!email) {
        return console.log(new Error('Email is required!'))
    }
    if(!password) {
        return console.log(new Error('Password is required!'))
    }

    // check for existence
    const candidate = await dal.users.getByEmail(email)
    if(candidate){
        return console.log(new Error('User with the same email is already exists!'))
    }

    const name = await helpers.createUniqueRandomName()
    const username = await helpers.createUniqueRandomName()
    const user = await dal.users.create(email, password, name, username)

    // default user's dictionary with entries
    const dictionary = await dal.dictionaries.create(user.id, name: 'Get Started')
    await dal.entries.create(dictionary.id, key: 'some word', value: 'some translation')

    return res.json({id: user.id})
}
```

Рисунок 3.4 – Функція реєстрації нового облікового запису користувача

Також варто згадати за функцію перевірки токена (рис. 3.5). Завдяки цій функції що є проміжною перевіркою, користувач без токену не зможе отримувати дані до моменту, поки не ввійде в свій обліковий запис. Це необхідно для того, щоб кожен окремий користувач системи мав змогу працювати лише зі своїм набором даних.

```
const authCheck = (req, res, next) => {
    if (req.method === "OPTIONS") {
        next()
    }

    try {
        let token = req.headers.authorization.split( separator: ' ')[1]
        if (!token) {
            return res.status(401).json({message: "Unauthorized"})
        }

        // throw 'error: jwt malformed' if not valid jwt
        const verifiedData = jwt.verifyAccessToken(token)
        if(verifiedData) {
            req.user = verifiedData
        }
        next()
    } catch (e) {
        return res.status(401).json({message: "Unauthorized"})
    }
}
```

Рисунок 3.5 – Функція перевірки токена

Далі можна згрупувати функції що мають схоже призначення. Такими є функція створення запису у словнику, функція видалення словника, функція зміни паролю, назви словника, імені і подібні. Для прикладу обрано функцію створення або оновлення словника (рис. 3.6). Головна мета таких функцій – це обробити дані та здійснити маніпуляцію з базою даних. Результат таких функцій – це оновлені дані та повідомлення зі статусом 200, що означає успішне виконання.

```
exports.createOrUpdateDictionary = async (req, res) => {
    const userId = req.params.userId
    const dictionaryId = req.body.dictionaryId || null
    const name = req.body.name
    if(!name) {
        return console.log(new Error('name is required!'))
    }

    if (dictionaryId) {
        await dal.dictionaries.update(dictionaryId, fields: {name})
        return res.json({dictionaryId, name})
    }
    const dictionary = await dal.dictionaries.create(userId, name)
    res.json(dictionary)
}
```

Рисунок 3.6 – Функція створення або оновлення словника

Крім контролерів, на серверній частині розроблено модель даних. Цей шар архітектури MVC відповідає за обробку даних та зв'язок з базою даних. Він виконаний з використанням ORM Sequelize для більш гнучкої та швидкої взаємодії з даними з таблиць бази даних PostgeSQL.

Для прикладу взято функцію створення нового користувача у базі даних (рис. 3.7). У цій функції створюється унікальний ідентифікатор, шифрується пароль, присвоюється стандартне значення ролі, перевірки пошти і подібні дії. Далі, ця інформація записуються як окремий рядок у базі даних. Функції, що виконують дії з іншими таблицями PostgreSQL працюють схожим чином. Змінюється лише дія, а саме якщо потрібно видалити дані то задається дія видалення та передається ідентифікатор запису, якщо потрібно оновити дані то передаються нові дані та ідентифікатори записів для оновлення.

```
exports.create = async (email, password, name, username, emailVerified :boolean = false) => {
    const id = uuid.v4()

    let hashedPassword
    if (password) {
        hashedPassword = await hash.hashPassword(password)
    }

    const user = {
        id,
        email: email.toLowerCase(),
        emailVerified,
        username,
        password: hashedPassword,
        role: constants.USER_ROLES.USER,
        name
    }
    await db.user.create(user)
    return user
}
```

Рисунок 3.7 – Функція створення нового користувача в базі даних

Функції, що знаходяться у шарі моделі виконують дії над моделями що мають низку атрибутів. У об’єкті атрибутів прописано стандартні значення для нових записів, унікальність, дозволи на пустоту і тому подібне (рис. 3.8). При створенні бази даних, за даними атрибутами формуються поля цієї бази даних.

```
const user = sequelize.define( modelName: 'user',  attributes: {
    id: {type: DataTypes.UUID, primaryKey: true, allowNull: false},
    email: {type: DataTypes.STRING, allowNull: false},
    emailVerified: {type: DataTypes.BOOLEAN, defaultValue: false},
    username: {type: DataTypes.STRING, allowNull: false, unique: true},
    password: {type: DataTypes.STRING, allowNull: false},
    role: {type: DataTypes.STRING, defaultValue: constants.USER_ROLES.USER},
    friends: {type: DataTypes.JSON, defaultValue: []},
    name: {type: DataTypes.STRING},
    image: {type: DataTypes.STRING},
}, options: {underscored: true})
```

Рисунок 3.8 – Модель даних для користувача

Одним з допоміжних модулів серверної частини є модуль з бібліотеками. Основне призначення цієї частини програмного коду – це додати функціонал, що буде використовуватись у інших шарах архітектури як допоміжний і не буде перевантажувати кодом функції де написано і так багато логіки. Важлива особливість цього модуля це те, що він може бути використаними повторно у різних частинах програмного коду. Більшість модулів цього проекту використовують сторонні модулі та бібліотеки. Для прикладу взято один з важливих для правильної роботи програми модулів що відповідає за шифрування та співставлення паролів (рис. 3.9). Його задача зашифрувати пароль щоб той надалі міг бути збереженим у базу даних. Для перевірки правильності паролю буде використовуватись функція для порівняння зашифрованих паролів.

```
const bcrypt = require("bcryptjs")

exports.hashPassword = async (password) => {
    return await bcrypt.hashSync(password, salt: 7)
}

exports.comparePasswords = async (password, hash) => {
    return await bcrypt.compare(password, hash)
}
```

```
const compared = await hash.comparePasswords(password, user.password)
if (!compared) {
    return console.log(new errors.NotFound('user does not exist or wrong password'))
}
```

Рисунок 3.9 – Функція шифрування та порівняння паролів а також її використання

3.3.2 Клієнтська частина

Кореневою складовою клієнтської частини застосунку можна вважати файл з HTTP запитами (рис. 3.10) за якими клієнтська частина отримує необхідні дані з серверу. Файл з функціями-запитами на сервер можуть робити запит з додаванням JWT токену або без, в залежності від необхідності. Такі функції приймають параметром дані для запиту, як приклад ідентифікатор користувача чи поля що слід оновити.

```
const createOrUpdateDictionary = async (data) => {
    const userId = data.userId
    const body = {
        name: data.name
    }
    if (data.dictionaryId) {
        body.dictionaryId = data.dictionaryId
    }
    return await host.post( url: `/users/${userId}/dictionaries/`, body)
}

const getDictionaries = async (userId) => {
    return await host.get( url: `/users/${userId}/dictionaries/`)
}

const deleteDictionary = async (userId, dictionaryId) => {
    return await host.delete( url: `/users/${userId}/dictionaries/${dictionaryId}/`)
}

const shareDictionary = async (userId, dictionaryId, recipientId) => {
    return await authHost.post( url: `/users/${userId}/share-dictionary/`,  data: {dictionaryId, recipientId})
}
```

Рисунок 3.10 – Маршрути з HTTP запитами на сервер

Функції-запити використовуються у компонентах, з логікою для окремих сторінок застосунку (рис. 3.11), а також у допоміжних компонентах що є окремими частинами користувацького інтерфейсу.

```
useEffect( effect: () => {
    updateData()
}, deps: [])

const updateData = () => {
    getDictionaries(user.id).then((response : AxiosResponse<any> ) => {
        let dictionaries = response.data.reverse()

        // add edit, icon rows for every
        for (const entity of dictionaries) {
            entity.edit = false
            entity.iconIndex = generateRandomDigit()
        }
        setData(dictionaries)
    })
}

const newDictionary = async () => {
    const newDictionary = await createOrUpdateDictionary( data: {userId: user.id, name: state.nameOfNew})
    setData( value: prevState => [newDictionary.data, ...prevState])
    setState( value: {...state, nameOfNew: ''})
}
```

Рисунок 3.11 – Приклад використання функцій що здійснюють запит на сервер

Існує всього 5 сторінок: сторінка зі словниками (головна), сторінка з записами в словник, сторінка з налаштуваннями, сторінка для пошуку друзів та сторінка автентифікації. Усі ці сторінки мають схожу архітектуру. Слід розглянути основні відмінні частини для загального розуміння роботи окремих частин програми з клієнтської частини.

Одна з важливих функцій застосунку – це функція створення та відслідковування сповіщень (Рис. 3.12). Вона працює таким чином, що один користувач підписується на подію з відправкою повідомлення та постійно відслідковує нові запити. Інший користувач, в свою чергу, відправляє другу повідомлення з словником чи запрошенням у друзі по ідентифікатору чи псевдоніму. Повідомлення відразу з'являється у отримувача в сповіщеннях з діями прийняти та відхилити запит. У випадку коли користувач приймає запит, відбувається запит на створення копії словника та відразу перезавантажується користувацький інтерфейс з оновленими даними на екрані. Після успішного виконання цієї дії, отримувач знов підписується на запити. Цей підхід називається Event Sourcing [29]

```
const [notifications, setNotifications] = useState( initialState: [])
useEffect( effect: () => {
    subscribe().catch(e => console.log(e))
}, deps: [])

const subscribe = async () => {
    const eventSource = new EventSource( url: `http://localhost:8000/notifications/`)
    eventSource.onmessage = function (event : MessageEvent ) {
        const data = JSON.parse(event.data) // {dictionaryId, recipientId, message, id, senderImageUrl}
        const newPost = {
            ...data,
            action: async () => await shareCurrentDictionary(data),
            // TODO fix dropdown shutdown on setState. Save notifications in db?
            cancel: () => setNotifications( value: [...notifications.filter(item => item.id !== data.id)])
        }
        setNotifications( value: prev => [newPost, ...prev])
    }
    console.log(notifications)
}
```

Рисунок 3.12 – Функція для відслідковування нових повідомлень у панелі сповіщень

Також, не менш важливою є функція оновлення налаштувань профілю користувача (рис. 3.13). Вона виконує запит на сервер з даними що мають бути оновленими. Функція використовує систему відправки форм що дозволяє відправити картинку що може бути збережена та потім використана для відображення у користувацькому інтерфейсі. Після відправки запиту з’являється повідомлення про виконання запиту. Це повідомлення формується допоміжною сторонньою бібліотекою що має гнучку систему налаштувань.

```
const [preview, setPreview] = useState( initialState: '')
const updateUserProfile = async () => {
    try {
        if (toUpdate.name && toUpdate.name !== '') {
            await updateProfile(user.id,  fields: {name: toUpdate.name})
            setToUpdate( value: {...toUpdate, name: ''})
        }
        if (toUpdate.image) {
            const formData = new FormData()
            formData.append( name: "name", toUpdate.image.name)
            formData.append( name: "file", toUpdate.image)
            const response = await sendProfileImage(user.id, formData)

            // update global store
            context.user.updateUserData( toUpdate: {image: response.data.image})
        } else {
            console.log('no file to upload')
        }

        // notification
        toast.success( content: 'Profile updated',  options: {
            position: "top-right", autoClose: 2000,
            hideProgressBar: false, closeOnClick: true,
            pauseOnHover: true, draggable: true
        })
    } catch (err) {
        console.log(err)
    }
}
```

Рисунок 3.13 – Функція оновлення налаштувань користувача

Для збереження даних користувача, таких як картинка профілю, ідентифікатор користувача та його друзів і подібні використовується клас з методами які оновлюються з використанням бібліотеки MobX [30] що дозволяє використовувати ці дані усюди де необхідно у програмі(рис. 3.14). Цей підхід значно полегшує навантаження на сервер оскільки нема необхідності кожен раз робити запит з метою заново отримати дані де не обхідно.

```
setUser(user) {
    this._user = user
}
updateUserData(toUpdate) {
    this._user = { ...this._user, userData: { ...toUpdate } }
}
```

Рисунок 3.14 – Функції збереження даних для подальшого використання у різних частинах програми

ВИСНОВОК ДО РОЗДІЛУ 3

У результаті процесу проектування і розробки багатокористувацької системи ведення словників, було розглянуто 7 основних сценаріїв взаємодії користувача з програмою.

Автентифікація користувача у системі дозволяє отримати повний доступ до функціоналу застосунку, що в свою чергу дає можливість створювати власні налаштування та словники з записами. Функції взаємодії з словниками дозволяють оновлювати необхідні поля у словнику, змінювати поля записів а також видаляти і те і інше. Функція поширення словників іншим користувачам зі списку друзів дозволяє створити копію словника на іншому акаунті після підтвердження дії отримувачем. Завдяки функції сповіщень, отримувач матиме можливість контролювати події відправки словника та додавання у друзі. Імпортування словника з файлу дозволяє створити новий словник з записами, що використовувались у інших програмах. Налаштування профілю дають можливість змінити картинку, ім’я, пароль, псевдонім користувача за його ініціативи. Є можливість бачити фото профілю до його збереження. Пошук друзів дозволяє додати інших користувачів системи у список друзів для подальшої відправки власних словників цим користувачам.

РОЗДІЛ 4. ТЕСТУВАННЯ РОЗРОБЛЕНОГО ЗАСТОСУНКУ

Перш за все, слід чітко відмежувати складові частини застосунку. Найбільш зручно буде зробити поділ по сторінкам, а саме: сторінка авторизації, сторінка зі словниками (головна сторінка), а також сторінки налаштування, пошуку друзів та записів у словник. Важливо зазначити, що інтерфейс застосунку спроектовано з використанням навігаційної панелі для швидкого переходу на необхідну сторінку (рис. 4.1). Компонент сповіщень розташовано на цій панелі разом з логотипом застосунку та картинкою користувача.

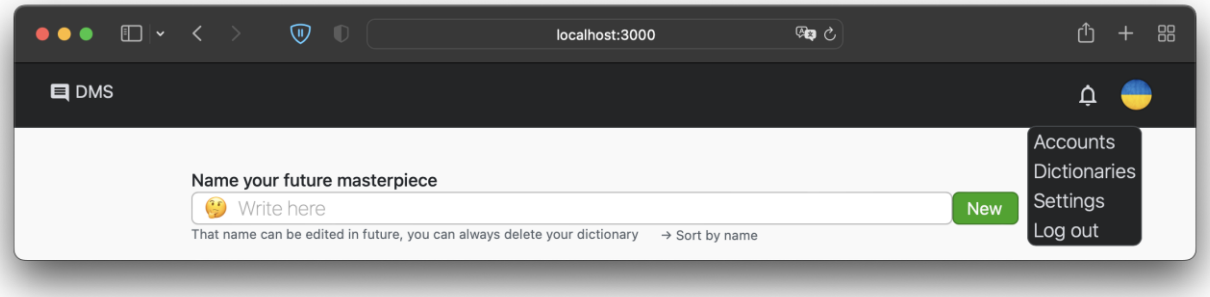


Рисунок 4.1 – Навігаційна панель з відкритим списком посилань на сторінки

4.1 Сторінка авторизації користувача.

Неавторизований користувач автоматично переадресовується на сторінку входу в систему де має можливість зареєструвати новий аккаунт або ж зайти в існуючий. У випадку реєстрації користувачу необхідно ввести свою пошту, придумати пароль, записати його та натиснути зелену кнопку знизу (рис 4.2). Надалі ці данні можуть бути використаними для входу в акаунт. Для цього необхідно натиснути на текст між зеленою кнопкою та полями вводу даних аккаунта. Далі ввести дані що вказувались при реєстрації та натиснути зелену кнопку входу. У випадку, якщо дані введено невірно, з’явиться сповіщення з

вказівкою спробувати ще раз. Якщо дані введено вірно то система дасть доступ до функціоналу застосунку і перенаправить користувача на головну сторінку зі словниками.

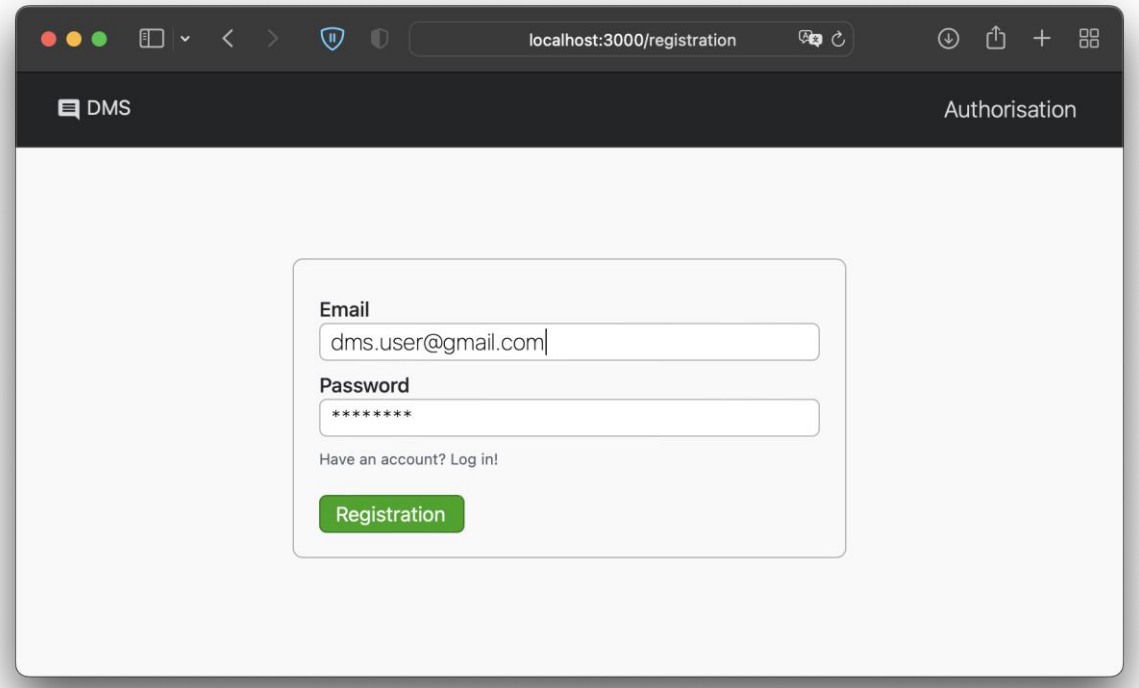


Рисунок 4.2 – Сторінка авторизації користувача

4.2 Сторінка зі словниками.

Після проходження процесу авторизації, користувач потрапляє на головну сторінку застосунку (рис. 4.3), а саме сторінку зі словниками. Тут розташовано основний функціонал застосунку:

- Компонент для створення нових словників
- Список з наявними словниками
- Кнопка сортування наявних словників
- Кнопка опцій для кожного окремого словника, за допомогою якої можна поширювати словник друзям, змінювати його назву та видаляти його у випадку необхідності.

- Жовта кнопка для імпортування в полі вводу зліва

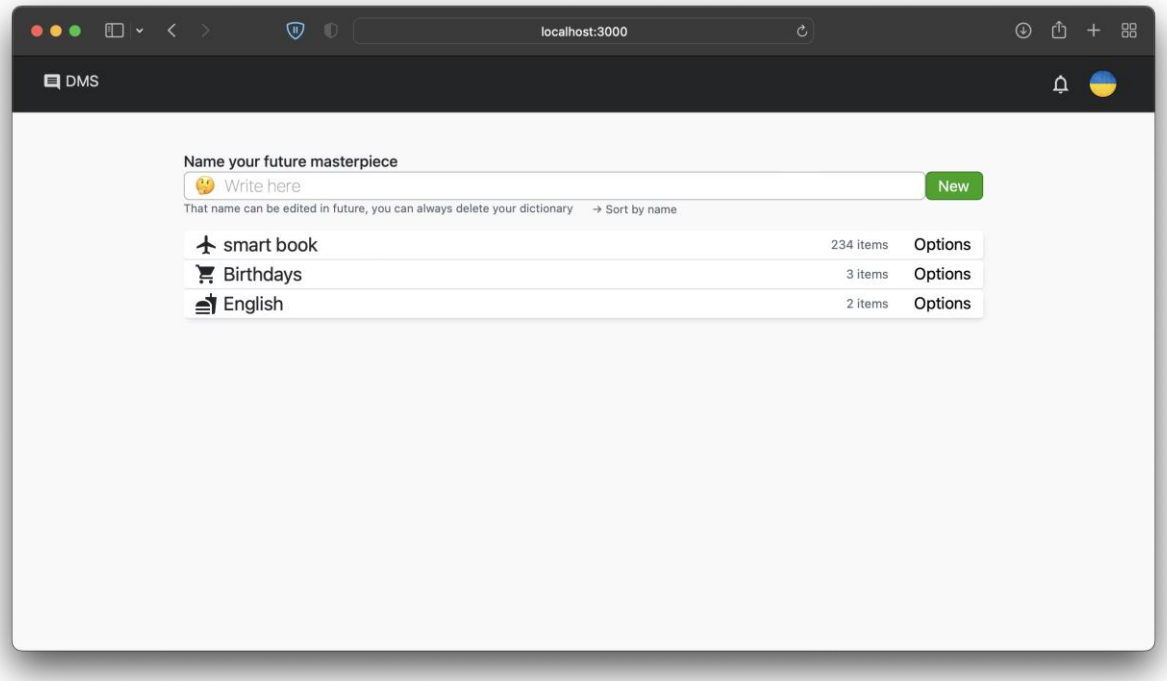


Рисунок 4.3 – Сторінка словників

Тепер детальніше про кожну з наведених функцій. Компонент для створення словників складається з поля вводу назви та зеленої кнопки, що при натисканні створює новий запис який відразу відображається на екрані користувача. У випадку, коли користувач залишає поле вводу пустим і натискає кнопку, система не створює запису і просить користувача ввести назву. Поле залишається незмінним незалежно від кількості створених словників чи їх розміру.

Список зі словниками складається з переліку наявних у користувача словників, що були додані власноруч або імпортовані з файлу. Він може змінюватись в залежності від дій людини. Як приклад, користувач може створювати нові словники і кожен наступний буде переміщуватись на початок списку що дозволяє відслідковувати який з них був доданий раніше чи пізніше.

Кожен з елементів списку має показник де вказано кількість слів всередині словника, іконку біля назви та кнопку опцій.

Кнопка опцій відображає панель, що випадає при натисканні (рис. 4.4). На цій панелі присутні три кнопки. Кнопка ‘Share’ при натисканні, відображає панель з переліком друзів авторизованого користувача (рис. 4.5). При натисканні на одного з користувачів з переліку, йому відправляється повідомлення з прикріпленим словником. Йому залишається підтвердити або ж відхилити запит. Кнопка ‘Edit’ дозволяє змінювати назву словника. При натисканні, назва словника змінюється на поле для вводу з кнопками для підтвердження або відміни (рис. 4.6). У цьому полі можна ввести нову назву та зберегти. Кнопка ‘Delete’ видалить словник а також усі записи у ньому.

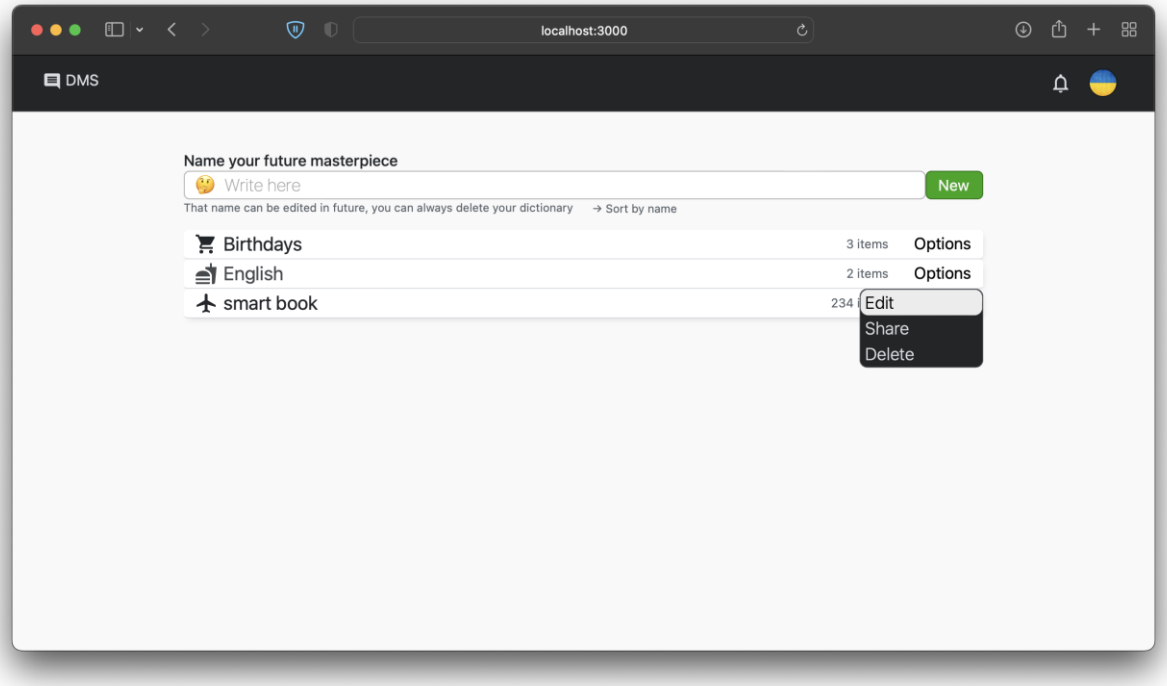


Рисунок 4.4 – Опції окремого словника

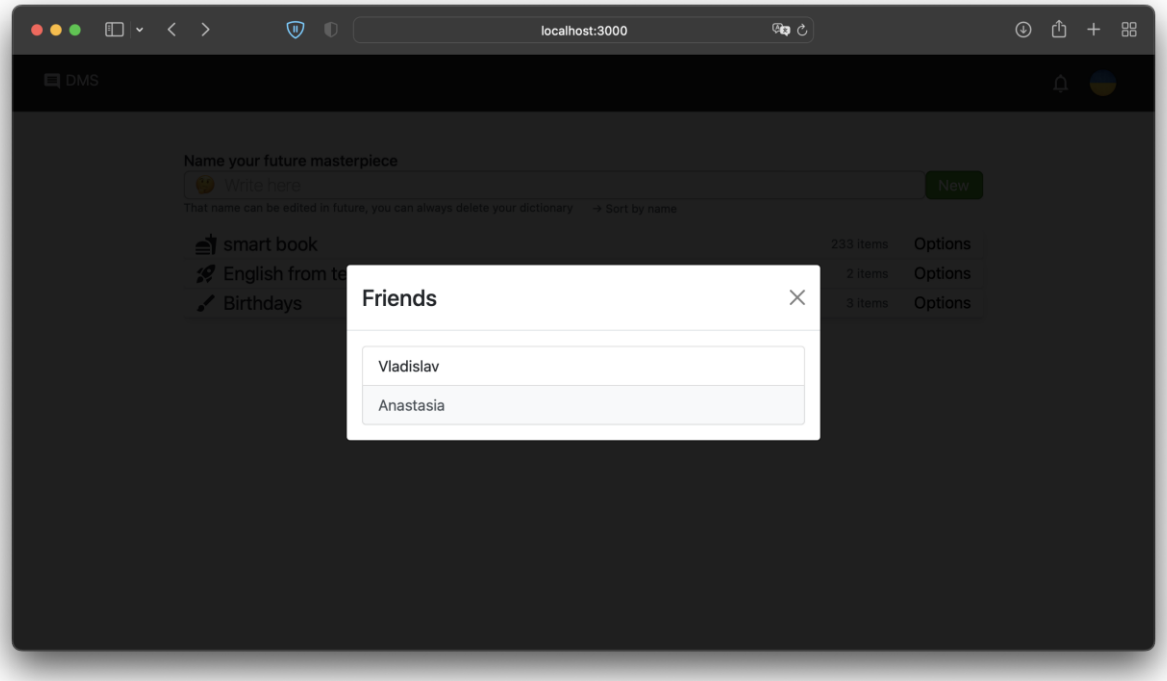


Рисунок 4.5 – Опція відправки вибраного словника друзям

Між списком словників та полем для створення нових словників є кнопка для сортування. При натисканні словники переміщуються, та упорядковуються за алфавітом.

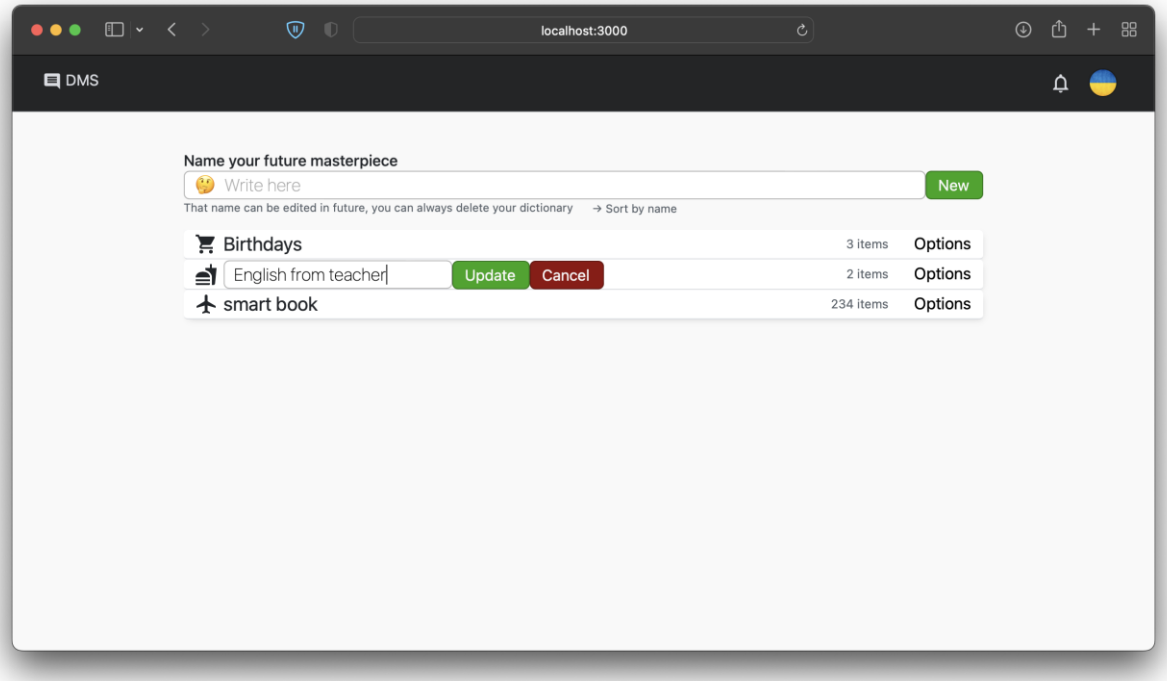


Рисунок 4.6 – Опція редагування назви словника

Один з важливих функціоналів застосунку, це імпорт словників зі сторонніх текстових файлів експортованих з інших програм (рис. 4.7). Кнопка знаходить зліва від поля вводу та при натисканні відкриває панель де можна обрати файл для імпорту. Після підтвердження, сформується новий словник із записами що знаходились у файлі.

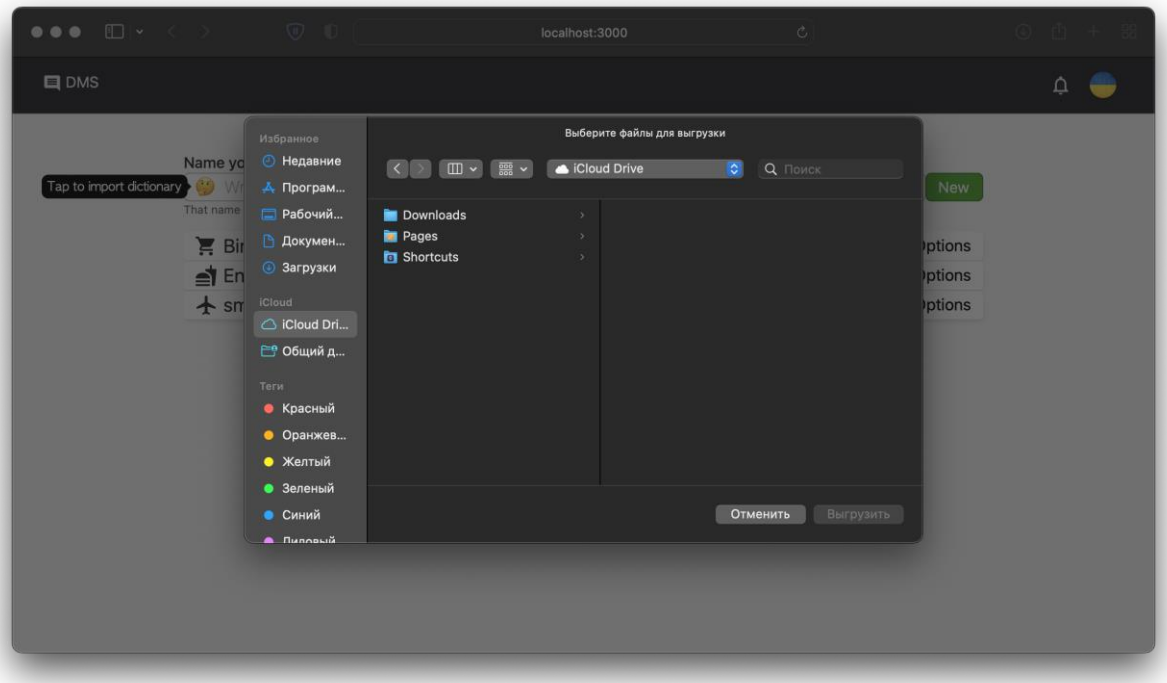


Рисунок 4.7 – Імпортування словника з файлу.

4.3 Сторінка записів вибраного словника.

Авторизований користувач, що натискає на вибраний словник з головної сторінки, переходить на сторінку з записами цього словника (рис. 4.8). На цій сторінці є поля для вводу назви та значення запису. Це може бути як приклад ім’я та дата народження когось із друзів чи родичів або іншомовне слово та його переклад. Таких варіантів може бути безліч. При натисканні зеленої кнопки ‘New’ створюється та зберігається новий запис та відображається першим у списку. Запис можна видалити на кнопку у формі хрестика справа.

Також на сторінці присутня кнопка для сортування слів за назвою по алфавіту. Функція зручна у випадках коли слів стає надто багато.

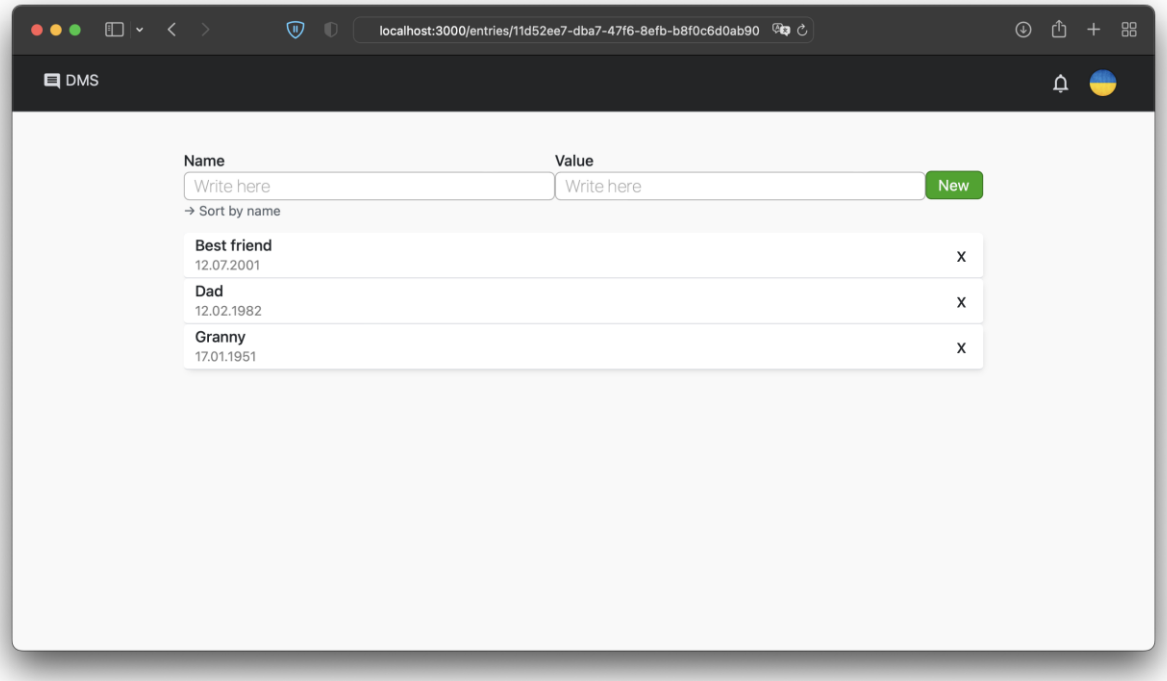


Рисунок 4.8 – Записи у словнику

4.4 Сповіщення.

Панель сповіщень знаходиться у навігаційній панелі. Натискаючи на іконку з зображенням дзвону відкривається вікно з усіма сповіщеннями. Це можуть бути сповіщення про додавання у друзі (рис. 4.9) чи запит на отримання нового словника від друга (рис. 4.10).

Справа від тексту сповіщення розташовано дві кнопки: зелена та червона. При натисканні на зелену користувач приймає запит, а на червону відхиляє та видаляє повідомлення. Зліва від тексту знаходиться картинка профілю відправника.

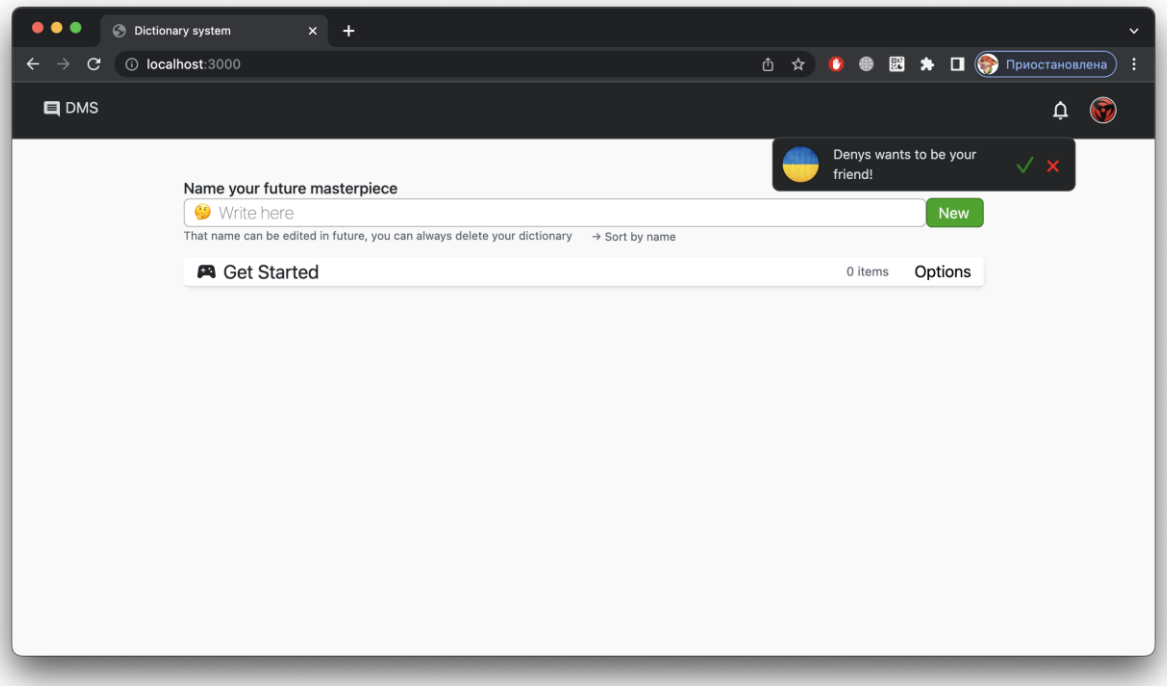


Рисунок 4.9 – Сповіщення з повідомленням про додавання у друзі

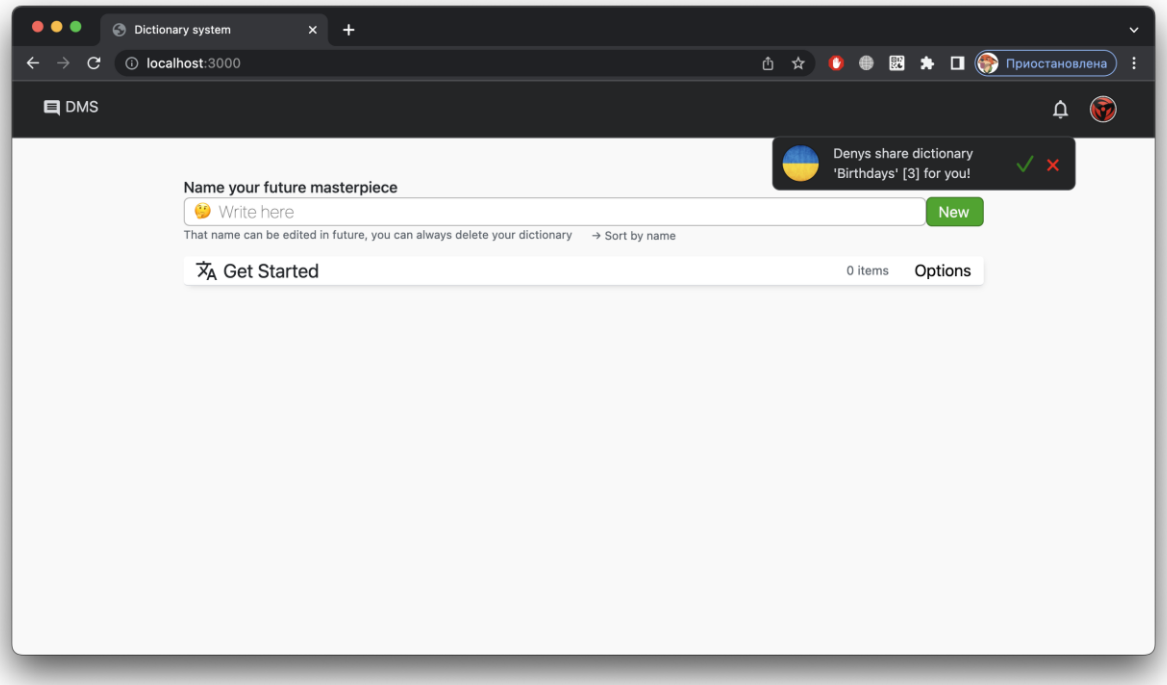


Рисунок 4.10 – Меню сповіщень з повідомленням про отримання словника

4.5 Сторінка пошуку друзів.

Авторизований користувач може здійснювати пошук інших користувачі на сторінці пошуку друзів (рис. 4.11). Інша назва – сторінка акаунтів. Щоб здійснити пошук потрібно ввести псевдонім користувача (вказується на сторінці налаштування) та натиснути на зелену кнопку пошуку. Усі варіанти з’являться у списку нижче. Кожен елемент списку буде складатись з імені та псевдоніма користувача. Натискаючи на цей елемент, користувачу зі списку буде відправлено повідомлення про додавання у друзі. У випадку, якщо цей користувач вже є у списку друзів, з’явиться повідомлення у правій верхній частині екрану про те що користувач вже є другом.

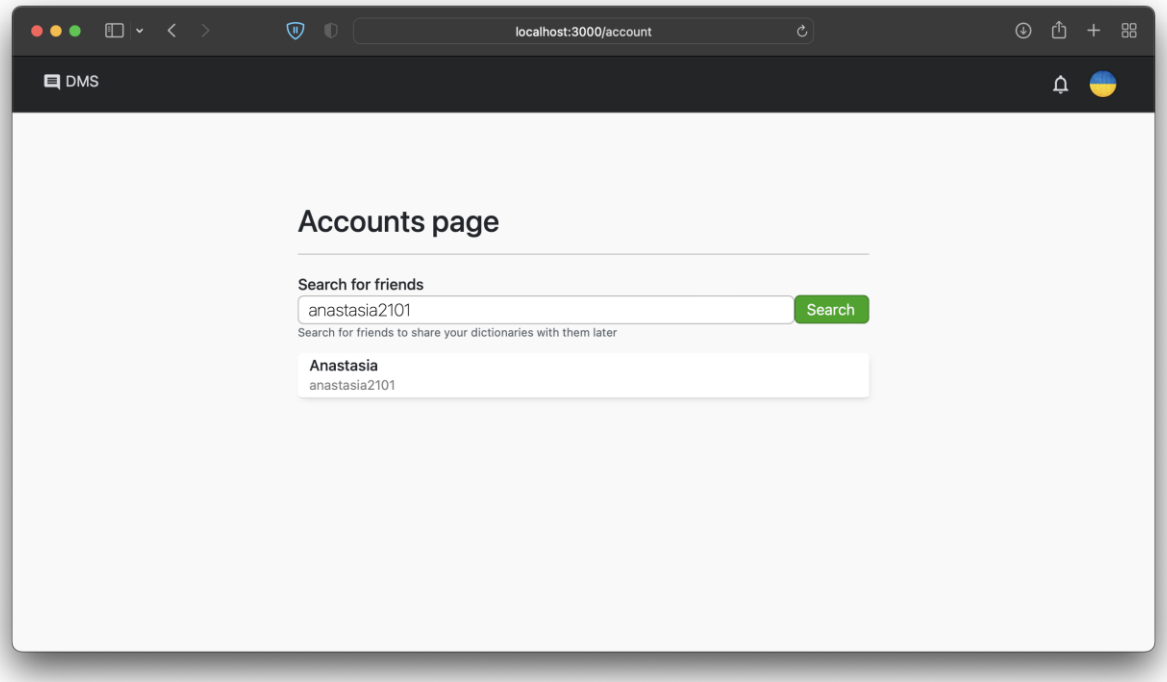


Рисунок 4.11 – Сторінка для пошуку друзів

4.6 Сторінка налаштувань.

Авторизований користувач може робити зміни у власному профілі на сторінці налаштувань. Сторінка складається з двох вкладок.

Перша вкладка – це налаштування профілю. Тут користувач може змінювати ім’я та картинку профілю (рис. 4.12). Зміни збережуться після натискання зеленої кнопки. Це означає що користувач може змінити картинку, відразу побачити як вона буде виглядати у вікні і лише потім, за бажанням, зберегти результат. У випадку якщо зміни не подобаються, користувач може просто не зберігати і картинка залишиться такою, що була раніше. Картинку профілю бачать усі користувачі що здійснюють пошук інших користувачів чи відправляють словники друзям. Також картинка відображається у правій верхній частині сторінки у навігаційній панелі та дає візуальне розуміння в якому акаунті авторизований користувач.

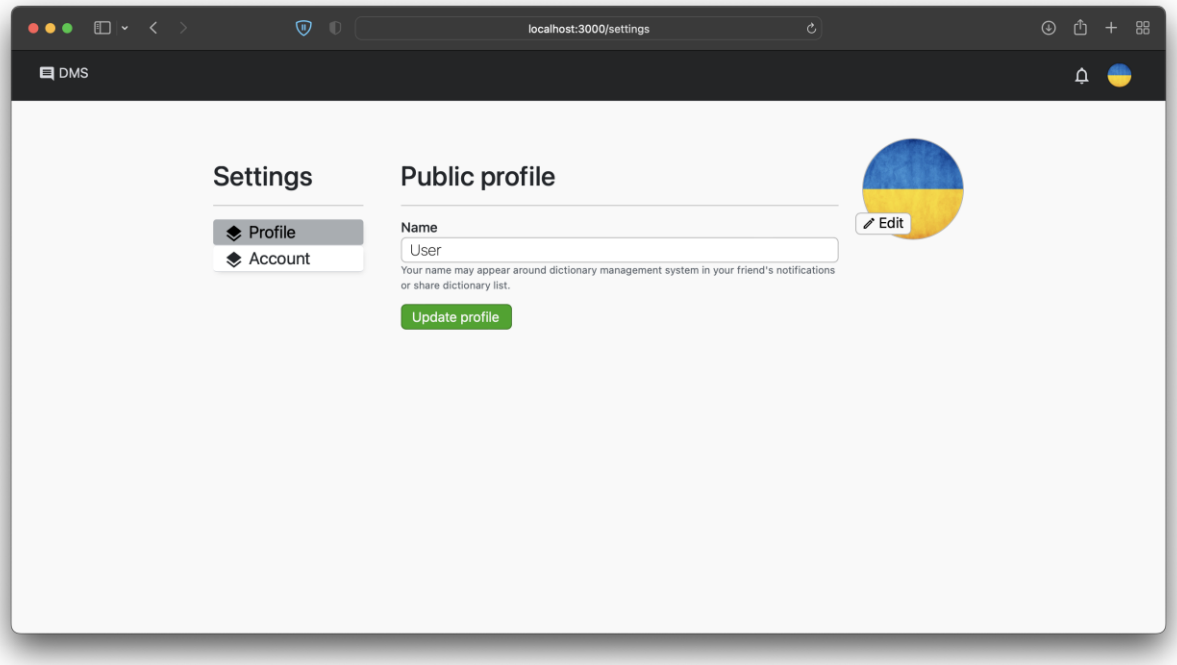


Рисунок 4.12 – Налаштування профілю.

Друга вкладка – це налаштування акаунта (рис. 4.13). Тут користувач може змінювати псевдонім, завдяки якому здійснюється пошук друзів. Також є поле для зміни паролю.

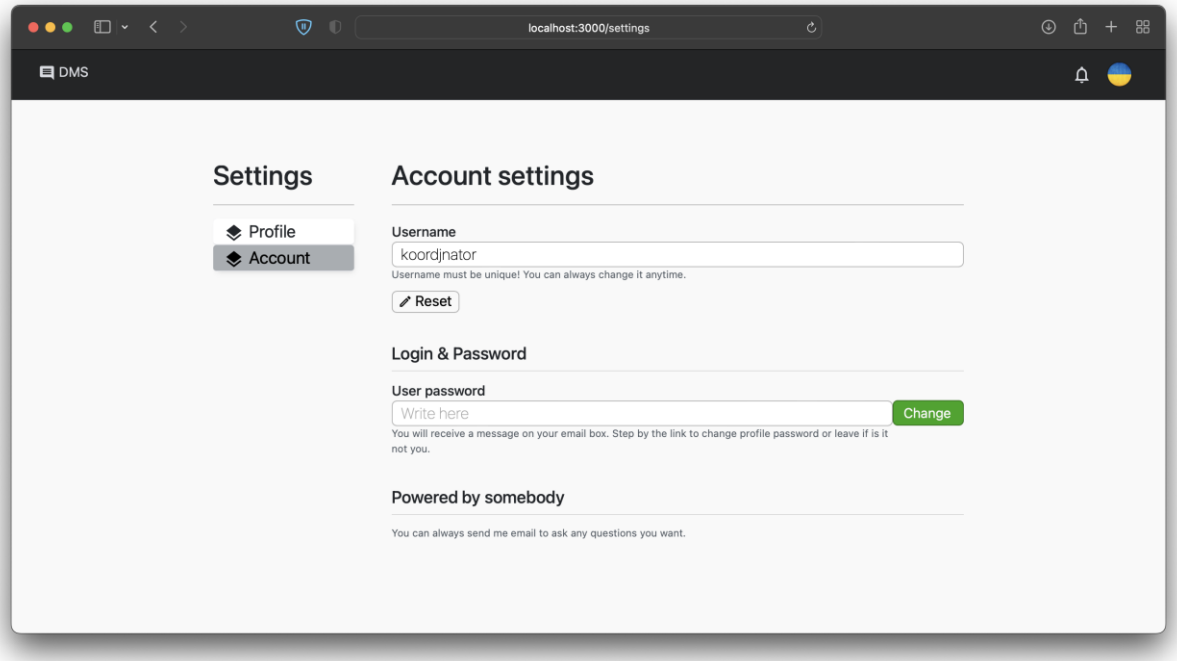


Рисунок 4.13 – Налаштування акаунта.

ВИСНОВОК ДО РОЗДІЛУ 4

Було опрацьовано усі можливі сценарії роботи програми. Авторизований користувач отримує повний доступ до функціоналу програми що включає в собі створення словників, їх редагування, наповнення, видалення та поширення. Завдяки сторінці налаштувань, користувач має змогу змінювати картинку профілю, псевдонім, пароль та ім’я. Здійснювати пошук можна на сторінці акаунтів. Є функціональність імпортування словників з інших програм. Система сповіщень реалізована на навігаційній панелі, що дозволяє побачити повідомлення з будь-якої сторінки застосунку. Неавторизований користувач потрапляє на сторінку авторизації та має можливість створити новий акаунт та користуватись ним.

ВИСНОВКИ

В роботі розглянуто проблему з величезними обсягами інформації, запропоновано шляхи її вирішення і розроблено систему ведення словників з можливістю поділити дані на категорії.

Огляд та аналіз існуючих аналогів показує, що є функція яка не була реалізована у жодному з застосунків або є реалізованою частково. Цією функцією є поширення записів у вигляді словників іншим користувачам.

Багатокористувацька система ведення словників була розроблена у вигляді вебзастосунку на мові JavaScrypt що робить її доступною з усіх пристроїв що мають доступ до мережі Інтернет. Було реалізовано низку функцій для зручного використання. Функція поширення словників іншим користувачам дозволяє поширювати власні записи серед інших зацікавлених груп осіб. Функція імпорту дозволяє використовувати записи з інших додатків. Налаштування профілю дає можливість змінювати оформлення та підписи користувача.

Розробка має потенціал для росту. Програма може бути використаною у комерційних проектах як основа для іншомовного словника. Система дозволяє фільтрувати потік текстової інформації та поділити його на категорії, що спрощує процес пошуку даних.

СПИСОК ВИКОРИСТАНОЇ ЛІТЕРАТУРИ

1. Spaced Repetition: Remembering What You Learn. URL: https://www.kpu.ca/sites/default/files/Learning%20Centres/Think_SpacedRepetition_LA.pdf
2. Davis, Ian. "What Are The Benefits of MVC?". Internet Alchemy. Retrieved 2016-11-29.
3. Description of the Model-View-Controller User Interface Paradigm in the Smalltalk-80 System
4. Languages popularity in github. URL: https://madnight.github.io/githut/#/pull_requests/2021/3
5. Languages popularity in github. URL: https://madnight.github.io/githut/#/pull_requests/2021/3
6. Changes in JavaScript over time. URL: https://codeburst.io/changes-in-java-script-over-time-8243f67259dc
7. Top 5 Programming languages for Web development in 2022. URL: https://medium.com/javarevisited/top-5-programming-languages-for-web-development-in-2021-f6fd4f564eb6 (дата звернення 22.05.2022)
8. Most used programming languages among developers worldwide, as of 2021. URL: https://www.statista.com/statistics/793628/worldwide-developer-survey-most-used-languages/
9. Smashing Node.js: JavaScript Everywhere, John Wiley & Sons, 14-Aug-2012
10. Cross Platform Implementation. URL: https://reactnative.dev/architecture/xplat-implementation
11. What are .io games. URL: https://io-games.fandom.com/wiki/.io_Games_Wiki
12. JavaScript data types and data structures. URL: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Data_structures

13. TypeScript stands in an unusual relationship to JavaScript. URL: https://www.typescriptlang.org/docs/handbook/typescript-in-5-minutes.html
14. The merits of strong typing. URL: https://dhh.dk/arc/000074.html
15. ECMAScript 2021 Language Specification. URL: https://262.ecma-international.org/12.0/
16. Stack Overflow Developer Survey 2021. URL: https://insights.stackoverflow.com/survey/2021#section-most-popular-technologies-programming-scripting-and-markup-languages (дата звернення 22.05.2022)
17. High-performance JavaScript and WebAssembly engine V8. URL: https://v8.dev/docs
18. Asynchronous programming. Blocking I/O and non-blocking I/O. URL: https://luminousmen.com/post/asynchronous-programming-blocking-and-non-blocking
19. Express web framework (Node.js/JavaScript). URL: https://developer.mozilla.org/en-US/docs/Learn/Server-side/Express_Nodejs
20. What makes Node.js so fast. URL: https://www.peerbits.com/blog/why-nodejs-fast-and-its-best-use-cases.html
21. What is the Event Loop? URL: https://nodejs.org/en/docs/guides/event-loop-timers-and-nexttick/#what-is-the-event-loop
22. Stack Overflow Developer Survey 2021. URL: https://insights.stackoverflow.com/survey/2021#section-most-popular-technologies-web-frameworks (дата звернення 22.05.2022)
23. React Lifecycle. URL: https://www.w3schools.com/react/react_lifecycle.asp
24. Performance Overview. URL: https://reactnative.dev/docs/performance
25. JSX. URL: https://facebook.github.io/jsx/
26. HTTP Messages. URL: https://developer.mozilla.org/en-US/docs/Web/HTTP/Messages
27. DB-Engines Ranking - Trend of PostgreSQL Popularity. URL: https://db-engines.com/en/ranking_trend/system/PostgreSQL

28. JSON Web Token (JWT). URL: https://datatracker.ietf.org/doc/html/rfc7519
29. Pattern: Event sourcing. https://microservices.io/patterns/data/event-sourcing.html
30. MobX documentation. URL: https://mobx.js.org/README.html

ДОДАТОК 1

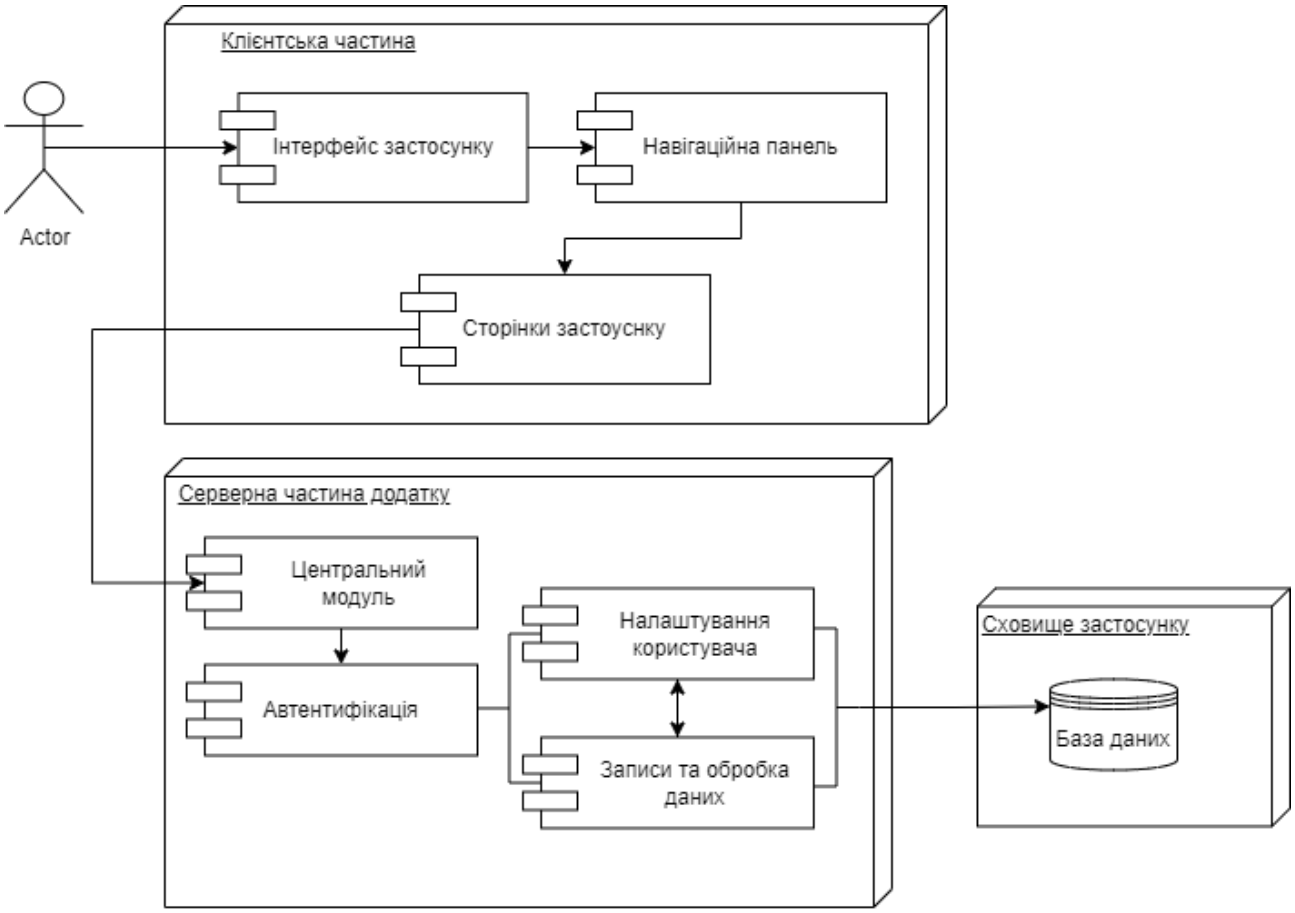
Багатокористувацька система ведення словників

Структурна схема – діаграма структури системи

ІАЛЦ.467200.004 Д1

Аркушів 1

Київ 2022 р



					ІАЛЦ.467200.004 Д1		
		№ докум.	Підпис	Дата			
Розробив		Мітячкін Д. Є.			**Багатокористувацька система ведення словників** Структурна схема – діаграма структури системи	Літ.	Аркуш 1 / Аркушів 1
Перевірив		Молчанова А.А					
						НТУУ КПІ ім. Ігоря Сікорського, ФІОТ, ІО-83	
Н. Контр.		Сімоненко В. П.					
Затвердив							

ДОДАТОК 2

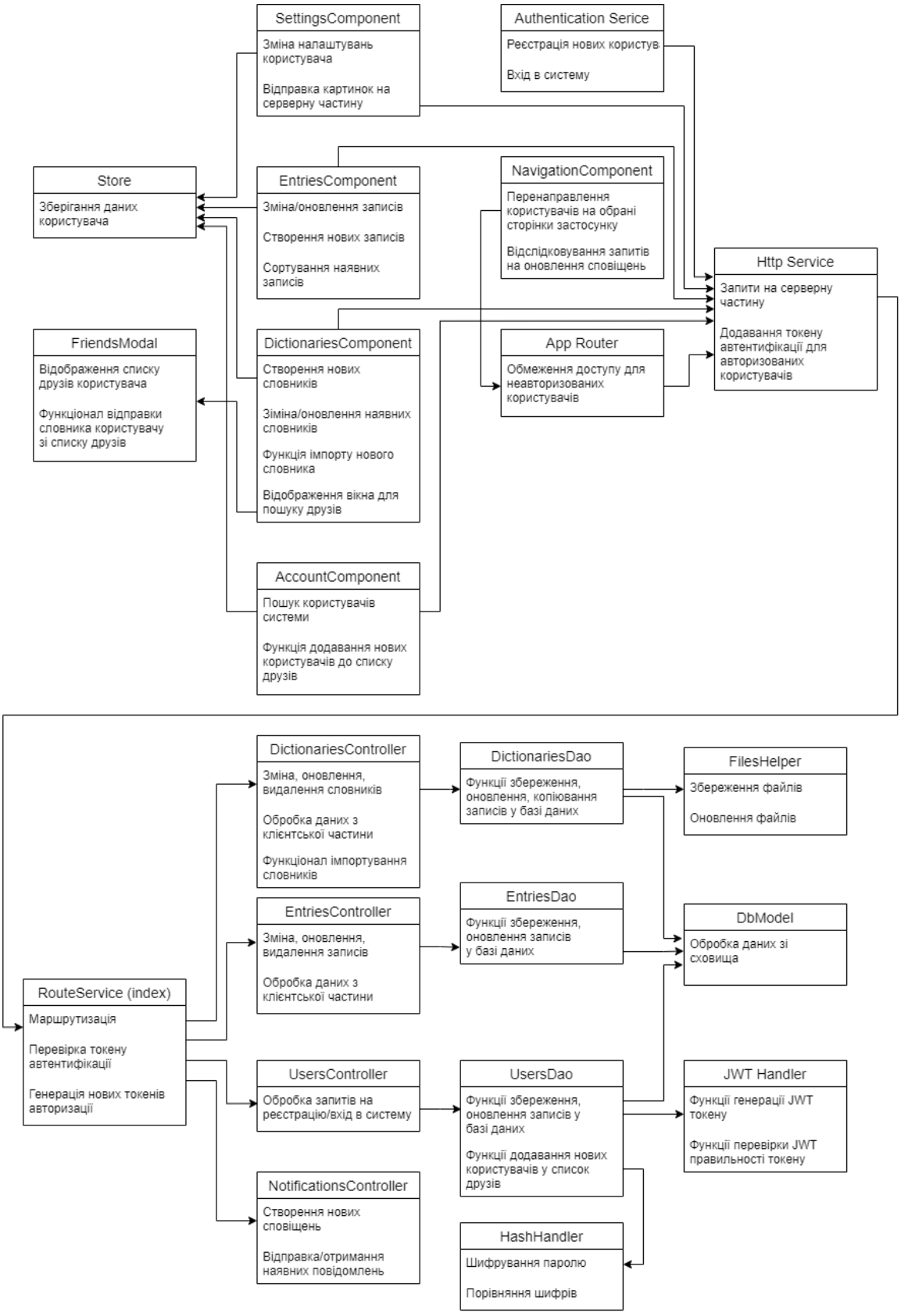
Багатокористувацька система ведення словників

Функціональна схема – діаграма програмних компонентів

ІАЛЦ.467200.005 Д2

Аркушів 1

Київ 2022 р



					ІАЛЦ.467200.005 Д2	
		№ докум.	Підпис	Дата		
Розробив		Мітячкін Д. Є.			**Багатокористувацька система ведення словників** Функціональна схема – діаграма програмних компонентів	Літ. / Аркуш 1 / Аркушів 1
Перевірив		Молчанова А.А				
Н. Контр.		Сімоненко В. П.				НТУУ КПІ ім. Ігоря Сікорського, ФІОТ, ІО-83
Затвердив						

ДОДАТОК 3

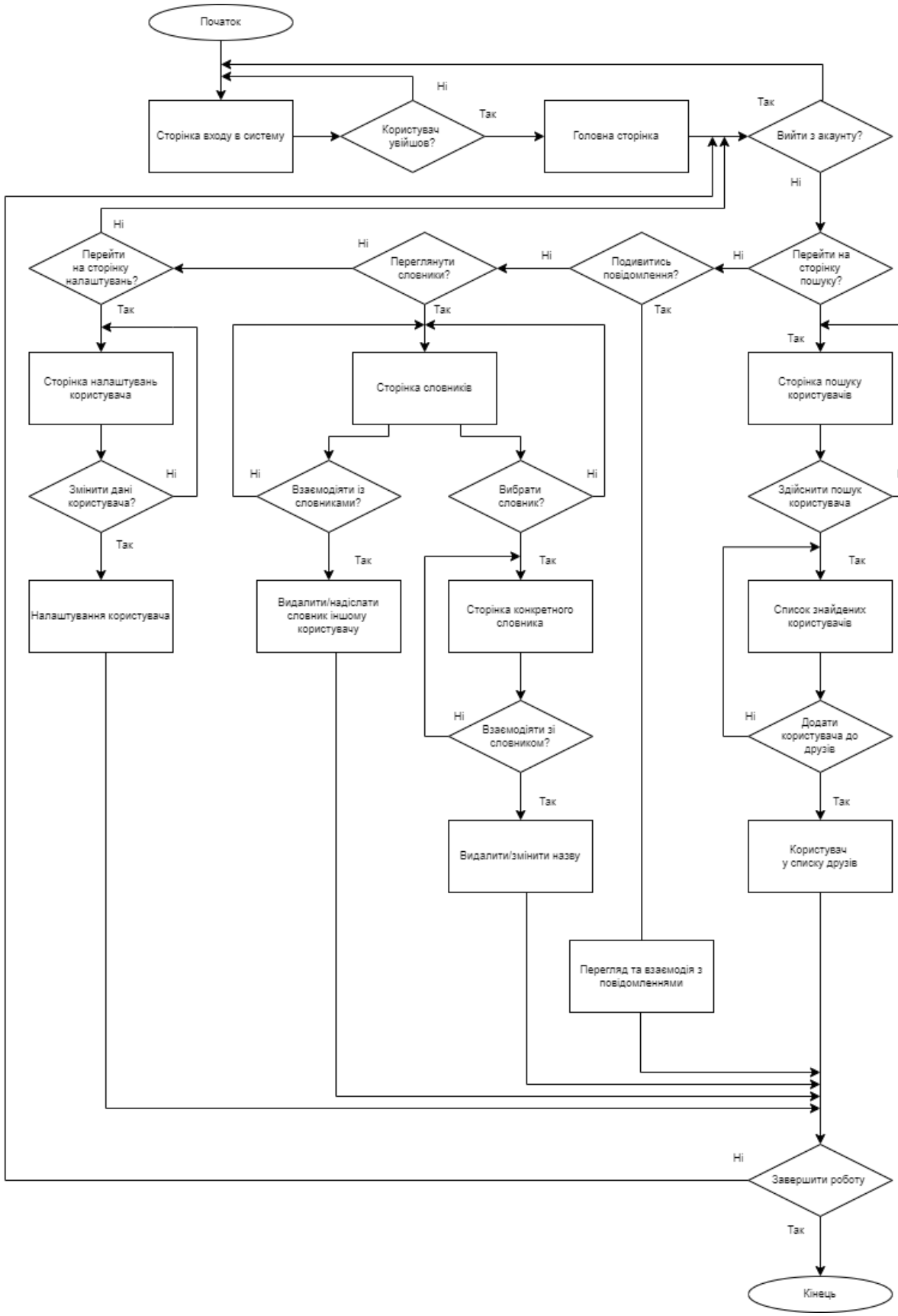
Багатокористувацька система ведення словників

Принципова схема – алгоритм роботи програми

ІАЛЦ.467200.006 Д3

Аркушів 1

Київ 2022 р



					ІАЛЦ.467200.006 Д3	
		№ докум.	Підпис	Дата		
Розробив		Мітячкін Д. Є.			**Багатокористувацька система ведення словників** Принципова схема – алгоритм роботи програми	Літ. / Аркуш 1 / Аркушів 1
Перевірив		Молчанова А.А				НТУУ КПІ ім. Ігоря Сікорського, ФІОТ, ІО-83
Н. Контр.		Сімоненко В. П.				
Затвердив						

ДОДАТОК 4

Багатокористувацька система ведення словників

Текст програмного коду

ІАЛЦ.467200.007 Д4

Аркушів 27

Київ 2022 р

index.js

```
'use strict'

const express = require('express')
const cors = require('cors')
const jwt = require('./lib/jwt')
const multer = require('multer')
const path = require('path')
const uuid = require('uuid')

const status = require('./controllers/status')
const users =  require('./controllers/users')
const email =  require('./controllers/email')
const verificationCodes = require('./controllers/verificationCodes')
const entries = require('./controllers/entries')
const dictionaries = require('./controllers/dictionaries')
const notifications = require('./controllers/notifications')


const app = express()

app.use(express.static(path.resolve(__dirname, '../dev-deploy/persistent/image-
data/')))
app.use(express.json())
app.use(cors())


const authCheck = (req, res, next) => {
    if (req.method === "OPTIONS") {
        next()
    }

    try {
        let token = req.headers.authorization.split(' ')[1]
        if (!token) {
            return res.status(401).json({message: "Unauthorized"})
        }

        // throw 'error: jwt malformed' if not valid jwt
        const verifiedData = jwt.verifyAccessToken(token)
        if(verifiedData) {
            req.user = verifiedData
        }
```

```
        next()
    } catch (e) {
        return res.status(401).json({message: "Unauthorized"})
    }
}

app.get('/status/', status.getStatus)

app.post('/users/', users.create)
app.post('/users/login/', users.login)
app.get('/users/check/', authCheck, users.check)

app.get('/users/', authCheck, users.getUsers)
app.delete('/users/:userId/', authCheck, users.deleteOne)
app.get('/users/:userId/', users.getUser)


app.post('/users/:userId/friends/', users.addFriends)
app.get('/users/:userId/friends/', authCheck, users.getFriends)

app.post('/users/:userId/update-profile/', authCheck, users.updateProfile)
app.post('/users/:userId/username/', authCheck, users.updateUsername)


app.post('/verification-code/', email.sendEmail)

app.post('/users/:userId/verification-codes/', verificationCodes.create)
app.post('/users/:userId/verification-codes/:codeId/', verificationCodes.setAsUsed)


app.post('/users/:userId/dictionaries/:dictionaryId/entries/', entries.createEntry)
app.get('/users/:userId/dictionaries/:dictionaryId/entries/', entries.getEntries)
app.delete('/users/:userId/dictionaries/:dictionaryId/entries/:entryId/',
entries.deleteEntry)
app.get('/random/:dictionaryId/', entries.getRandomOne)

app.post('/users/:userId/share-dictionary/', dictionaries.shareDictionary)



app.post('/users/:userId/dictionaries/', dictionaries.createOrUpdateDictionary)
app.get('/users/:userId/dictionaries/', dictionaries.getDictionaries)
app.delete('/users/:userId/dictionaries/:dictionaryId/',
dictionaries.deleteDictionary)

app.post('/notifications/', notifications.newNotification)
app.get('/notifications/', notifications.getNotifications)
const storage = multer.diskStorage({
    destination: (req, file, cb) => {
        if (file.fieldname === 'profile-image') {
            cb(null, 'dev-deploy/persistent/image-data')
```

```
        }
        else if (file.fieldname === 'text-file') {
            cb(null, 'dev-deploy/persistent/imported-data')
        }
        else {
            cb(null, 'dev-deploy/persistent/image-data')
        }
    },
    filename: (req, file, cb) => {
        if (file.mimetype === 'text/plain') {
            const caption = `${uuid.v4()}.${file.originalname}`
            cb(null, caption)
        } else {
            const caption = `${uuid.v4()}.${file.mimetype.split('/')[1]}`
            cb(null, caption)
        }
    }
})
const upload = multer({ storage })


app.post("/users/:userId/import-dictionary/", upload.single('text-file'),
dictionaries.importDictionary)
app.post("/users/:userId/upload-profile-image/", upload.single("file"),
users.uploadProfileImage)

module.exports = app
```

db.js

```
const {DataTypes, Sequelize} = require('sequelize')
const constants = require('./lib/constants')

const sequelize = new Sequelize(
    process.env.DB_NAME || 'dms',
    process.env.DB_USER || 'root',
    process.env.DB_PASSWORD || 'root',
    {
        dialect: 'postgres',
        host: process.env.LOCALHOST || process.env.REMOUTE_HOST || '0.0.0.0',
        port: parseInt(process.env.DB_PORT || '5432')
    }
)


const user = sequelize.define('user', {
    id: {type: DataTypes.UUID, primaryKey: true, allowNull: false},
    email: {type: DataTypes.STRING, allowNull: false},
    emailVerified: {type: DataTypes.BOOLEAN, defaultValue: false},
    username: {type: DataTypes.STRING, allowNull: false, unique: true},
    password: {type: DataTypes.STRING, allowNull: false},
```

```
    role: {type: DataTypes.STRING, defaultValue: constants.USER_ROLES.USER},
    friends: {type: DataTypes.JSON, defaultValue: []},
    name: {type: DataTypes.STRING},
    image: {type: DataTypes.STRING},
}, {underscored: true})


const verificationCodes = sequelize.define('verificationCodes', {
    id: {type: DataTypes.UUID, primaryKey: true, allowNull: false},
    userId: {type: DataTypes.UUID, allowNull: false},
    code: {type: DataTypes.STRING},
    used: {type: DataTypes.BOOLEAN, defaultValue: false},
    createdAt: {type: DataTypes.TEXT},
    expiresAt: {type: DataTypes.TEXT}
}, {underscored: true, updatedAt: false, createdAt: false})


const dictionaries = sequelize.define('dictionaries', {
    id: {type: DataTypes.UUID, primaryKey: true, allowNull: false},
    userId: {type: DataTypes.UUID, allowNull: false},
    name: {type: DataTypes.TEXT, defaultValue: 'My dictionary'},
    type: {type: DataTypes.TEXT, defaultValue:
constants.DICTIONARY_TYPES.STANDARD},
    count: {type: DataTypes.INTEGER, defaultValue: 0},
}, {underscored: true})


const entries = sequelize.define('entries', {
    id: {type: DataTypes.UUID, primaryKey: true, allowNull: false},
    dictionaryId: {type: DataTypes.UUID, allowNull: false},
    key: {type: DataTypes.TEXT},
    value: {type: DataTypes.TEXT},
    image: {type: DataTypes.STRING},
}, {underscored: true})


module.exports = {
    sequelize,
    user,
    verificationCodes,
    dictionaries,
    entries
}
```

dictionaries.js

```
const dal = require('../dal')
const files = require('../lib/files')


exports.createOrUpdateDictionary = async (req, res) => {
```

```
    const userId = req.params.userId
    const dictionaryId = req.body.dictionaryId || null
    const name = req.body.name
    if(!name) {
        return console.log(new Error('name is required!'))
    }

    if (dictionaryId) {
        await dal.dictionaries.update(dictionaryId, {name})
        return res.json({dictionaryId, name})
    }
    const dictionary = await dal.dictionaries.create(userId, name)
    res.json(dictionary)
}

exports.getDictionaries = async (req, res) => {
    const userId = req.params.userId
    const dictionaries = await dal.dictionaries.getByUserId(userId)
    res.json(dictionaries)
}

exports.deleteDictionary = async (req, res) => {
    const id = req.params.dictionaryId
    await dal.dictionaries.deleteById(id)
    await dal.entries.deleteAllByDictionaryId(id)
    res.json({id})
}

exports.updateDictionary = async (req, res) => {
    // const userId = req.params.userId
    const {dictionaryId, name} = req.body
    if(!dictionaryId || !name) {
        return console.log(new Error('dictionaryId and name is required!'))
    }

    const fields = {
        name,
    }
    const dictionary = await dal.dictionaries.update(dictionaryId, fields)
    res.json(dictionary)
}

exports.shareDictionary = async (req, res) => {
    // const userId = req.params.userId
    const {recipientId, dictionaryId} = req.body

    const newDictionary = await dal.dictionaries.copyDictionary(recipientId,
dictionaryId)

    const entries = await dal.entries.getByDictionaryId(dictionaryId)
    if(entries) {
```

```
        for(let entry of entries) {
            await dal.entries.create(newDictionary.id, entry.key, entry.value)
        }
    }
    res.json({message: "OK"})
}

exports.importDictionary = async (req, res) => {
    const userId = req.params.userId

    const newDictionary = await dal.dictionaries.create(userId,
req.file.originalname.split('.')[0])
    const entries = await files.getDataFromImportedFile(req.file.path)
    if(entries) {
        for(const entry of entries) {
            await dal.entries.create(newDictionary.id, entry[0], entry[1])
        }
    }
    res.json({message: "OK"})
}
```

entries.js

```
const dal = require('../dal')

const _randomOne = (arr) => {
    return arr[Math.floor(Math.random() * arr.length)]
}

exports.createEntry = async (req, res) => {
    const dictionaryId = req.params.dictionaryId
    const {key, value} = req.body
    if(!key || !value) {
        throw new Error('key and value is required!')
    }

    const entry = await dal.entries.create(dictionaryId, key, value)
    res.json(entry)
}

exports.getEntries = async (req, res) => {
    const dictionaryId = req.params.dictionaryId
    const entries = await dal.entries.getByDictionaryId(dictionaryId)
    res.json(entries)
}

exports.deleteEntry = async (req, res) => {
    const {entryId} = req.params
    await dal.entries.deleteById(entryId)
    res.json({id: entryId})
```

```
}

exports.getRandomOne = async (req, res) => {
    const dictionaryId = req.params.dictionaryId
    const entries = await dal.entries.getByDictionaryId(dictionaryId)
    return res.json(_randomOne(entries))
}
```

users.js

```
const dal = require('../dal')
const jwt = require('../lib/jwt')
const errors = require('../lib/errors')
const hash = require('../lib/hash')
const helpers = require('../lib/helpers')


exports.create = async (req, res) => {
    const {email, password} = req.body

    if(!email) {
        return console.log(new Error('Email is required!'))
    }
    if(!password) {
        return console.log(new Error('Password is required!'))
    }

    // check for existence
    const candidate = await dal.users.getByEmail(email)
    if(candidate){
        return console.log(new Error('User with the same email is already
exists!'))
    }

    const name = await helpers.createUniqueRandomName()
    const username = await helpers.createUniqueRandomName()
    const user = await dal.users.create(email, password, name, username)

    // default user's dictionary with entries
    const dictionary = await dal.dictionaries.create(user.id, 'Get Started')
    await dal.entries.create(dictionary.id, 'some word', 'some translation')

    return res.json({id: user.id})
}


exports.login = async (req, res) => {
    const {email, password} = req.body
    // const browser = req.headers['user-agent']
```

```
    // const ip = req.ip

    if (email == null) {
        return console.log(new errors.BadRequest('email isn\'t provided'))
    }
    if (password == null) {
        return console.log(new errors.BadRequest('password isn\'t provided'))
    }

    const user = await dal.users.getByEmail(email)
    if (!user) {
        return console.log(new errors.NotFound('user does not exist or wrong
password'))
    }

    const compared = await hash.comparePasswords(password, user.password)
    if (!compared) {
        return console.log(new errors.NotFound('user does not exist or wrong
password'))
    }

    const token = jwt.generateAccessToken(user.id, user.role)
    return res.json({token, user})
}

exports.check = async (req, res) => {
    const token = jwt.generateAccessToken(req.user.id, req.user.role)
    const user = await dal.users.getById(req.user.id)
    user.password = null
    return res.json({token, user})
}


exports.getUsers = async (req, res) => {
    const users = await dal.users.getAll()
    res.json(users)
}

exports.getUser = async (req, res) => {
    const userId = req.params.userId // or username

    let user = await dal.users.getByUsername(userId)
    if (!user) {
        user = await dal.users.getById(userId).catch(() => {
            return res.json(null)
        })
    }

    if (user) {
        user.password = null
    }
```

```
    res.json(user)
}


exports.updateUsername = async (req, res) => {
    const userId = req.params.userId
    const username = req.body.username
    await dal.users.updateUsername(userId, username).catch(err => {return
console.log(err)})
    return res.json("OK")
}


exports.deleteOne = async (req, res) => {
    const userId = req.params.userId
    await dal.users.deleteById(userId)
    res.json({message: "OK"})
}

exports.updateProfile = async (req, res) => {
    const userId = req.params.userId

    /**
     * fields: {name, city, theme...}
     */
    const fields = req.body.fields
    await dal.users.updateUserFields(userId, fields)
    res.json({message: "OK"})
}

exports.addFriends = async (req, res) => {
    const userId = req.params.userId
    const friendId = req.body.friendId

    const user = await dal.users.getById(userId)
    for (const userId of user.friends) {
        if (userId === friendId) {
            return res.status(500).send('User already has this id in friend list')
        }
    }

    // add user to the friend's friend list too
    await dal.users.addToFriendsList(friendId, userId)

    const response = await dal.users.addToFriendsList(userId, friendId)
    res.json(response)
}

exports.getFriends = async (req, res) => {
    const userId = req.params.userId
    const user = await dal.users.getById(userId)
```

```
    let friends = []
    for (const userId of user.friends) {
        const friend = await dal.users.getById(userId)
        friends.push({id: userId, name: friend.name})
    }

    res.json(friends)
}


exports.uploadProfileImage = async (req, res) => {
    const userId = req.params.userId
    try {
        const user = await dal.users.updateUserFields(userId, {image:
req.file.filename})
        console.log(`db.users.image updated, filename: ${req.file.filename}`)
        res.json({image: user.image})
    } catch (err) {
        console.log(err)
    }
}
```

dal/users.js

```
const uuid = require('uuid')

const db = require('../db')
const constants = require('../lib/constants')
const hash = require('../lib/hash')


exports.create = async (email, password, name, username, emailVerified = false) =>
{
    const id = uuid.v4()

    let hashedPassword
    if (password) {
        hashedPassword = await hash.hashPassword(password)
    }

    const user = {
        id,
        email: email.toLowerCase(),
        emailVerified,
        username,
        password: hashedPassword,
        role: constants.USER_ROLES.USER,
        name
```

```
    }
    await db.user.create(user)
    return user
}

exports.signIn = () => {

}

exports.getByEmail = async (email) => {
    const result = await db.user.findOne({ where: { email: email.toLowerCase() } })
    if (!result) {
        return null
    }
    return result
}

exports.getByUsername = async (username) => {
    const result = await db.user.findOne({ where: { username:
username.toLowerCase() } })
    if (!result) {
        return null
    }
    return result
}

exports.updateUsername = async (userId, username) => {
    await db.user.update({username}, {where: {id: userId}})
}


exports.getByEmailOrUsername = async (emailOrUsername) => {
    const result = await db.user.findAll({
        where: {
            email: emailOrUsername.toLowerCase(),
            username: emailOrUsername.toLowerCase(),
        }
    })
    if (!result[0]) {
        return null
    }
    return result[0]
}

exports.getByName = async (name) => {
    const result = await db.user.findOne({ where: { name } })
    if (!result) {
        return null
    }
    return result
}
```

```
exports.updateUserFields = async (userId, fields) => {
    const {name, image} = fields

    const context = {}
    if (name) context.name = name
    if (image) context.image = image

    await db.user.update(context, {where: {id: userId}})
    return await exports.getById(userId)
}

exports.getById = async (id) => {
    const result = await db.user.findByPk(id)
    if (!result) {
        return null
    }
    return result
}

exports.deleteById = async (userId) => {
    return await db.user.destroy({ where: {id: userId} })
}

exports.getAll = async () => {
    return await db.user.findAll()
}

exports.addToFriendsList = async (userId, friendId) => {
    const user = await exports.getById(userId)
    const context = [
        ...user.friends,
        friendId
    ]
    await db.user.update({friends: context}, {where: {id: userId}})
    return context
}
```

dal/dictionaries.js

```
const db = require("../db")
const uuid = require('uuid')


exports.create = async (userId, name) => {
    const id = uuid.v4()
    const dictionary = {
        id,
```

```
        userId,
        name,
    }
    await db.dictionaries.create(dictionary)
    return dictionary
}


exports.getByUserId = async (userId) => {
    const result = await db.dictionaries.findAll({ where: { userId } })
    if (!result) {
        return null
    }
    return result
}

exports.deleteById = async (dictionaryId) => {
    return await db.dictionaries.destroy({ where: {id: dictionaryId} })
}

exports.getById = async (id) => {
    const result = await db.dictionaries.findByPk(id)
    if (!result) {
        return null
    }
    return result
}

exports.copyDictionary = async (userToCopy, dictionaryId) => {
    const dictionary = await exports.getById(dictionaryId)
    return await exports.create(userToCopy, dictionary.name)
}

exports.update = async (id, fields) => {
    return await db.dictionaries.update(fields, {where: {id}})
}
```

dal/entries.js

```
const db = require("../db")
const uuid = require('uuid')


exports.create = async (dictionaryId, key, value) => {
    const id = uuid.v4()
    const entry = {
        id,
        dictionaryId,
        key,
```

```
        value,
    }
    await db.entries.create(entry)

    // dictionary counter update
    const dictionaryLength = (await db.entries.findAll({where:
{dictionaryId}})).length
    await db.dictionaries.update({count: dictionaryLength}, {where: {id:
dictionaryId}})
    return entry
}


exports.getByDictionaryId = async (dictionaryId) => {
    const result = await db.entries.findAll({ where: {dictionaryId} })
    if (!result) {
        return null
    }
    return result
}


exports.deleteAllByDictionaryId = async (dictionaryId) => {
    await db.entries.destroy({ where: {dictionaryId} })
}


exports.deleteById = async (id) => {
    const entry = await db.entries.findByPk(id)
    await db.entries.destroy({ where: {id} })

    // dictionary counter update
    const dictionaryLength = (await db.entries.findAll({where: {dictionaryId:
entry.dictionaryId}})).length
    await db.dictionaries.update({count: dictionaryLength}, {where: {id:
entry.dictionaryId}})

    return id
}
```

http/index.js

```
import axios from 'axios'
import jwt_decode from 'jwt-decode'

const PORT = 8000
const baseURL = `http://localhost:${PORT}/`
const host = axios.create({
    baseURL: baseURL,
```

```
    timeout: 1000,
    headers: {'X-Custom-Header': 'foobar'}
})

const authHost = axios.create({
    // baseURL: `http://192.168.1.113:${PORT}`,
    baseURL: `http://localhost:${PORT}`,
    timeout: 1000,
    headers: {'X-Custom-Header': 'foobar'}
})

const authInterceptor = config => {
    config.headers.authorization = `Bearer ${localStorage.getItem('token')}`
    return config
}
authHost.interceptors.request.use(authInterceptor)


const createOrUpdateDictionary = async (data) => {
    const userId = data.userId
    const body = {
        name: data.name
    }
    if (data.dictionaryId) {
        body.dictionaryId = data.dictionaryId
    }
    return await host.post(`/users/${userId}/dictionaries/`, body)
}

const getDictionaries = async (userId) => {
    return await host.get(`/users/${userId}/dictionaries/`)
}

const deleteDictionary = async (userId, dictionaryId) => {
    return await host.delete(`/users/${userId}/dictionaries/${dictionaryId}/`)
}

const shareDictionary = async (userId, dictionaryId, recipientId) => {
    return await authHost.post(`/users/${userId}/share-dictionary/`, {dictionaryId,
recipientId})
}



const createEntry = async (data) => {
    const {userId, dictionaryId, key, value} = data
    return await
host.post(`/users/${userId}/dictionaries/${dictionaryId}/entries/`, {key, value})
}

const getEntries = async (userId, dictionaryId) => {
```

```
    return await host.get(`/users/${userId}/dictionaries/${dictionaryId}/entries/`)
}

const deleteEntry = async (userId, dictionaryId, entryId) => {
    return await
host.delete(`/users/${userId}/dictionaries/${dictionaryId}/entries/${entryId}/`)
}

const getRandomOne = async (dictionaryId) => {
    return await host.get(`/random/${dictionaryId}`)
}



const registration = async (email, password) => {
    const {data} = await host.post('/users/', {email, password})
    localStorage.setItem('token', data.token)
    return jwt_decode(data.token)
}

const login = async (email, password) => {
    const response = await host.post('/users/login/', {email, password})
    localStorage.setItem('token', response.data.token)
    return {...jwt_decode(response.data.token), userData: response.data.user}
}

const check = async () => {
    const response = await authHost.get('/users/check/' )
    localStorage.setItem('token', response.data.token)
    return {...jwt_decode(response.data.token), userData: response.data.user}
}

const updateProfile = async (userId, fields) => {
    return await authHost.post(`/users/${userId}/update-profile/`, {fields})
}


const addToFriendsByUsername = async (userId, friendId) => {
    return await authHost.post(`/users/${userId}/friends/`, {friendId: friendId})
}

const updateUsername = async (userId, username) => {
    return await authHost.post(`/users/${userId}/username/`, {username: username})
}

const getByIdOrUsername = async (data) => {
    return await authHost.get(`/users/${data}`)
}

const getFriends = async (userId) => {
    return await authHost.get(`/users/${userId}/friends/`)
```

```
}


const subscribeNotifications = async () => {
    const eventSource = new EventSource(`http://localhost:${PORT}/notifications/`)
    eventSource.onmessage = function (event) {
        return JSON.parse(event.data)
    }
}

const sendNotification = async (message, dictionaryId, recipientId, senderImageUrl)
=> {
    return await authHost.post('/notifications/', {
        message, dictionaryId, recipientId, senderImageUrl
    })
}

const sendProfileImage = async (userId, formData) => {
    return await authHost.post(`/users/${userId}/upload-profile-image/`, formData)
}


const importDictionary = async (userId, formData) => {
    return await authHost.post(`/users/${userId}/import-dictionary/`, formData)
}


export {
    baseURL,
    createOrUpdateDictionary,
    getDictionaries,
    deleteDictionary,
    shareDictionary,

    createEntry,
    getEntries,
    deleteEntry,

    registration,
    login,
    getRandomOne,

    check,

    updateProfile,

    addToFriendsByUsername,
    getByIdOrUsername,
    getFriends,

    subscribeNotifications,
```

```
    sendNotification,

    sendProfileImage,
    importDictionary,
    updateUsername,
}
```

store.js

```
import {makeAutoObservable} from 'mobx'

class UserStore {
    constructor() {
        this._isAuth = false
        this._user = {}
        makeAutoObservable(this)
    }


    setIsAuth(bool) {
        this._isAuth = bool
    }
    get isAuth() {
        return this._isAuth
    }


    setUser(user) {
        this._user = user
    }
    updateUserData(toUpdate) {
        this._user = { ...this._user, userData: { ...toUpdate } }
    }
    get user() {
        return this._user
    }
}


export {
    UserStore
}
```

App.js

```
import React, {useContext, useEffect, useState} from 'react';
import {BrowserRouter} from "react-router-dom";
import {observer} from "mobx-react-lite";
import {Spinner} from "react-bootstrap";
import 'react-toastify/dist/ReactToastify.css'

import {check} from "./http";
import {Context} from "./index";

import AppRouter from "./components/AppRouter";
import Navigation from "./components/Navigation";

import './styles/App.css';


const App = observer(() => {
    const {user} = useContext(Context)
    const [loading, setLoading] = useState(true)

    useEffect(() => {
        check().then(data => {
            user.setUser(data)
            user.setIsAuth(true)
        }).finally(() => setLoading(false))
    }, [])

    if (loading) {
        return (
            <div
                className={"d-flex justify-content-center align-items-center "}
                style={{height: window.innerHeight - 12}}
            >
                <Spinner animation={"border"}/>
            </div>
        )
    }

    return (
        <BrowserRouter>
            <Navigation />
            <AppRouter />
        </BrowserRouter>
    )
})

export default App;
```

Naviagation.js

```
import React, {useContext, useEffect, useState} from 'react';
import {useNavigate} from "react-router-dom";
import {observer} from "mobx-react-lite";

import NotificationsNoneIcon from '@mui/icons-material/NotificationsNone';
import CommentIcon from '@mui/icons-material/Comment';

import {baseURL, shareDictionary} from "../http";

import {Context} from "../index";
import {ROUTES} from "../constants";
import {Dropdown} from './Dropdown'

import avatarDefault from "../assets/images/profile-image-default.jpg";
import '../styles/Navigation.css';


const Navigation = observer(() => {
    const navigate = useNavigate()
    const context = useContext(Context)
    const user = context.user.user


    const [notifications, setNotifications] = useState([])
    useEffect(() => {
        subscribe().catch(e => console.log(e))
    }, [])

    const subscribe = async () => {
        const eventSource = new EventSource(`http://localhost:8000/notifications/`)
        eventSource.onmessage = function (event) {
            const data = JSON.parse(event.data) // {dictionaryId, recipientId,
message, id, senderImageUrl}
            const newPost = {
                ...data,
                action: async () => await shareCurrentDictionary(data),
                cancel: () => setNotifications([...notifications.filter(item =>
item.id !== data.id)])
            }
            setNotifications(prev => [newPost, ...prev])
        }
        console.log(notifications)
    }

    const shareCurrentDictionary = async (data) => {
        const {dictionaryId, recipientId, id} = data
        await shareDictionary(user.id, dictionaryId, recipientId)
        setNotifications([...notifications.filter(item => item.id !== id)])
    }

    const accountList = [
```

```
        {message: 'Accounts', action: () => navigate(ROUTES.ACCOUNT)},
        {message: 'Dictionaries', action: () => navigate(ROUTES.DICTIONARIES)},
        {message: 'Settings', action: () => navigate(ROUTES.SETTINGS)},
        {message: 'Log out', action: () => logOut()},
    ]

    const logOut = () => {
        context.user.setUser({})
        context.user.setIsAuth(false)
        navigate(ROUTES.LOGIN)
        localStorage.clear()
    }

    const Navbar = (props) => (
        <nav className="navbar">
            <ul className="navbar-nav">
                {props.children}
            </ul>
        </nav>
    )

    const ProfileImage = () => (
        <img src={`${baseURL + user.userData.image}` || avatarDefault}
             className="navigation-profile-image" alt="profile image"/>
    )

    return (
        <Navbar>
            <a href="/" className="a-logo" onClick={() =>
navigate(ROUTES.DICTIONARIES)}>
                {/*<AccessibleForwardIcon/>DMS*/}
                <CommentIcon fontSize="small"/> DMS
            </a>
            <div className="nav-items-right">
                {context.user.isAuth ?
                    <div className="nav-one-item-right">
                        <li key={notifications.message} className="nav-item">
                            <Dropdown
                                items={notifications}
                                icon={<NotificationsNoneIcon className="icon"/>}
                                className='nav-notification-dropdown'
                                option='notification'
                            />
                        </li>
                        <li key={notifications.message} className="nav-item">
                            <Dropdown
                                items={accountList}
                                icon={<ProfileImage/>}
                                className='nav-image-dropdown'
                            />
                        </li>
```

```
                    </div>
                    :
                    <div>
                        <a className="menu-item" onClick={() =>
navigate(ROUTES.LOGIN)}>
                            Authorisation
                        </a>
                    </div>
                }
            </div>
        </Navbar>
    )
})

export default Navigation;
```

Dictionaries.js

```
import React, {useContext, useEffect, useState} from 'react'
import {useNavigate} from "react-router-dom";
import {observer} from "mobx-react-lite";

import {generateRandomDigit} from "../helpers";
import {ROUTES} from "../constants";
import {Context} from "../index";

import Friends from "../modals/Friends";
import {Dropdown} from '../components/Dropdown'
import {Form, FormInput, FormInputExplanation, FormTitle} from "../lib/Forms";
import {TextButton} from "../lib/Buttons";
import {Icon} from "../lib/Icons";

import {
    createOrUpdateDictionary,
    getDictionaries,
    deleteDictionary,
    importDictionary,
} from '../http'

import '../styles/Dictionaries.css';
import '../styles/Lists.css';


const DictionariesPage = observer(() => {
    const navigate = useNavigate()
    const context = useContext(Context)
    const user = context.user.user

    // Global state
```

```
    const [state, setState] = useState({
        nameOfNew: '',
        nameToEdit: '',
        showModal: false,
        currentItem: null
    })

    // All dictionaries data
    const [data, setData] = useState([])

    const dropdownListFunc = (item) => [
        {message: 'Edit', action: () => setEdit(item.id)},
        {message: 'Share', action: () => setState({...state, showModal: true,
currentItem: item})},
        {message: 'Delete', action: () => deleteCurrentDictionary(item.id)},
    ]

    useEffect(() => {
        updateData()
    }, [])

    const updateData = () => {
        getDictionaries(user.id).then((response) => {
            let dictionaries = response.data.reverse()

            // add edit, icon rows for every
            for (const entity of dictionaries) {
                entity.edit = false
                entity.iconIndex = generateRandomDigit()
            }
            setData(dictionaries)
        })
    }

    const newDictionary = async () => {
        const newDictionary = await createOrUpdateDictionary({userId: user.id,
name: state.nameOfNew})
        setData(prevState => [newDictionary.data, ...prevState])
        setState({...state, nameOfNew: ''})
    }

    const openDictionary = async (dictionaryId) => {
        navigate(ROUTES.ENTRIES + '/' + dictionaryId)
    }

    const deleteCurrentDictionary = async (dictionaryId) => {
        await deleteDictionary(user.id, dictionaryId).catch(e => console.log(e))
        setData([...data.filter(item => item.id !== dictionaryId)])
    }

    const updateCurrentDictionary = async (dictionaryId, name) => {
```

```
        await createOrUpdateDictionary({
            userId: user.id,
            dictionaryId,
            name: name
        })

        for (const entity of data) {
            if (entity.id === dictionaryId) {
                entity.name = name
            }
        }

        // close
        setEdit(dictionaryId, false)
        setState({...state, nameToEdit: ''})
    }

    const setEdit = (currentId, toOpen = true) => {
        let withEditTrueForCurrent = []
        for (const entity of data) {
            if (entity.id === currentId) {
                if (toOpen) {
                    setState({...state, nameToEdit: entity.name})
                    entity.edit = true
                } else {
                    entity.edit = false
                }
            }
            withEditTrueForCurrent.push(entity)
        }
        setData(withEditTrueForCurrent)
    }

    const importNewDictionary = async (userId, formData) => {
        await importDictionary(userId, formData)
        // update state
    }

    return (
        <div className="dictionary-container">
            <div className="dictionary-position-container">
                <div style={{display: 'flex'}}>
                    <Form style={{marginTop: 6, width: '100%'}}>
                        <FormTitle text="Name your future masterpiece"/>
                        <FormInput
                            variant='space-left'
                            value={state.nameOfNew}
                            onChange={e => setState({...state, nameOfNew:
e.target.value})}
                        >
                            <label htmlFor="select-file">
```

```
                                <span style={{marginLeft: 10, marginTop: 0, cursor:
"pointer"}}
                                      data-tootik-conf='left'
                                      data-tootik="Tap to import
dictionary">&#x1f914;
                                </span>
                                <input type="file" accept=".txt" id="select-file"
style={{display: "none"}}
                                       onChange={async event => {
                                           const formData = new FormData()
                                           formData.append("text-file",
event.target.files[0])
                                           await importNewDictionary(user.id,
formData)
                                       }}
                                />
                            </label>
                        </FormInput>
                        <div style={{display:"flex"}}>
                            <FormInputExplanation
                                text="That name can be edited in future, you can
always delete your dictionary"/>
                            <div className="dictionary-sort-button-div">
                                <a onClick={() => {
                                    const sortedData = data.sort((a, b) => a.name <
b.name ? -1 : 1)
                                    setData([...sortedData])
                                }}>
                                    <span className="dictionary-sort-button-span">→
Sort by name</span>
                                </a>
                            </div>
                        </div>

                    </Form>
                    <TextButton style={{marginTop: 25}} onClick={() =>
newDictionary()} text="New"/>
                </div>

                <div className="dictionary-list-div">
                    {data.map((item) =>
                        <>
                            <div className="list-item-div">
                                <div className="list-edit-form-general-div">
                                    {item.edit ?
                                        <div key={item.id} className="list-edit-
form-div">
                                            <Icon icon={item.iconIndex}
style={{marginRight: 6}}/>

                                            <Form>
```

```
                                        <FormInput
                                            style={{marginLeft: 10}}
                                            value={state.nameToEdit}
                                            onChange={e =>
setState({...state, nameToEdit: e.target.value})}
                                        />
                                    </Form>
                                    <TextButton
                                        onClick={() =>
updateCurrentDictionary(item.id, state.nameToEdit)}
                                        text="Update"
                                    />
                                    <TextButton
                                        variant='cancel'
                                        onClick={() => setEdit(item.id,
false)}
                                        text="Cancel"
                                    />
                                </div>
                                :
                                <div className="list-edit-form-div">
                                    <Icon icon={item.iconIndex}
style={{marginRight: 6}}/>
                                    <a key={item.id} className="list-item-
a">
                                        <span onClick={() =>
openDictionary(item.id)}>
                                            {item.name}
                                        </span>
                                    </a>
                                </div>
                            }
                        </div>
                        <div className="dictionaries-right">
                            <span className="list-counter">
                                {item.count} items
                            </span>
                            <Dropdown
                                className='item-dropdown' icon='Options'
                                items={dropdownListFunc(item)}
                            />
                        </div>
                    </div>

                    <Friends
                        item={state.currentItem}
                        show={state.showModal}
                        onHide={() => setState({...state, showModal:
false})}
                    />
                </>
```

```
                    )}
                </div>

            </div>
        </div>
    )
})

export default DictionariesPage
```