ДЕРЖАВНИЙ УНІВЕРСИТЕТ ТЕЛЕКОМУНІКАЦІЙ

НАВЧАЛЬНО–НАУКОВИЙ ІНСТИТУТ ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ

Кафедра інженерії програмного забезпечення

Пояснювальна записка

до бакалаврської роботи
на ступінь вищої освіти бакалавр

на тему: «**РОЗРОБКА WEB-ДОДАТКУ САМОМЕНЕДЖМЕНТУ ЗАСОБАМИ REACT.JS, NODE.JS**»

Виконав: студент 4 курсу, групи ПД–44
спеціальності
121 Інженерія програмного забезпечення
(шифр і назва спеціальності/спеціалізації)

Приходько Д.А.
(прізвище та ініціали)

Керівник Золотухіна О.А.
(прізвище та ініціали)

Рецензент _____
(прізвище та ініціали)

Київ –2022

ДЕРЖАВНИЙ УНІВЕРСИТЕТ ТЕЛЕКОМУНІКАЦІЙ
НАВЧАЛЬНО-НАУКОВИЙ ІНСТИТУТ ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ

Кафедра Інженерії програмного забезпечення
Ступінь вищої освіти -«Бакалавр»
Спеціальність підготовки – 121 «Інженерія програмного забезпечення»

ЗАТВЕРДЖУЮ
Завідувач кафедри
Інженерії програмного
забезпечення
_____Негоденко О.В.
"____" _____2022 року

З А В Д А Н Н Я
НА БАКАЛАВРСЬКУ РОБОТУ СТУДЕНТА

ПРИХОДЬКА ДМИТРІЯ АНАТОЛІЙОВИЧА
(прізвище, ім'я, по батькові)

1. Тема роботи: «Розробка web-додатку самоменеджменту засобами React.js, Node.js»

Керівник роботи: Золотухіна О.А., к.т.н., доцент
(прізвище, ім'я, по батькові, науковий ступінь, вчене звання)

Затверджені наказом вищого навчального закладу від «18» лютого 2022 року №__.
2. Строк подання студентом роботи «3» червня 2022 року
3. Вхідні дані до роботи
Наукова література з дисципліни самоменеджменту та дослідження існуючих додатків для самоменеджменту;

4. Зміст розрахунково-пояснювальної записки(перелік питань, які потрібно розробити).
4.1. Дослідження існуючих додатккв для самоменеджменту та .
4.2. Розробка стркутури додатку для самоменеджменту з метою підняття продуктивності в менеджменті та управлінні часом та життєвими цілями.
4.3. Написання коду додатку.
4.4. Висновки.

5. Перелік демонстраційного матеріалу (назва основних слайдів)

1. Аналоги;
2. Таблиця порівнянь аналогів;
3. Мета, об’єкт та предмет дослідження;
4. Завдання бакалаврської роботи;
5. Програмні засоби реалізації;
6. Діаграма прецедентів;
7. Схема бази даних;
8. Діаграма класів;
9. Розробка форми авторизації;
10. Основні форми взаємодії з користувачем;
11. Головна сторінка;
12. Апробація результатів дослідження;
13. Висновки;

6. Дата видачі завдання «11» квітня 2022р.

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів бакалаврської роботи	Строк виконання етапів роботи	Примітка
1	Підбір науково-технічної літератури	11.04.22 - 12.04.22	Виконано
2	Аналіз та дослідження існуючих аналогів	13.04.22 - 14.04.22	Виконано
3	Проектування архітектури програмного засобу	15.04.22 - 16.04.22	Виконано
4	Моделювання об’єкту проектування	17.04.22 - 18.04.22	Виконано
5	Вступ, висновки, реферат	13.04.22 – 14.04.22	Виконано
6	1 розділ	13.04.22 – 15.04.22	Виконано
7	2 розділ	18.04.22 – 28.04.22	Виконано
8	Розробка функціоналу додатку для самоменеджменту	18.04.22 -	
9	3 розділ		
10	Розробка обов‘язкових демонстраційних матеріалів		
11	Попередній захист роботи		

12	Здача роботи	3.06.22	

Студент _____
(підпис) (прізвище та ініціали)

Керівник роботи _____ _____
(підпис) (прізвище та ініціали)

РЕФЕРАТ

Текстова частина бакалаврської роботи 75 с., 58 рис., 9 джерел.

Об’єкт дослідження – процеси самоменеджменту.

Предмет дослідження – додатки для самоменеджменту.

Мета роботи – налагодження та спрощення процесів управління власним часом та ресурсами за рахунок впровадження програмного додатку самомененджменту.

Основні задачі:

- Аналіз існуючих додатків самоменеджменту, з метою виявлення недоліків та дослідженням існуючих патернів побудови користувацьких інтерфейсів за цим напрямком;
- Розробка архітектури додатку для самоменеджменту;
- Розробка архітектури бази даних;
- Розробка серверної частини за “багато-слойоною” архітектурою;
- Розробка архітектури клієнтської частини на основі компонентного підходу;

Основні результати:

- Вивчені додатки самоменеджменту, а також їх переваги та недоліки;
- Розроблено діаграму класів, схему бази даних та діаграму прецентів;
- Розроблено основні модулі додатку.

Для розробки використовувалася мова програмування TypeScript, для клієнтської частини було використано бібліотеку React.js, сервер було реалізовано за допомогою платформи Node.js.

ЗМІСТ

ВСТУП

Наразі додатки автоматизації самоменеджменту мають великий вплив на організацію роботи у великих компаніях та відіграють важливу роль у лайф-менеджменті багатьох людей, це зумовлено зручною організацію та менеджментом поставлених цілей для управління командами або власними цілями.

У великих компаніях за допомогою додатків самоменеджменту ведеться налагодження роботи великих груп людей, постановка задач, розподілення між робітниками обов'язків, визначення пріоритетів та важкості поставленої задачі, розбиття задач на певні дати виконання та визначення часу яке на їх виконання необхідно витратити, за допомогою всіх цих інструментів можна швидко ввести робітника в курс справ та надати йому всю необхідну інформацію для виконання поставленої цілі. Також важливим аспектом є швидкий аналіз поточних завдань та визначення пріоритетів за якими виконавцю буде простіше визначити у якому порядку і як виконувати поставлені цілі. Загалом додатки самоменеджменту для роботи в командах значно покращують рівень комунікації між менеджерами та виконавцями.

Додатки самоменеджменту у лайф-менеджменті відіграють важливу роль у постановці власних цілей та визначенні їх актуальності завдяки постійному доступу до перегляду нагальних цілей та веденню статистики виконання і постійному відображенню нагального прогресу, що дозволяє зосередить саме на тих аспектах в житті які на даний момент більше всього потребують цього, також можуть бути використанні як списки з завданнями на день.

Загалом додатки самоменджменту можна впроваджувати у любу сферу діяльності в якій потрібна систематизація та чітка постановка цілі і шляхів її досягнення.

РОЗДІЛ 1. ДОСЛДІЖЕННЯ АКТУАЛЬНОСТІ ДОДАТКІВ САМОМЕНЕДЖМЕНТУ

1.1 Огляд процесів самоменеджменту

Самоменеджмент - це сукупність певних процесів які пов'язані з плануванням, організацією, стимулюванням і контролем людини над власною діяльністю, ці процеси забезпечують її ефективне функціонування в системі управління. Система самоменеджменту ґрунтується на основі таких підходів як :

- Планування власної діяльності або групи людей;
- Реалізація та ітеграція поставлених цілей у зовнішнє оточення;
- Саморозвиток та здобуття нових навичок;
- Самоорганізація;
- Аналіз та моніторинг доцільності поставлених задач;
- Оцінка фінального результату на підставі поставлених цілей та засобів реалізації;
- Стимулювання власної дяльності.

Ефектвність у самоменеджмені здобувається за допомогою таких факторів :

- Скорочення витрат часу на певну операцію або види робіт;
- Планування та реалізація виділеного часу;
- Делегування та перерозподіл робіт;
- Постановка чіткої цілі та способів її досягнення;

Отже самоменеджмент - це один з найважливіших напрямків розвитку людини в управлінській науці, на сам перед передбачає можливість не тільки спланувати та упорядкувати власну діяльність, а також надає можливість налагодити процеси управління та реалізації великих груп людей.

1.2 Основні напрямки та концепції самоменеджменту

Насамперед самоменджмент є дуже широкою наукою тому що може здійснювати вплив на безліч аспектів нашого життя та сприяти нашим результатам у досягненні тих чи інших цілей, серед основних напрямків самоменеджменту можна виділити такі як :

- Управління часом(time-management) – формування і розвиток навичок цілеполягання, та управління власним часом та ресурсами;
- Управління саморозвитком – формування та розвиток навичок постійного самовдосконалення;
- Менеджмент ресурсами – формування та вдосконалення навичок управління власними ресурсами для досягнення поставлених цілей;
- Менеджмент справлення враження(impression management) – формування навичок справляти бажане враження на оточуючих себе людей;
- Лайф-менеджмент – передбачає управління різними сферами життя(є одним з найновіших напрямків);
- Кар’єрний менеджмент – формування навичок управління власною кар’єрою;
- Психологічний менеджмент – формування та розвиток навичок готувати себе психологічно до можливих проблем та способів їх вирішення;
- Менеджмент особистих обмежень – формування та розвиток навичок головна суть яких, підвищення впевненості в собі та можливість критично мислити і примати складні для себе рішення;

Всі ці напрямки можна віднести до основних концепцій самоменеджменту таких як :

1. Економія власного часу та ресурсів – основним критерієм концепції є збільшення кількості вільного часу, який здобувається за допомогою самонавчанню раціонального розподілення часу та спрямованій впевненій роботі з оцінкою якості її виконання щодня, до концепції відносять такі напрямки:
   - Управління часом;
   - Менеджмент ресурсами;
2. Подолання власних обмежень та упереджень – концепція зав’язана на аналізі проблем які перешкоджають повній реалізації власних можливостей, та роботи над їх вирішенням, основні обмеження що заважають саморозвитку : нечіткі цілі, відсутність або зупинка саморозвитку, низькі показники навичок вирішення проблем, низький рівень впливу на людей, основні напрмяки концепції:
   - Психологічний менеджмент;
   - Менеджмент особистих обмежень;
3. Саморозвиток та визначення себе як творчої особистості – концепція звертає увагу на характеристики сучасних менеджерів, їх творчий характер, можливість до саморозвитку та самореалізації в одному напрямку, а частіше навіть у декількох, основні напрямки :
   - Управління саморозвитком;
4. Підвищення культури ведення ділового життя – ідеологія цієї концепції знаходиться в балансі та усвідомленні як організувати власне життя та відпочинок і водночас підтримувати культуру ділового життя (проводити заходи та вміти їх організувати, бути більш уважним до інших людей) основні напрямки :

- Кар’єрний менеджмент;
- Менеджмент справлення враження;
- Лайф-менеджмент;

1.3 Функції самоменеджменту

Самоменеджмент як управлінська дисципліна виконує ряд певних функцій які знаходяться в залежності між собою та виконуються у послідовності :

1. Визначення та встановлення цілей – дослідження мети, аналіз ситуації, визначення методів та стратегій досягнення мети;
2. Планування – планування послідовності задач та розподіл часу на виконання кожної з них, документування отриманого плану;
3. Ухвалення рішень – установка пріоритетів та делегування повноважень;
4. Організація та реалізація – оформлення робочих планів, та документування результату з витраченим на нього часом;
5. Контроль – контролювання процессу виконання поставлених задач, та аналіз результатів за певні проміжки часу;
6. Інформація та комунікація – більш відноситься до групового менеджменту, полягає в веденні комунікації між діючими особами, та обговоренню нагального результату в порівнянні з бажаним;

1.4 Огляд додатків самоменеджменту

1.4.1 Jira

Atlassian JIRA — система відстеження помилок, призначена для організації спілкування з користувачами, і для управління проєктами. Розроблена компанією Atlassian в 2002 році. Доступна в двох версіях: «хмарній» і серверній.

На даний момент Jira є однією най успішніших та популярних систем самоенеджменту, для великих компаній та проектів, в чому ж її переваги і чому великі компанії обирають саме Jira :

1. Jira має дуже високий рівень захищеності та аутентифікації, це досягається за допомогою високого рівня доступу – тільки адміністратор робочого простору може надавати змогу долучитись новим користувачам до проектів, також адмін має доступ перегляду історії усіх змін паролів та логінів користувачі в його робочому просторі, що надає змогу зручніше протидіяти;
2. Повну інтеграцію з такими системами як Confluence, BitBucket та Slack так як є дочірніми проектами 1-ї компанії Atlasian, це сприяє більш чіткій системі контролю версій та комунікативності;
3. Неймовірно велику кількість тулів для обробки та перегляду даних що надає змогу користувачу обирати найзручніший для нього спосіб роботи;

Це все робить Jira одним з найкращих додатків для самоменеджменту та менеджменту групами, але як і в любій іншій системі в неї недоліки :

1. Складний процес налаштування робочого простору – займає багато часу і в контексті особистого самоменджменту в цьому немає необхідності;

2. Більш спрямований на автоматизацію та підтримку великих проектів та груп людей – більшість з функціоналу не буде використовуватись і буде тільки заважати сконцентруватись на основних цілях;
3. Доволі складний UI на вивчення якого та зручне користування доведеться витратити багато часу;

Наведення прикладів відображення даних у додатку Jira :

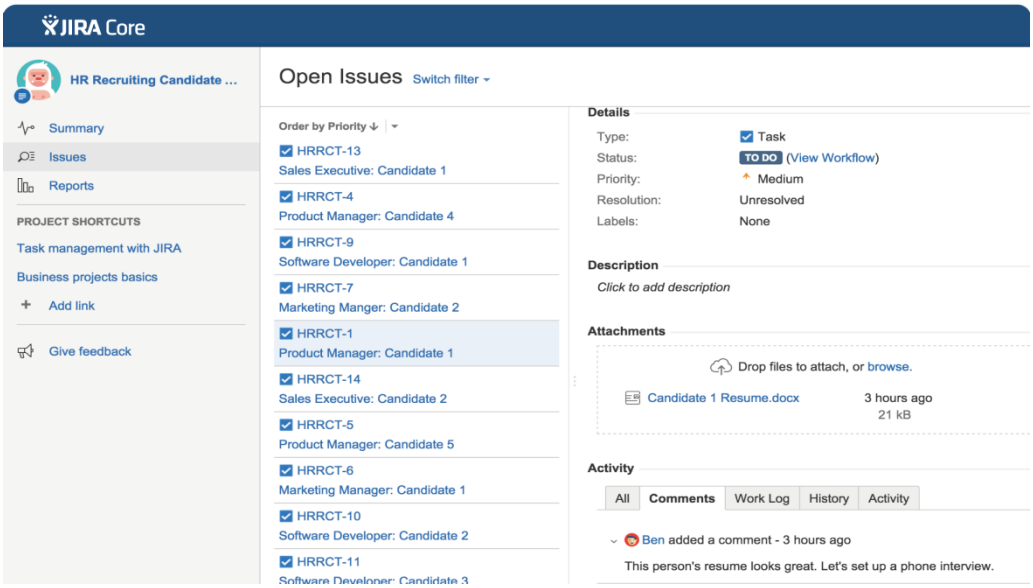


Рисунок 1.1 – Список наявних завдань та форма редагування.

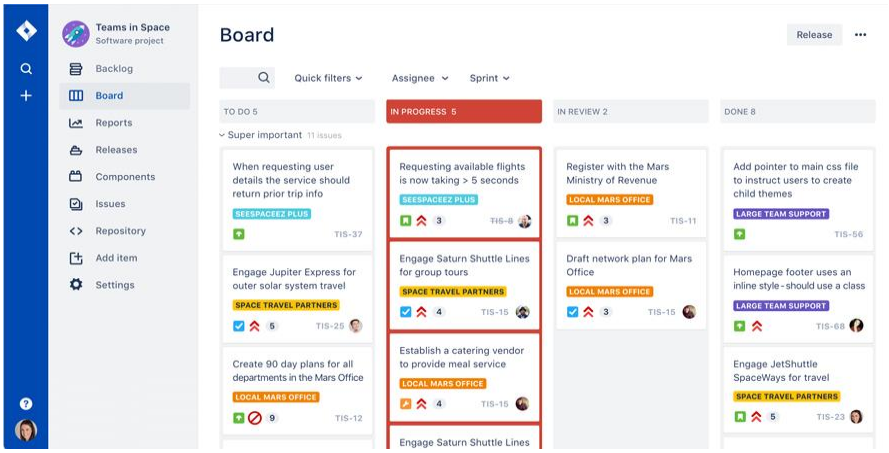


Рисунок 1.2 – Kanban дошка у додатку Jira для відображення завдань.

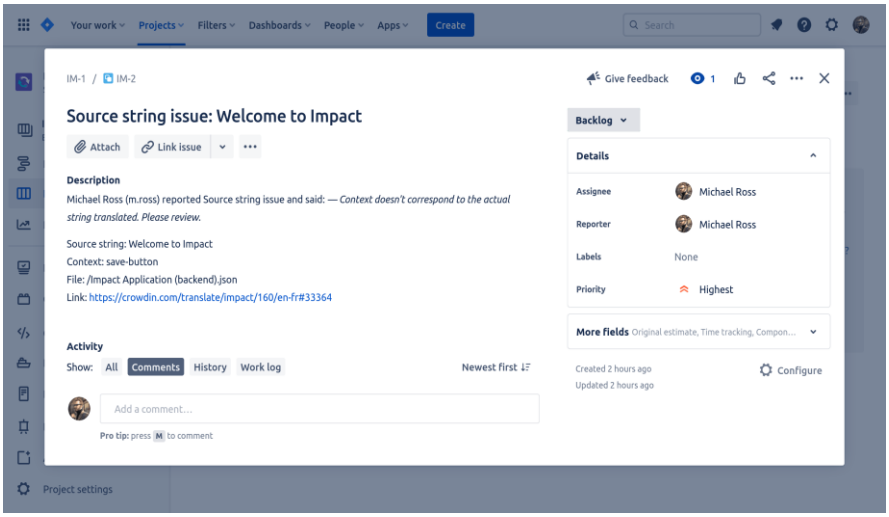


Рисунок 1.3 – Модальне вікно для перегляду більш детальної інформації про завдання у Jira.

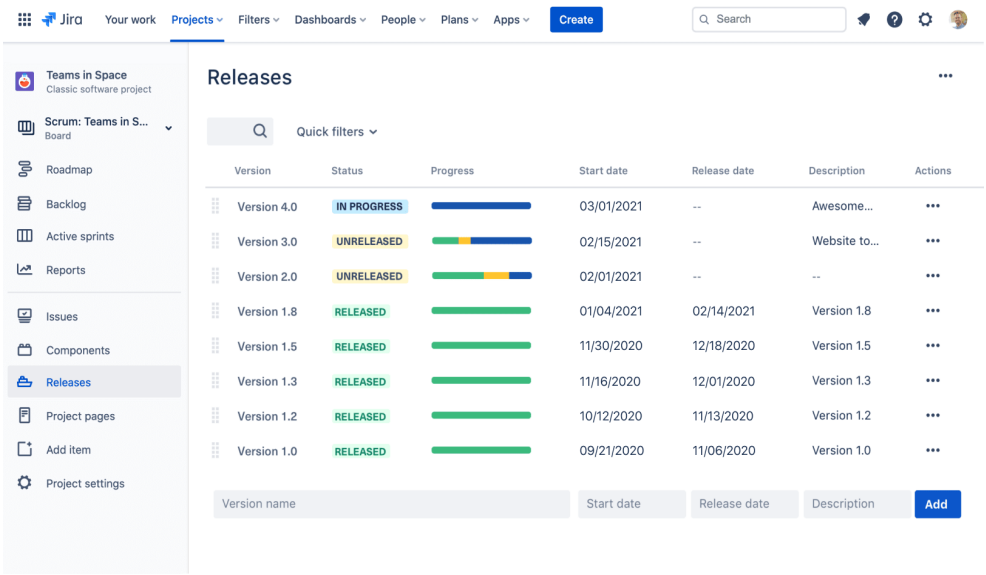


Рисунок 1.4 – Сторінка перегляду версій релізів.

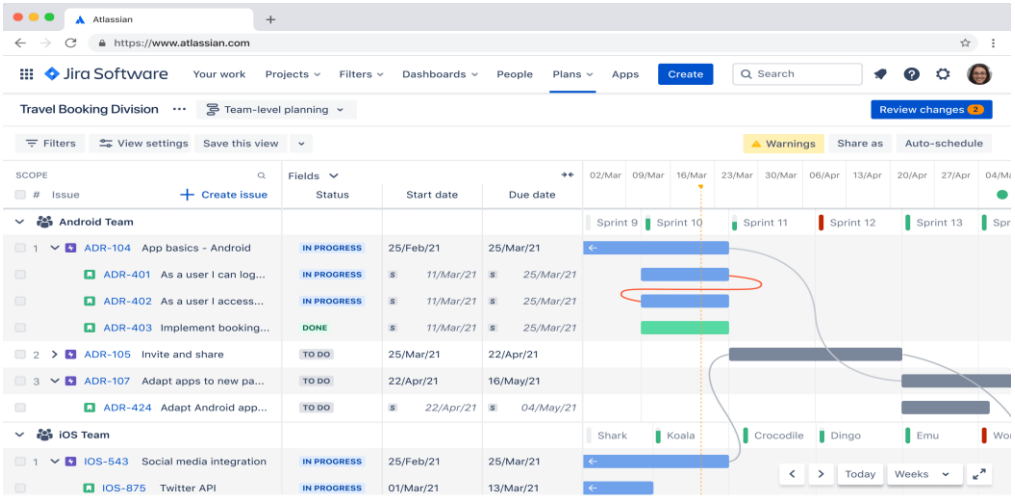


Рисунок 1.5 – Графік залежності(Gantt chart) поставлених цілей.

Рисунок 1.6 – Сторінка графіків продуктивності та тайм-лайну.

1.4.2 Trello

Trello - безплатна багатоплатформна система управління проєктами, розроблена Trello Enterprise, дочірньою компанією Atlassian. Створена у 2011 році компанією Fog Creek Software для створення окремої компанії в Нью-Йорку у 2014 році й продана Atlassian в січні 2017 року.

Trello на відміну від Jira є більш простим додатком для самоменеджменту, але не менш популярним частіше використовується в

невеличких проектах та командах, або навіть для індивідуального самоменеджменту.

Головні відмінності між Jira та Trello :

1. Більш простий інтерфейс та взаємодія з ним;
2. Слабша система безпеки але більш швидке налаштування інфраструктури;
3. Менше часу потрібно на освоєння додатку;
4. Єдиний формат відображення даних Kanban, хоч цей формат і є зараз ідеалом, але все ж інколи є необхідність репрезентувати данні більш розгорнуто;
5. Менша кількість функціоналу;
6. Можливий для використання в цілях особистого самоменеджменту;

Приклади відображення даних у Trello:

Рисунок 1.7 – Kanban дошка для відображення завдань у Trello

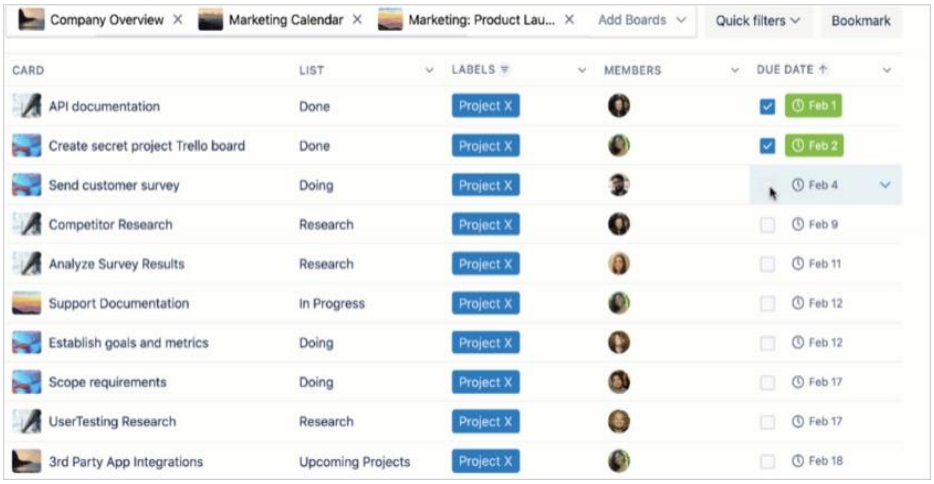


Рисунок 1.8 – Менеджмент робочим простором

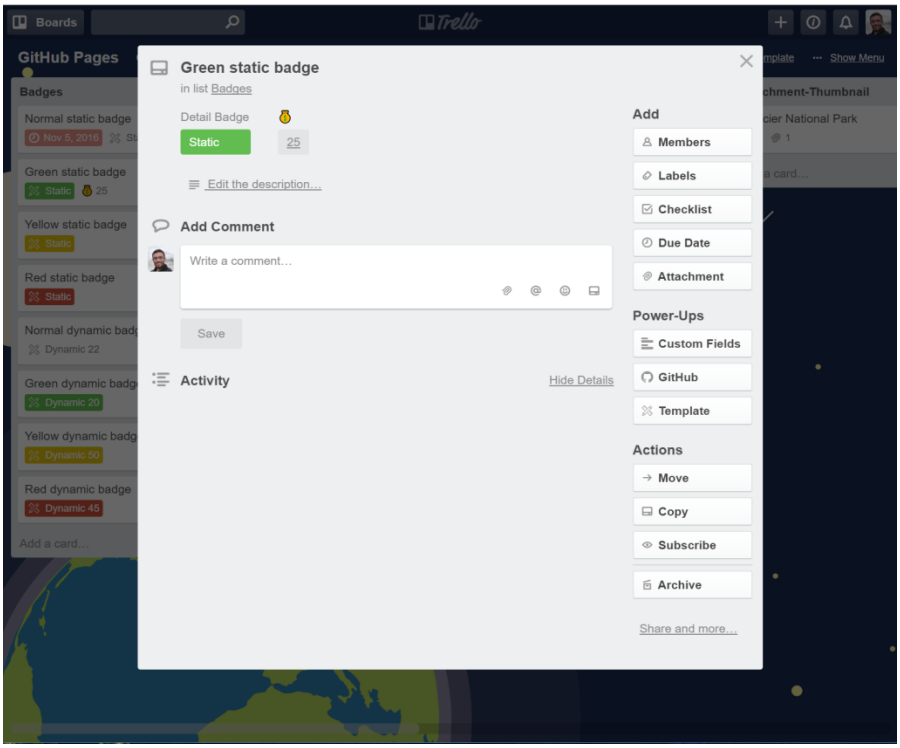


Рисунок 1.9 – Модальне вікно для перегляду більш детальної інформації про завдання у Trello.

1.4.3 LifeWheel

LifeWheel - один з додатків самоменеджменту який використовує підхід лайф-менеджменту і спрямований на самореалізацію та покращення власних показників у обраних користувачем сферах життя, має такі можливості :

1. Заповнення базових сфер життя в яких розвиток людини є необхідним для її самореалізації та введення додаткових сфер на власний розсуд, відображення даних у вигляді графіку;
2. Створення щоденних або разових цілей;
3. Відображення цілей у вигляді списку;
4. Також є посилання на ресурси з порадами як покращити свій самоменеджмент;

З мінусів можна виділити можливість користуватись лише з мобільного телефону, доступний лише на IOS плаформі, більша частина контенту платна;

Наведення скріншотів сторінок з додатку :

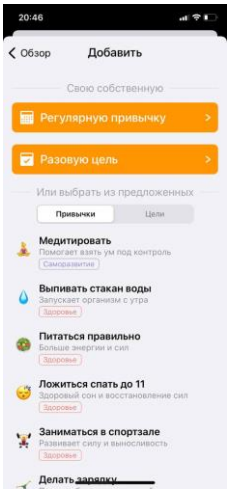


Рис. 1.10 – Створення цілі

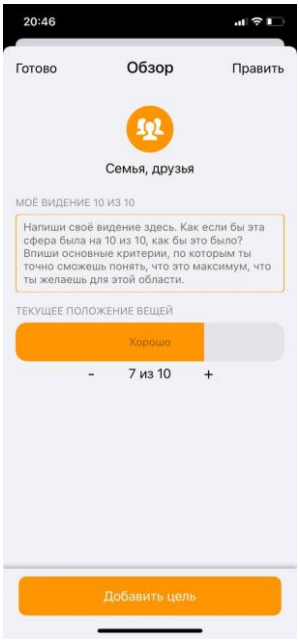


Рис. 1.11 – перерозподіл балів у сферах життя

Рис. 1.12 – головна сторінка

1.5 Wheel-of-life (колесо життєвого балансу) як сучасний спосіб самоменеджменту

Колесо життєвого балансу – це коло поділене на певну кількість частин, кожна частина якого відповідає за певну сферу в житті, в класичному варіанті частин 8, але проаналізувавши власну діяльність можна додати ще декілька. Для того що заповнити власне колесо життя, потрібно взяти кожну сферу життя окремо та об’єктивно визначити ступінь задоволеності нею, далі обрати максимальну відмітку задоволеності, і виділити початкові бали кожній сфері на підставі раніше проведеного аналізу та ступені задоволеності відносно максимального балу.

Після отримання початкового графіку, потрібно проаналізувати чого не вистачає для повного задоволення сферою, та створити цілі які при їх успішному виконанні, підвищать ваш рівень в відповідній сфері, таким чином завжди можна бачити реальний результат та спрямовувати свій фокус на вирішення нагальних проблем.

Коло має оновлюватись кожні пів-року або рік з метою оцінки своїх цілей та перерозподілу фокусу на вирішення саме тих проблем які мають вищій пріоритет у даний момент часу.

1.6 Постановка задач дипломної роботи

Після проведеного аналізу додатків аналогів та дослідження напрямку самоменеджменту – лайф-менеджмент було виділено основні задачі бакалаврської роботи :

1. Розробка архітектури додатку для самоменеджменту;

2. Розробка архітектури бази даних;
3. Розробка серверної частини за “багато-слойоною” архітектурою;
4. Розробка архітектури клієнтської частини на основі компонентного підходу;
5. Реалізація головних модулів додатку :
    - Авторизація користувача;
    - Створення початкового робочого простору для подальшого його використання;
    - Додавання нових цілей та редагування вже існуючих;
    - Можливість перегляду даних у декількох видах відображення;
    - Відображення поточного результату діяльності користувача у обраних сферах;

РОЗДІЛ 2. СТРУКТУРА ДОТАТКУ ДЛЯ САМОЕНЕДЖМЕНТУ ЗА ДОПОМОГОЮ БІБЛІОТКИ REACT.JS ТА NODE.JS

2.1 Про React.js

React — відкрита JavaScript бібліотека для створення інтерфейсів користувача, розроблена з метою вирішення проблем оновлення вмісту на односторінкових веб-додатках, випущена компанією Facebook у 2013році, але вийшов на пік популярності лише у 2020 з введенням можливості функціонального підходу.

Наразі React є топ-1 фреймворком для розробки SPA(Single page application) з декількох причин:

1. Велике ком'юніті – величезна купа статей та форумів які допомагають як при вивченні фреймворку так і при комерційній розробці, значно пришвидшує пошуку оптимального вирішення проблеми також має величезну базу готових компонентів одного типу для вирішення різних задач.
2. Постійний вихід оновлень з доданням нових можливостей та виправленням старих проблем;
3. Має гарну та швидку інтеграцію з багатьма іншими фреймворками для веб розробки;
4. Поріг входження набагато нижчий ніж в інших фреймворках або бібліотеках, здобувається за допомогою меншої кількості окремих директив на вивчення яких потрібно витратити багато часу також проста система об'єднання логіки та розмітки, що одночасно є і проблемою і перевагою тому, що не досвідченій людині набагато простіше зробити не якісний код, а в досвідченої навпаки займе менше часу на розробку ніж за допомогою інших

фреймворків, також не обов’язкова інтеграція TypeScript до проекту, що дозволяє починаючим програмістам сконцентруватись саме на вивченні самого фреймворку на початку, а потім вже підвищувати свою кваліфікацію вивченням TypeScript;

5. Підтримка мультипарадигменості, що дозволяє використовувати кращі практики як з об’єктно орієнтованого підходу так і функціонального;
6. Велика варіативність у підходах організації коду та архітектури, що робить процес розробки дуже гнучким;
7. Компонентний підхід – архітектура будується за допомогою композиції а не наслідування, що також робить процес розробки більш гнучким, швидким та елегантним;

Принци роботи React та його інструменти:

Virtual DOM – Це легке уявлення JavaScript об’єктної моделі документа, головна перевага над звичайним DOM(Document Object Model) заключається в тому, що звичайний DOM оновлюється кожного разу після внесення в нього змін, наприклад ми додали новий товар у до списку, звичайний DOM прийме оновлений список товарів і заново їх відмалює в той час як Virtual DOM це легка копія нашого основного DOM, він проаналізує який саме елемент було змінено або додано і перемалює або домалює лише те що необхідно, це здобувається за допомогою observable підходу, коли кожен компонент у нашому додатку має свій стан і при зміні значення або ссилки якогось з його параметрів наш Virtual DOM розуміє, що потрібно заново відмалювати всі компоненти, що використовують цей параметр , в той же час залишити не змінним ті компоненти на які зміна параметру немає ніякого впливу;

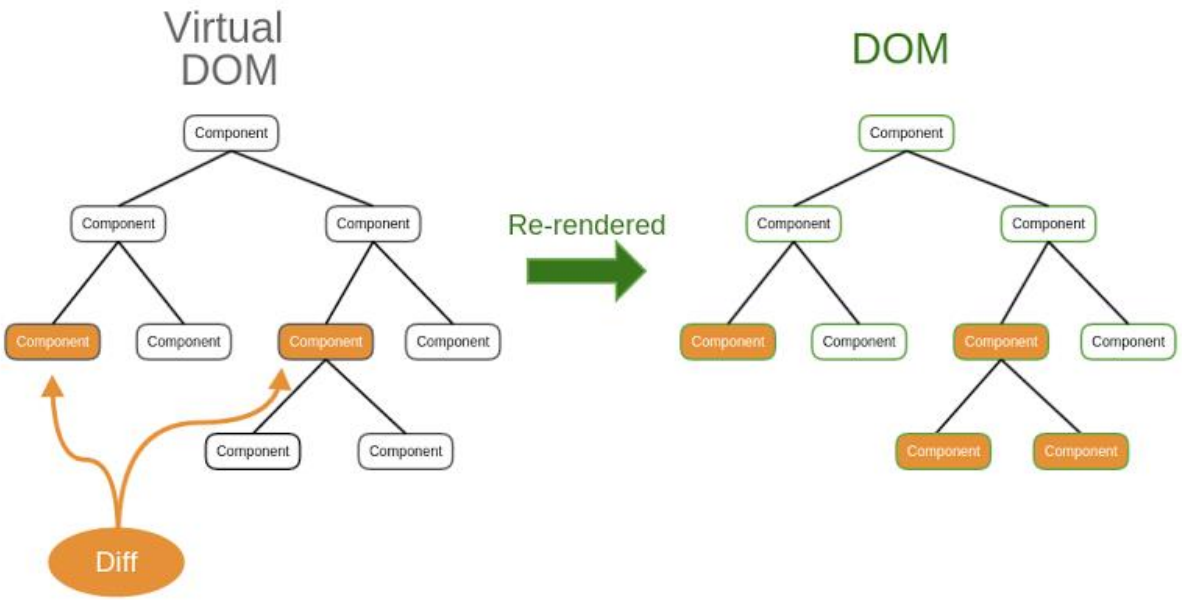


Рисунок 2.1 – Перехід станів дерева компонентів при внесенні змін

Вбудовані методи життєвого, циклу та менеджмент стану компонентів – до 2019 року React мав набагато складнішу архітектуру тому як вимагав лише об’єктно орієнтованого підходу, та використанню великої кількості методів життєвого циклу таких як :

- ComponentDidMount – виклається при першому відображенні компонента і виконує в побічні задачі такі як наприклад отримання необхідної інформації для фкнціонування компонента з сервера;
- ComponentDidCatch – використовується в «компонентах обгортках» для відлову помилок з сервера і відображенню окремої розмітки для випадків коли в компоненті стались помилка;
- ComponentDidUpdate – є схожим з ComponentDidMount, але викликається не при першому відображенні компонента, а коли умова його спрацьовування виконалась, наприклад ми можемо слідкувати за тим як людина пише інформацію в пошукову строку і при зміні її стану запитувати нові дані;

– ComponentWillUnmount – викликається коли компонент зникає з потоку – простіше кажучи більше відображається на інтерфейсі користувача, корисний при видаленні прослуховувачів або підписок які ми декларуємо при спрацьовуванні ComponentDidMount;

```
class FriendStatusWithCounter extends React.Component {
  constructor(props) {
    super(props);
    this.state = { count: 0, isOnline: null };
    this.handleStatusChange = this.handleStatusChange.bind(this);
  }

  componentDidMount() {
    document.title = `Ви натиснули ${this.state.count} разів`;
    ChatAPI.subscribeToFriendStatus(
      this.props.friend.id,
      this.handleStatusChange
    );
  }

  componentDidUpdate() {
    document.title = `Ви натиснули ${this.state.count} разів`;
  }

  componentWillUnmount() {
    ChatAPI.unsubscribeFromFriendStatus(
      this.props.friend.id,
      this.handleStatusChange
    );
  }

  handleStatusChange(status) {
    this.setState({
      isOnline: status.isOnline
    });
  }
  // ...
```

Рисунок 2.2 – Приклад використання методів життєвого циклу

Для менеджменту стану компонентів використовується props це ті властивості які ми можемо передавати дочірньому компоненту з батьківського , state для збереження локального стану компонента та метод setState для його зміни, також має є потужний спосіб обміну даними Context який дозволяє обгорнути одразу декілька компонентів і передати їм однакові властивості замість того щоб передавати в кожен окремо через props.

```
class Example extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      count: 0
    };
  }

  render() {
    return (
      <div>
        <p>Ви натиснули {this.state.count} разів</p>
        <button onClick={() => this.setState({ count: this.state.count + 1 })}>
          Натисни мене
        </button>
      </div>
    );
  }
}
```

Рисунок 2.3 – Приклад використання props, state та setState

Наразі все набагато простіше в функціональному React відсутні методи життєвого циклу та окремий об’єкти state та props на заміну ним прийшли hooks це функції які виконують ті самі задачі, але реалізуються набагато простіше, хоче звісно в цьому є свої проблеми – при кожному новому відображенні даних змінються ссилки на дані тому відбувається більше зайвих рендерів але ці проблеми можна усунути знову ж таки за допомогою інших hooks.

```jsx
import React, { useState } from 'react';

function Example() {
  // Оголошуємо нову змінну стану "count"
  const [count, setCount] = useState(0);

  return (
    <div>
      <p>Ви натиснули {count} разів</p>
      <button onClick={() => setCount(count + 1)}>
        Натисни мене
      </button>
    </div>
  );
}
```

Рисунок 2.4 – Приклад використання хука стану

```jsx
import React, { useState, useEffect } from 'react';

function Example() {
  const [count, setCount] = useState(0);

  // Подібно до componentDidMount та componentDidUpdate:
  useEffect(() => {
    // Оновлюємо заголовок документа, використовуючи API браузера
    document.title = `Ви натиснули ${count} разів`;
  });

  return (
    <div>
      <p>Ви натиснули {count} разів</p>
      <button onClick={() => setCount(count + 1)}>
        Натисни мене
      </button>
    </div>
  );
}
```

Рисунок 2.5 - Приклад використання хука ефекту(життєвого циклу)

2.2 Застосування GraphQL для обміну даними

Мови запитів відіграють одну з найважливіших ролей під час будування веб або десктоп додатків, то му що без наша клієнтська частина не змогла б спілкуватися із сервером, вже довгий час на першому місці знаходиться REST(REpresentational State Transfer – це підхід до організації архітектури мережевих протоколів, які надають доступ до інформаційних ресурсів, описаний і популяризований у 2000 році Роєм Філдінгом, який доречі є одним із творців протоколу HTTP) його головною перевагою є простота в використанні і отримання бажаних даних незалежно від попередніх кроків(в порівнянні з попередніми мовами запитів це був прорив тому, що вони запам'ятовували данні сесії користувача і обробці і відправленні даних покладались на попередні кроки користувача).

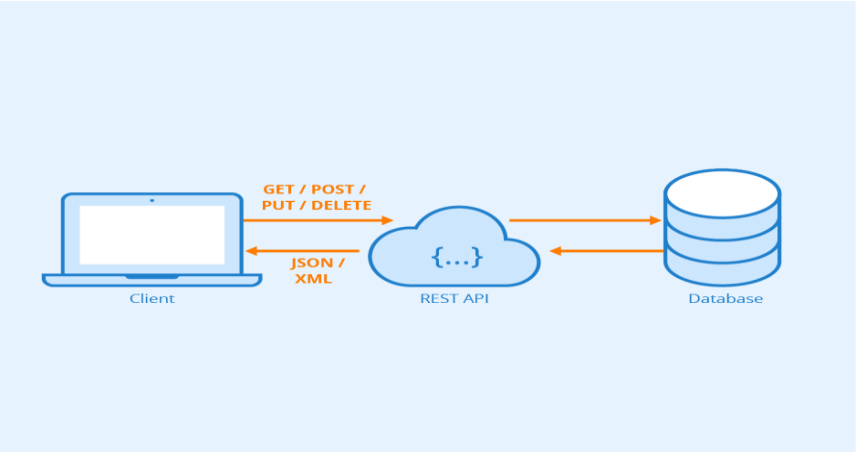


Рисунок 2.6 – Спосіб обміну даними між сервером та клієнтом за допомогою REST

Для того щоб наш REST API працював необхідно мати локальний або віддалений сервер, який буде слухати запити і оброблювати їх відповідно до правил REST таких як :

1. Описати назву endpoint(кінцева точка, за адресою якої буде визначено які саме дані ми хочемо отримати);
2. Метод запиту, їх взагалі багато, але варто знати 4 основних : GET – для отримання даних, POST – відправка, PUT – зміна даних, DELETE – видалення даних;
3. Додання path/query parameters – необхідні для уточнення як саме потрібно обробити дані;

Після обробки сервер здійснить запит до сховища даних знайде необхідні і поверне нам.

Отже на підставі вище описаних правил запит виглядав би так:

path:https://domain_of_api/popular_books?limit=50&skip=0

method: GET

Тут domain_of_api - це домен, де розміщується наш сервер, /popular_books - це наш endpoint, limit та skip це - query параметри які описують, що ми маємо отримати 50 перших книжок.

Приклад отриманих даних (розглянемо на прикладі SPA-Single Page Applications, використовується для складних веб-додатків, за архітектурою є схожим на десктопні додатки)

JSON-формат:

data:{books:[{id:”1”,name:”HarryPotter”,genre:”Fantastic”}, …]}

Далі потрібно серелізувати данні для підготовки до відображення користувачу.

Мінуси такого підходу:

1. Структура даних які ми отримуємо є фіксованою, тобто – в нашому випадку ми отримуємо такі дані про книгу як: id, name та genre, якщо наприклад ми захочемо додати ще поле description нам необхідно буде створити новий endpoint;

2. Якщо наші данні збережено децентралізовано – як приклад зараз часто використовують SLA (Server Less Architecture) тобто данні знаходяться на різних серверах тоді необхідно звернутись до кожного сервера окремо і потім ще додатково оброблювати ці данні;
3. Використання різних баз даних таж сама проблема, що і в 2-му пункті;
4. Немає власного кешування даних як на сервері так і на клієнті, тобто – кожного разу потрібно лізти в базу і діставати дані;

GraphQL – це мова запитів і маніпуляцій з даними для API розроблена та випущена компанією Facebook у 2015 році.

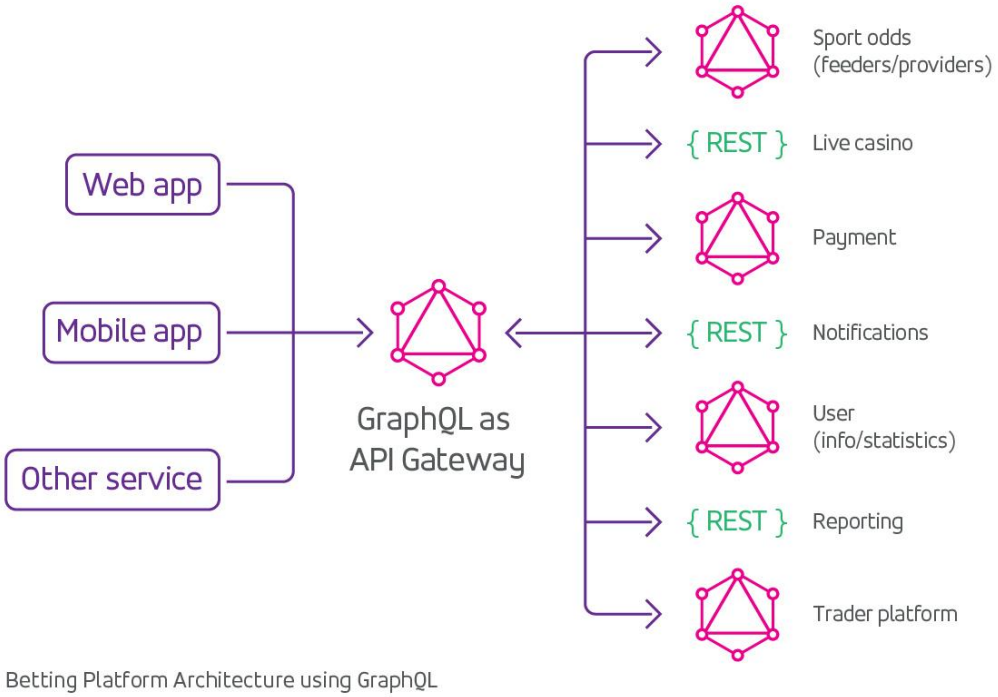


Рисунок 2.7 - Спосіб обміну даними між сервером та клієнтом за допомогою GraphQL

Порівняння інструментів GraphQL та REST :

1. На відміну від REST наші endpoints ніколи не змінюється тобто всі запити завжди йдуть на endpoint назвою - /graphql
2. Також не має різних методів, натомість має 2 типи запитів Query та Mutations, Query – для отримання даних, Mutations для їх зміни, видалення або збереження нових;
3. Schema – це схеми наших даних, наприклад ми можемо створити схему Book для нашого веб-сервісу по скачуванню книжок і описати в ній поля цього об’єкту це ті дані що ми отримували в описі роботи REST;
4. Resolvers – описують яким чином ми маємо отримати дані і звідки;
5. Query – які приходять на зміну різним endpoints в них ми описуємо поля які хочемо отримати на основі наших Schemas і Resolvers які ми будемо використовувати для їх отримання;
6. Apollo Client & Apollo Server – вирішив додати їх сюди так як вони легко інтрегруються з GraphQL і в наших реаліях неможливо уявити одне без одного, якраз вони відповідають за кешування даних на сервері та клієнті;

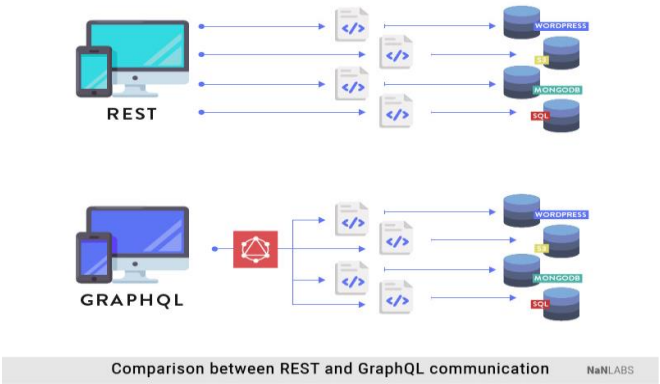


Рисунок 2.8 – Порівняння GraphQL та REST

2.3 Переваги tailwindCSS над іншими CSS фреймворками

Зараз розробку UI в веб-розробці важко уявити без використання CSS-frameworks, вони проносять дуже багато готових рішень та значно скорочують час на стилізацію розмітки, це їх вдається за допомогою :

1. Готові css класи оформлені за методологією BEM – розбиття логіки на маленькі незалежні блоки за допомогою яких можна буду вати складні великі компоненти;
2. Вбудована grid сітка та breakpoints(кінцеві точки при перетині яких css правила можна змінити);
3. Готові компоненти окремо під кожну бібілотеку/фреймворк;
4. Дозволяє дизайнеру та розробнику узгодити правила описання UI на основі документації фреймворку;

Однією з головних особливостей tailwindCSS є можливість динамічно за допомогою серверного JS створювати власні класи, також дуже зручна система використання псевдо селекторів, і велика кількість готових анімацій, що робить процес розробки з tailwindCSS набагато швидшим ніж з іншими фреймворками, окрім цього при створенні робочго розширення видаляє класси які не використовувались щоб зменшити кінцеву вагу та прооптимізувати роботу додатку, опираючись на всі ці фактори було прийняте рішення віддати перевагу саме цьому CSS фреймворку.

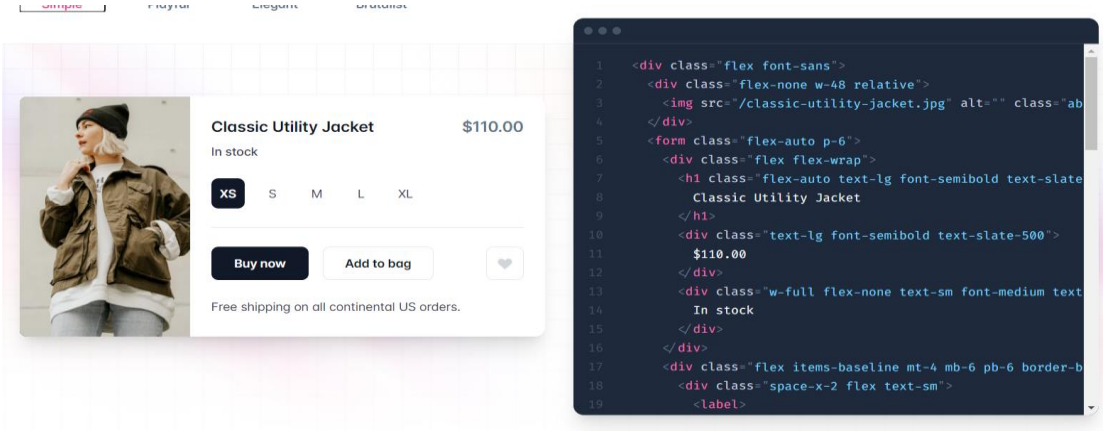


Рисунок 2.9 – Приклад використання tailwindCSS

2.4 Використання TypeScript при розробці веб та десктоп додатків

TypeScript — мова програмування, представлена Microsoft восени 2012; позиціонується як засіб розробки вебзастосунків, що розширює можливості JavaScript. Розробником мови TypeScript є Андерс Гейлсберг, який створив раніше C#, Turbo Pascal і Delphi.

Перш за все варто відмітити, що TypeScript як мова програмування не приносить якихось нових можливостей та не розширює функціонал JavaScript так як в кінцевому результаті компілюється в останній , але попри це є дуже корисним під час розробки завдяки таким властивостям :

1. Статична типізація – допомагає на стадії розробки, в першу чергу завдяки відлову помилок під час компіляції та автокомплітом так як усі наші об’єкти та функції протипізовані і наша IDEA знає точно які параметри та якого типу ми має використати в тій чи іншій функції, або які влативості має приймати наш компонент;
2. Додаткові інструменти розробки такі як interfaces, types, enums та generics, за допомогою них набагато простіше розібратися в чужому коді або провести рефактор власного коду;

Найкраще себе TS показує на великих проектах де є багато розробників, завдяки статичній типізації та компляції, завдяки цьому відсіюється купа помилок на стадії написання коду, та його надалі буде легше підтримувати.

2.5 Розробка схеми бази даних

Першим кроком на початку до написання коду, є проектування та розробка схеми бази даних. Це пов’язано з тим, що залежно від структури даних та способу їх збереження буде будуватись архітектура серверу бо саме він буде займатись обробкою, збереженням та відправкою даних, а вже лише після цього можна буде перейти до розробки клієнтської частини.

Спершу потрібно обрати сутності на основі яких будуть будуватись таблиці та зв’язки між ними, першою й най важливішою сутністю є User, так як усі данні будуть по’вязані з конкретним користувачем далі йде Wheel як сутність яка зберігає результат виконаних завдань та репрезентує данні для аналізу та більш чіткого формулювання цілей. Наступна таблиця Objectives яка зберігає інформацію про цілі користувача, та має певний статус і пріоритет як також розбиті на дві таблиці Status та Priority, також кожна ціль може бути розбита на певні завдання які знаходяться у таблиці Task виконавши які ціль буде досягнена. Замість видалення даних у таблиць Task та Objective є поля archived, це потрібно для більш чіткого аналізу реалізації поставлених цілей та в майбутньому можливості перевести ці данні у структури для відображення на графіках.

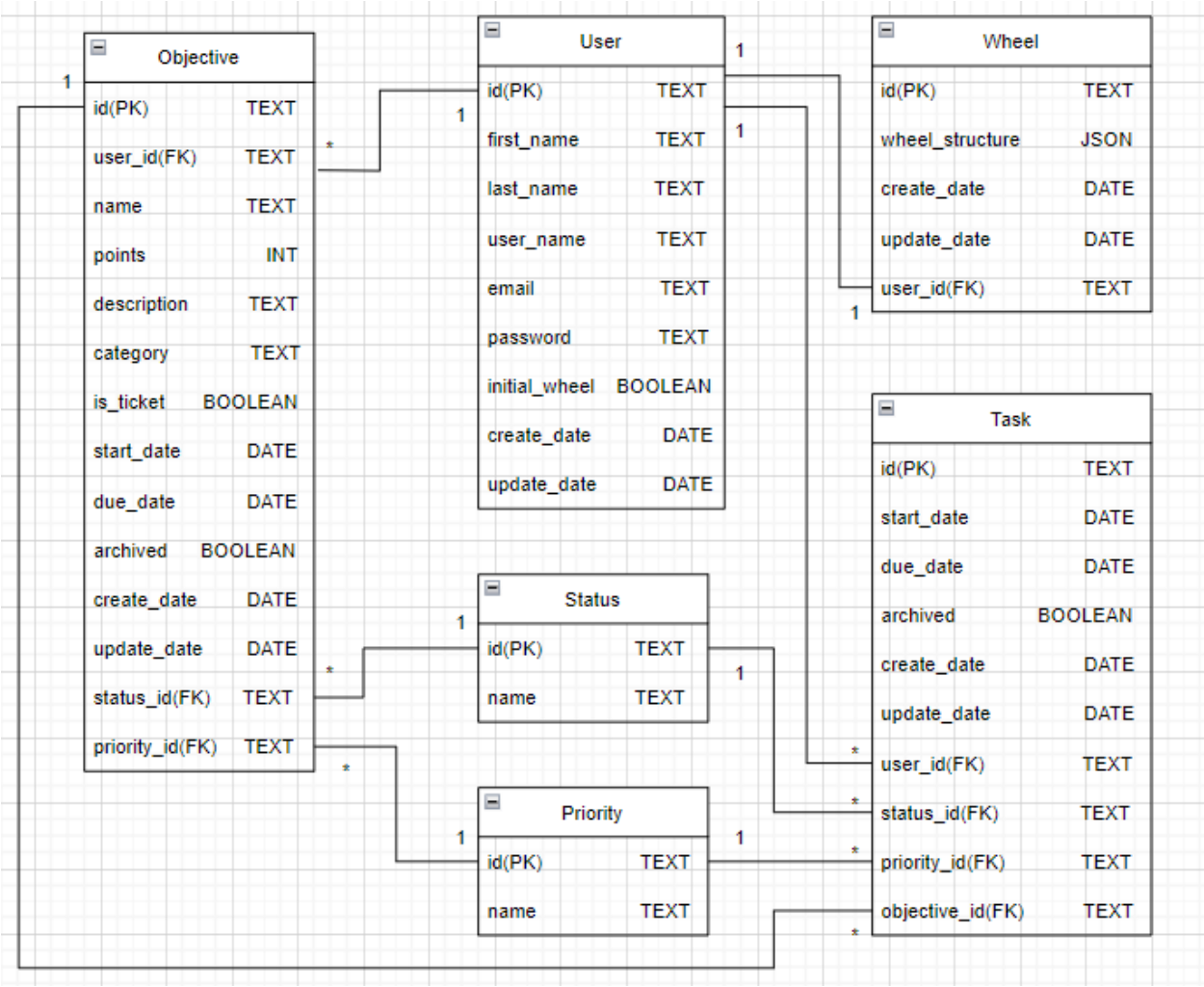


Рисунок 2.10 – Схема бази даних

Після визначення сутностей потрібно зробити зв`язки між ними, оскільки вся взаємодія починається з сутності User , зв’язується таблиця User з такими таблицями :

1. Objectives(цілі) оскільки користувач може їх створювати безліч то зв’язок буде один до багатьох;
2. Wheel яка зберігає в собі стан робочого простору оскільки ожин користувач може мати лише один робочий простір то і зв’язок буде один до одного;
3. Task таблиця яка зберігає задання є подібною до Objectives за виключенням того, що Obectives є батьківською таблицею для

неї, в цьому випадку користувач може також мати безліч завдань тому зв’язок один до багатьох;

Далі йдуть таблиці Task та Objective в них є зв’язки з таблицями Priority, Status та User. Зв’язок з таблицею User був описаний вище в цьому випадку це багато до одного, так само як і Status з Priority, оскільки в одного Task та Objective може бути лише один поточний пріоритет або статус, між собою залежать як Objective один до Task багатьох, оскільки цілі можуть розбиватись на декілька завдань. Типи даних кожної з таблиць наведені на схемі бази даних(див. рисунок – 2.9).

2.6 Діаграма прецедентів

Діаграма прецедентів відображає відношення між акторами та їх взаємодією із системою, в нашому випадку ми маємо 1 користувача(актора) який взаємодіє із свої робочим простором також усі користувачі мають однакові права та функціонал. Першочергово користувач має пройти реєстрацію або логін, якщо він уже зареєстрований, да користувача автоматично переводить до сторінки ініціалізації власного колеса, після ініціалізації користувач матиме змогу використовувати увесь функціонал, також користувач може завершити поточну сесію у своєму браузері.

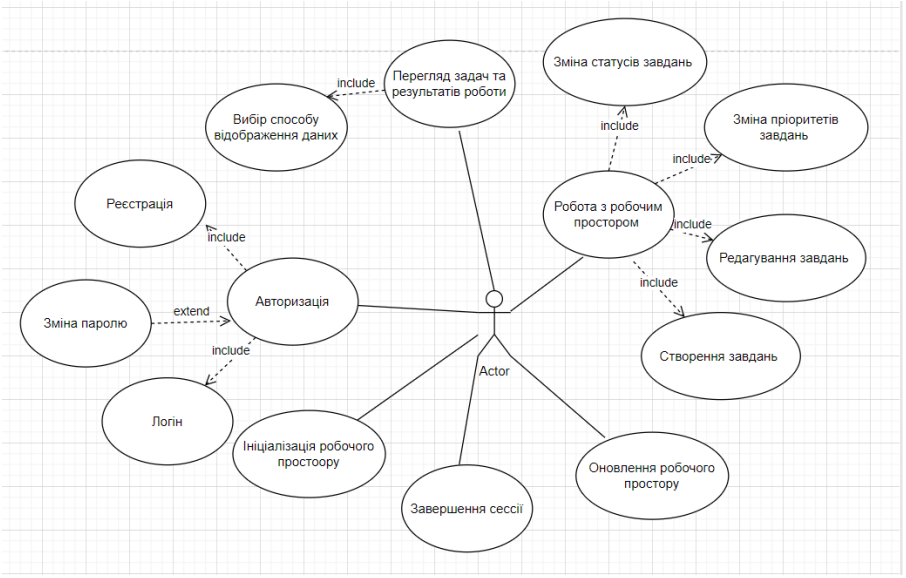


Рисунок 2.11 – Діаграма прецедентів веб-додатку самоменеджменту

2.7 Проектування діаграми класів

Після завершення проектування схеми бази даних, та отримання сутностей з їх властивостями, можна перейти до створення діаграми класів яка відображає наші сутності як об’єкти з певними властивостями та методами або функціями які вони можуть виконувати.

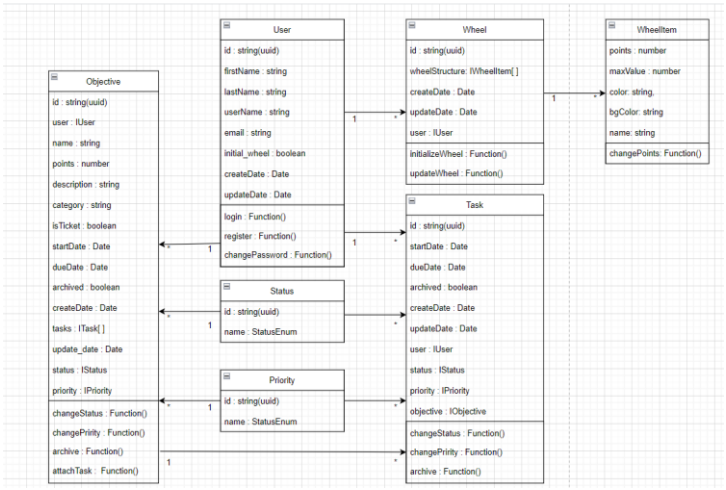


Рисунок 2.12 – Діаграма класів

Сама по собі діаграма дуже нагадує схему бази даних, але має додаткові властивості у вигляді методів або функцій які сутності можуть виконувати у коді, також замість вторинних ключів можна вказати сутності як властивості які ми можемо включити до об’єкту за допомогою запитів або маніпуляцій з даними під час їх обробки, так наприклад об’єкт Task тепер не просто знає первинний ключ Objective до якого він прив’язаний, а всю сутність Objective в той час як сам Objective має масив усіх Task які ло нього прив’язані, також вони маю схожі методи такі як changeStatus(), changePriority() та archive(), які відповідно роблять зміну статусу, пріоритету та архівування завдання, окрім цього сутність Objective має функцію attachTask() яка дозволяє додати до цілі нове завдання.

Однією з головних відмінностей діаграми класів так схемою бази даних є виділення окремої сутності WheelItem вона з’явилась в результаті сериалізації поля WheelStructure у таблиці Wheel і представляє собою окремо кожну сферу життя яку задекларував користувач при створенні початкового робочого простору, на діаграмі репрезентовано як властивість класу Wheel.

Також варто виділити об’єкт User так як у коді він буде не просто компонентом а обгорткою контекстом над усіма іншими сутностями, для більш зручної взаємодії з ним, також має такі функції як : логін, реєстрація та зміна паролю.

2.8 Алгоритм роботи додатку

Як вже було зазначено у попередньому підрозділі для отримання користувачем змоги використовувати повний функціонал першочергово йому необхідно виконати:

1. Реєстрацію та логін;

2. Проініціалізувати коло на підставі якого користувач в подальшому буде планувати свої кроки та створювати для них цілі;

Далі користувачу надається можливість :

3. Створювати нові задачі та для кожної сфери життя яку він обрав у колі при його ініціалізації та редагувати їх;
4. Редагувати коло – змінювати поточну оцінку категорії або додавати нові категорії;
5. Переглядати список завдань, користувача надається змогу використовувати для відображення даних як канбан дошку, так і звичайний список із завдань;
6. Переглядати статистику виконання завдань за обрані проміжки часу, та проводити аналіз продуктивності виконання та часу який було витрачено на реалізацію;
7. Переглядати поточний стан колеса;

Для створення нової задачі у додатку використовується модальне вікно, із усіма необхідними полями, після додання нової задачі вона з’являється на екрані у користувача у одному з двох видів відображення які користувач може самостійно обирати та перемикати у будь який час, також на кожній сторінці з лівої сторони завжди відображається коло, щоб користувач завжди міг бачити на що на даному етапі важливіше звернути увагу і правильно та виважено розставляти пріоритети.

РОЗДІЛ 3. РЕАЛІЗАЦЯ ВЕБ-ДОДАТКУ ДЛЯ САМОМЕНЕДЖМЕНТУ

3.1 Аргументація вибору стеку технологій

У другому розділі було дуже багато уваги приділено технологіям які використовувались під час розробки тому було б доречно розглянути їх в зв’язці. Першою перевагою стеку React.js та Node.js є те, що на сервері та на клієнтській частині використовується одна й та сама мова програмування, що безумовно є великим плюсом, багато речей такі interfaces, enums, types, helpers function , можна спокійно переносити з одного оточення в інше, лише з одним урахуванням helpers function повинні бути clean function(чиста функція, займається безпосередньо конкретною задачею і не має містити в собі побічних ефектів, clean function є однією з парадигм функціонального програмування, є одним з правил SOLID), друга проблема яка тут вирішується це кількість людей яка необхідна для розробки додатків на цьому стеку, оскільки знати потрібно лише одну мову, також Node.js зараз є одним з лідерів мов програмування для серверної розробки, оскільки має велику продуктивність і потребує меншу кількість ресурсів та часу для підтримки такого сервера, звісно в нього також є свої мінуси, але якщо головною метою є швидко розгорнути продуктивний сервер то Node.js у цьому немає рівних. React тут з тої самої причини, швидка розробка, максимально простий підхід до реалізації, велике ком’юніті з купою готових рішень, також має проблеми у вигляді відсутності CEO оптимізації яка необхідна для індексування додатку пошуковими ботами, але загалом великим веб додаткам вона не настільки необхідна як сайтам.

GraphQL в додатку використовується замість REST який безумовно є набагато легшим використанні, але не в реалізації, проблема в реалізації пов’язано з тим що багато часу витрачається на опис запитів та створення необхідної інфраструктури, щоб ці запити було зручно використовувати і

дуже часто доводиться робити одні й ті самі речі задля того щоб отримати трохи інший набір даних. В свою чергу головною перевагою GraphQL є готова інтеграція його до Node.js та React.js, яка дозволяє описувати запити, інтерфейси та типи на стороні серверу та автоматично генерувати їх на клієнті це досягається завдяки тому що GraphQL є строгою мовою запитів в якій ми чітко описуємо які властивості та сутності ми хочемо отримати, також це дозволяє використовувати одній ті самі запити з різним набором властивостей не створюючи нових, а просто змінити в запиті кількість полів яку ми хочемо отримати.

PostgreSQL був обраний через можливість зберігати данні у форматі JSON, до того ж він дуже гарно інтегрується з усіма ORM(Object Relation Mapping) це - технологія програмування, яка зв'язує бази даних з концепціями об'єктно-орієнтованих мов програмування, створюючи «віртуальну об'єктну базу даних».

3.2 Розробка серверної частини

3.2.1 Огляд структури серверу та його запуску

У розділі 2.5 вже підіймалась тема доцільності початку розробки саме з серверної частини, тому перше, що хотілося б розглянути це створення та розгортання серверу за допомогою платформи Node.js.

Для створення звичайного серверу який буде слухати запити з клієнтів за працювати з різноманітними сервісами в Node.js є вбудований API HTTP, але набагато зручніше використовувати вже готовий пакет express.

Рисунок 3.1 – Функція запуску сервера

На рисунку 3.1 показано застосування асинхронної функції яка запускає сервер, в ній описані ініціалізація серверу на рядку 22, далі підключаються middlewares(проміжні функції обробники) через команду app.use у дано випадку, першим ми підключаємо cors це пакет який відповідає за захищеність нашого серверу від запитів з невідомих ресурсів, у нашому випадку ми надаємо доступ до нашого локального клієнту та

ApolloServer який займається кешуванням даних та надає більш розширених можливостей GraphQL. Далі підключаються обробники які необхідні для сереалізації coockie та даних які передаються між запитами.

На 29 рядку ми декларуємо власний слухач запитів для аутентифікації, оскільки в додатку було впроваджено високий рівень захищеності через http-only-coockie, а GraphQL наразі не має можливості працювати з таким функціоналом тому цю ділянку коду довелось залишити на REST підході. Далі по коду йде підключення БД та ініціалізація ApolloServer та GraphQL, а вже в кінці запуск сервера, також все обгорнуто в блок try catch для відлову помилок під час запуску сервера.

Також важливо розглянути структуру серверної частини яка представлена на рисунку 3.2.

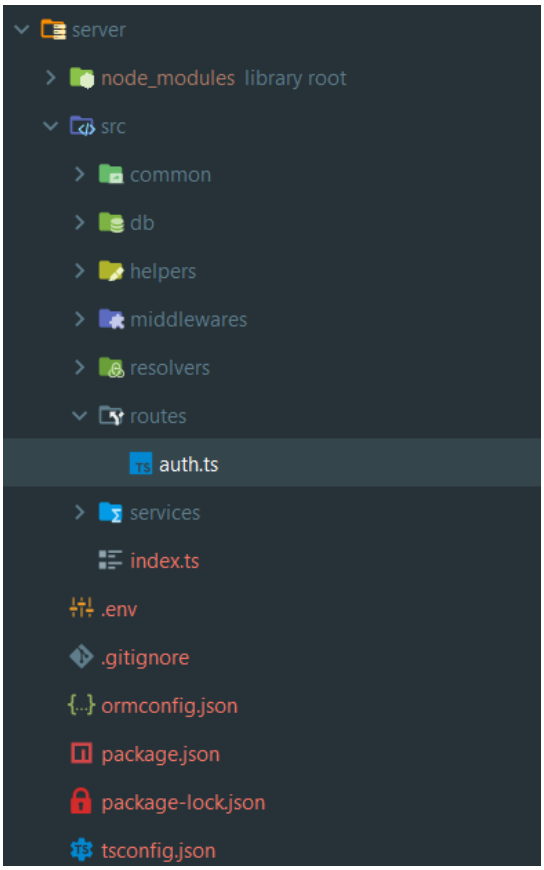


Рисунок 3.2 – Структура серверної частини

Першою і най головнішою папкою є node_modules так як у цій папці зберігаються усі залежності які встановлюються для розробки, без них наш код просто не запрацює, далі йде папка src в якій вже зберігається саме наш код написаний під час розробки, в ній знаходяться common папка в якій зберігаються описані нами типи та інтерфейси, db в якій ми описуємо відношення наших об’єктів до таблиць в базі даних та декларуємо типи для GraphQL, папка helpers зберігає в собі допоміжні функції які надалі ми будемо використовувати в коді, middelwares має в собі задекларовані функції обробки авторизації користувачів, routes має в собі описані REST запити в той час як resolvers зберігає GraphQL запити і остання папка це services в кій зберігається логіка роботи з поштою, а саме відправка листів під час авторизації.

В index.js знаходиться функція запуску серверу, так як він є початковою точкою роботи додатку, далі знаходяться конфіг файли в яких зберігаються:

- Конфігурації до підключення бази даних – файл ormconfig.json;
- .env основні глобальні змінні які ми хочемо заховати від користувачів;
- .gitignore необхідний для системи контролю версій git в ньому ми описуємо файли які не хочемо включати до списку файлів за зміною яких потрібно слідкувати;
- package.json та package-lock.json зберігають інформацію про наш проект, які залежності ми завантажували та якої версії а також скрипти запуску серверу або тесів вигляд файлу представлений на рисунку 3.3 та 3.4;
- tsconfig.json відповідає за налаштування середовище розробки мовою TypeScript;

```
{
  "name": "server",
  "version": "0.0.1",
  "description": "Awesome project develope
  "dependencies": {
    "apollo-server-express": "^3.6.2",
    "bcrypt": "^5.0.1",
    "body-parser": "^1.20.0",
    "brcypt": "^1.0.1",
    "class-validator": "^0.13.2",
    "cookie-parser": "^1.4.6",
    "cors": "^2.8.5",
    "dotenv": "^16.0.0",
    "express": "^4.17.2",
    "graphql": "15.3.0",
    "graphql-type-json": "^0.3.2",
    "jsonwebtoken": "^8.5.1",
    "nodemailer": "^6.7.3",
    "pg": "^8.7.3",
    "reflect-metadata": "^0.1.13",
    "type-graphql": "^1.1.1",
    "typeorm": "^0.2.41"
  },
  "devDependencies": {
    "@types/bcrypt": "^5.0.0",
    "@types/body-parser": "^1.19.2",
```

Рисунок 3.3 – Структура package.json файлу

```
  "@types/cookie-parser": "^1.4.2",
  "@types/cors": "^2.8.12",
  "@types/express": "^4.17.13",
  "@types/jsonwebtoken": "^8.5.8",
  "@types/node": "^17.0.14",
  "@types/nodemailer": "^6.4.4",
  "concurrently": "^6.2.1",
  "nodemon": "^2.0.15",
  "ts-node": "^10.4.0",
  "typescript": "^4.5.5"
},
"scripts": {
  "start": "nodemon src/index.ts",
  "generate": "ts-node ./node_modules/typeorm/cli.js migration:generate -n migrationName -p",
  "typeorm": "node --require ts-node/register ./node_modules/typeorm/cli.js"
}
```

Рисунок 3.3 – Структура package.json файлу

3.2.2 Використання typeORM для опису сутностей БД

Як вже зазначалося раніше для роботи з базою даних та взаємодією з нею у коді було використано typeORM, головною перевагою якої є декларування сутностей БД у класи з використанням анотацій які суттєво скорочують кількість необхідного коду, та роблять його більш легким для розуміння далі буде наведено приклад ініціалізації таблиці Status(див. рис.3.4) у коді :

```
@ObjectType()
@Entity()
export class Status extends BaseEntity {

    @Field( returnTypeFunction: () => String)
    @PrimaryGeneratedColumn( strategy: "uuid")
    id: string;

    @Field( returnTypeFunction: () => StatusEnum)
    @Column( options: {
        type: "enum",
        enum: StatusEnum
    })
    name: StatusEnum

    @OneToMany( typeFunctionOrTarget: () => Task,  inverseSide: task => task.status)
    task: Task

    @OneToMany( typeFunctionOrTarget: () => Objective,  inverseSide: objective => objective.status)
    objective: Objective

}
```

Рисунок 3.4 – Опис сутності БД за допомогою typeORM

У приладі вище видно що усі таблиці з БД можна представити у вигляді класів поля яких описуються та анотуються що-до їх представлення в БД GraphQL та коді , @Field відповідає за додання властивості до GraphQL

схеми, @Column за відображення та збереження в БД, сама ініціалізація поля відповідає за поведінку у коді. Також за допомогою анотацій можна вказувати додаткові властивості та привали такі як зв'язки між сутностями та типи в яких дані будуть зберігатись, або стратегії збереження первинних ключів.

3.2.3 Застосування Resolvers та Routes для описання запитів

Для декларування запитів у додатку використовується 2 підходи перший і основний з яких GraphQL другий REST який більш необхідний на стадії аунтентицікації так як дозволяє, більш легко та зручно працювати з coockie в яких зберігається авторизаційний токен.

Першим розглянемо REST підхід(див. рис. 3.5) :

```
const router = Router()
router.post( path: "/refresh_token",  handlers: async (req : Request<P, ResBody, ReqBody, ReqQuery, Locals> , res : Response<ResBody, Locals> ) => {
    const token = req.cookies.jid
    if (!token) {
        return res.send( body: {ok: false, accessToken: ""})
    }
    let payload: any = null
    try {
        payload = verify(token, process.env.SECRET_JWT_REFRESH_TOKEN!)
    } catch (e) {
        console.log(e)
        return res.send( body: {ok: false, accessToken: ""})
    }

    const user = await User.findOne( id: {id: payload.userId})

    if (!user) {
        return res.send( body: {ok: false, accessToken: ""})
    }

    if (user.tokenVersion !== payload.tokenVersion) {
        return res.send( body: {ok: false, accessToken: ""})
    }

    sendRefreshToken(res, createRefreshToken(user))
    return res.send( body: {ok: true, accessToken: createAccessToken(user)})
```

Рисунок 3.5 – Організація REST підходу у коді

З коду видно що розробка слухачів запитів починається зі створення router який буде прослуховувати запити, да лі ми викликаємо метод який необхідно використати для запиту (більш детально про роботу GraphQL та REST йдеться в розділі 2.3), також вказується назва кінцевої точки в нашому випадку це “refresh_token”, далі викликається асинхронна функція яка буде оброблювати сам запит, серед доступних параметрів у функції нема обов’язкових але ми маємо доступ до req(об’єкт з даними про запит), res(об’єкт для відправки результату до клієнту) та next(можна викликати щоб одразу перейти до наступного middelware), в самій функції описується логіка оновлення токену для аутентифікації, першим перевіряється наявність токену у coockie якщо він відсутній або не валідний, то на клієнт відправляється об’єкт з помилкою, якщо токен валідний, він декодується і перевіряється що зашифрований в ньому id користувача є валідним і такий користувач є в базі, при проходжені всіх перевірок в кінці користувач отримує об’єкт з новим токеном для подальшого продовження роботи з додатком.

Створення слухачів за допомогою GraphQL описується класами Resolvers (див. рис. 3.6) які між собою розподіляються за певним направленням у використанні :

```
@Resolver()
export class WheelResolvers {

    @Query( returnTypeFunc: () => Wheel,  options: {nullable: true})
    @UseMiddleware(isAuth)
    async getWheelByUserId(@Arg( name: "userId") userId: string): Promise<Wheel | null> {
        try {
            const wheel = await Wheel.findOne( id: {where: {userId}})
            if (wheel) {
                return wheel
            } else {
                return null
            }
        } catch (e) {
            throw new Error("Wheel not initialized")
        }
    }
```

Рисунок 3.6 – Декларація Resolver та методу Query

```
    @Mutation( returnTypeFunc: () => Boolean)
    @UseMiddleware(isAuth)
    async createUserWheel(@Arg( name: "structure") structure: string, @Arg( name: "userId") userId: string) {
        try {
            const user = await User.findOne(userId)
            if (!user?.initialWheel) {
                await getConnection().createQueryBuilder().update(User).set({
                    initialWheel: true
                }).execute()
                const res = await getConnection().getRepository(Wheel).insert( entity: {
                    user,
                    wheelStructure: structure
                })
                return !!res
            } else {
                const res = await getConnection().createQueryBuilder().update(Wheel).set({
                    wheelStructure:structure
                }).execute()
                return !!res
            }
        } catch (e) {
            return false;
        }
    }
}
```

Рисунок 3.7 – Декларація Mutation

За структурою самі запити більше нагадують звичайні функції, але з використанням анотацій, також самі запити поділяються на Query(див. рис. 3.6) та Mutation(див. рис. 3.7), перші відповідають лише за отримання даних в той час як мутації спрямовані на збереження та оновлення даних, описуються за допомогою анотацій @Query або @Mutation, також можна підключити до запитів middlewares через анотацію @UseMiddlewares в даному випадку було підключено анотацію для перевірки користувача на оновлений refreshToken.

Після написання Resolver він підключається в index.ts файлі реалізація представлена на рисунку 3.1.

3.2.4 Огляд Services та Middlewares

На разі серед сервісів серверна частина використовує, лише сервіс для роботи з поштою :

```
export class MailService {

    protected static getTransport() {
        const transporter = createTransport( transport: {
            service: process.env.SMTP_SERVICE!,
            host: process.env.SMTP_HOST!,
            auth: {
                pass: process.env.SMTP_PASS!,
                user: process.env.SMTP_USER!
            }
        })
        return transporter
    }

    protected static createMail({from, to, subject, html}: SendMailOptions) {
        return {
            from,
            to,
            subject,
            html
        }
    }
```

Рисунок 3.8 – Декларація Mail сервісу та ініціалізація методів createMail(створення листу) та getTransport(регламентування пошти відправника)

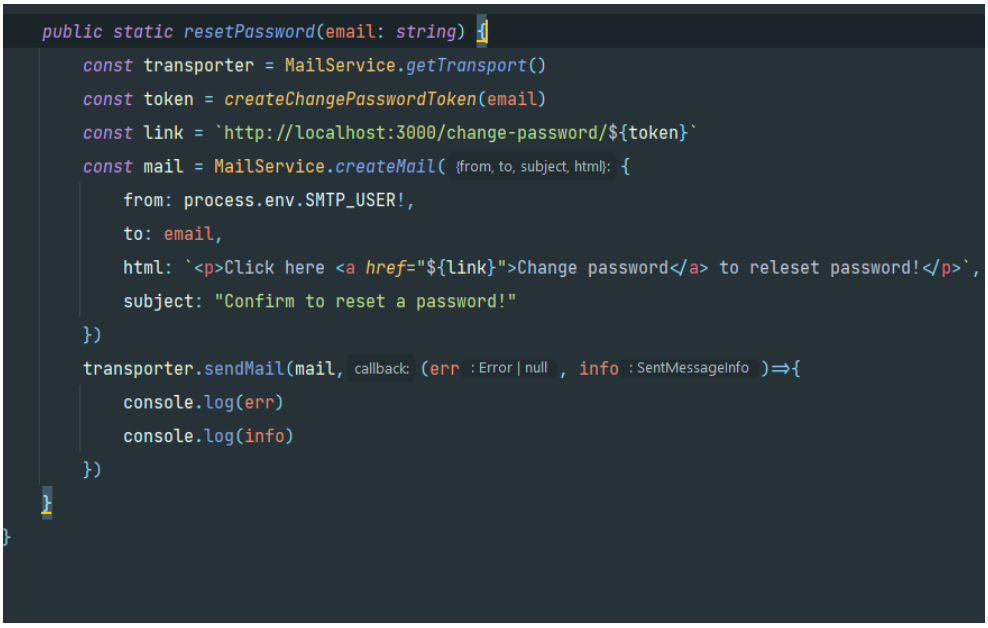


Рисунок 3.9 – Декларація методу відправки листа

Сам сервіс реалізований за допомогою класу MailService, насправді патерн створення сервісів через класи зі статичними методами є дуже зручним, через гарну читаємість таких сервісів та легку реалізацію, також можливість виклику методу сервісу без створення його екземпляру зменшує кількість зайвих об’єктів у пам’яті, сам сервіс має 2 захищені методи і 1 публічний:

- createMail - створює об’єкт листа(див. рис. 3.8);
- getTrapsort – декларує клієнт з якого буде відбуватись розсилка для ініціалізації використовує захищені змінні які розміщені у файлі .env(див. рис. 3.8);
- resetPassword – для відправки повідомлення на пошту(див. рис.);

Оскільки додаток спрямований на використання одним користувачем в робочому просторі, і система рівнів доступу відсутня, необхідність в великій кількості проміжних обробників в запитах є низькою, тому при розробці було

створено один middleware необхідний для перевірки автентифікації користувача який надсилає запит :

```
import {MiddlewareFn} from "type-graphql";
import {verify} from "jsonwebtoken";
import {ApolloReqResCtx} from "../common/types/apollo-req-res-ctx";

export const isAuth: MiddlewareFn<ApolloReqResCtx> = ({context :ApolloReqResCtx }, next :NextFn ) => {
    const authorization = context.req.headers["authorization"];
    if (!authorization) {
        throw new Error("not authenticated");
    }
    try {
        const token = authorization.split( separator: " ")[1];
        const payload = verify(token, process.env.SECRET_JW_ACCESS_TOKEN!);
        context.payload = payload as any;
    } catch (err) {
        console.log(err);
        throw new Error("not authenticated");
    }

    return next();
};
```

Рисунок 3.10 – Приклад реалізації middleware

Обробник спрацьовує до того як запит дійде то кінцевої точки в якій описана логіка його обробки, сама перевірка відбувається за допомогою пошуку заголовку authorization у запиті який зазвичай зберігає в собі авторизаційний токен користувача, далі він перевіряється на валідність, у випадку успіху обробник викликає функцію next та передає роботу наступному обробнику, у випадку помилки, надсилає користувачу повідомлення з помилкою.

3.3 Розробка клієнтської частини

3.3.1 Огляд структури клієнтської частини

В цілому структура клієнтської частини дуже схожа на серверну, окрім деяких папок, структура клієнтської частини представлена на рисунку 3.5.

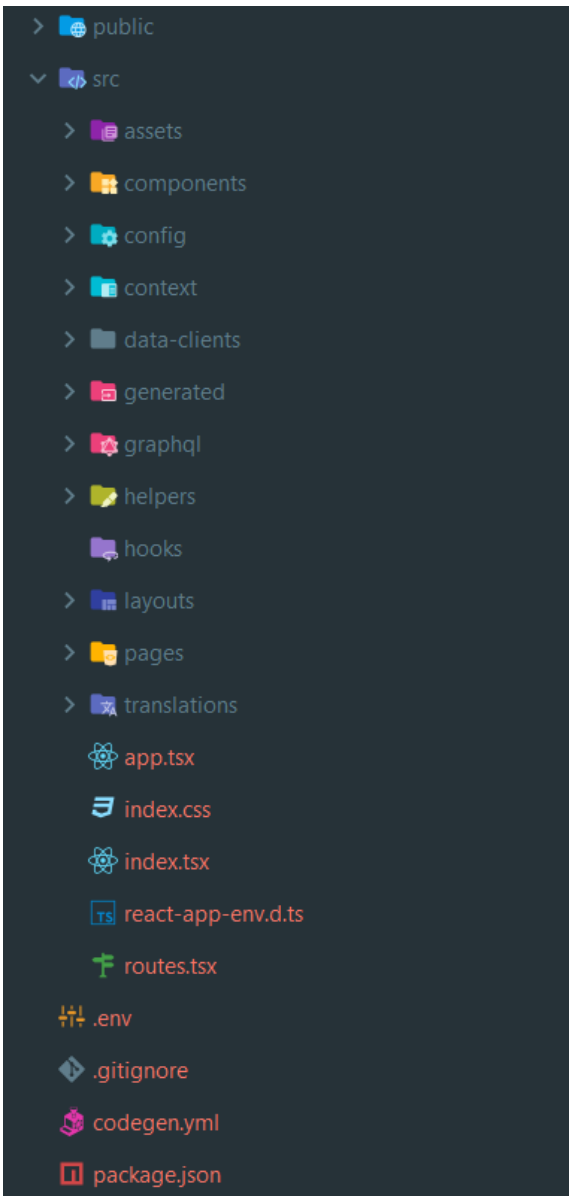


Рисунок 3.11 – Структура клієнтської частини

Головною відмінністю від серверу є більша кількість модулів на які розбивається логіка роботи додатку :

- Asstes – зберігає шрифти, іконки та фото-файли;
- Components – зберігає компоненти необхідні для відображення даних на інтерфейсі користувача;
- Config – містить файли налаштувань .env змінних та AppSetings;
- Contexts – зберігає обгортки контексту для певних частин додатку;
- Data-client – містить описані правила отримання даних для REST;
- Generated – зберігає згенеровані hooks для запитів мовою GraphQL, також автоматично згенеровані types та interfaces з серверної частини;
- GraphQL – у файлі описано які саме запити потрібно згенерувати;
- Hooks – зберігає наші власні кастомні хуки;
- Layouts – містить компоненти обгортки для різних, типів відображення в нашому випадку це відображення авторизаційних сторінок та основних;
- Pages – зберігає описані в додатку сторінки;
- Translation – потрібен для мульти-язичного налаштування в додатку;

Все інше залишається не змінним.

3.3.2 Запуск клієнтської частини

Ініціалізація клієнтської частини починається з однаково з сервером з файлу index.tsx(Розширення tsx використовується для описання файлів з JSX розміткою яку повертають компоненти):

```
const cache = new InMemoryCache()

const httpLink = new HttpLink( options: {
    uri: "http://localhost:4000/graphql",
    credentials: "include"
})
```

Рисунок 3.12 – Декларація кешу та посилання на запити

```
const refreshLink = new TokenRefreshLink( params: {
    accessTokenField: "accessToken",
    isTokenValidOrUndefined: () => {
        const token = AppSettings.getToken()
        if (!token) {
            return true;
        }
        try {
            const decoced: any = jwtDecode(token);
            if (Date.now() ≥ decoced!.exp * 1000) {
                return false;
            } else {
                return true;
            }
        } catch {
            return false;
        }
    },
```

Рисунок 3.13 – Декларація методу оновлення токену доступу до запитів;

```
    fetchAccessToken: () => {
        return fetch( input: 'http://localhost:4000/api/v1/auth/refresh_token', init: {
            method: "POST",
            credentials: "include"
        })
    },
    handleFetch: accessToken => {
        AppSettings.setToken(accessToken);
    },
    handleError: (err : Error , operation : Operation ) => {
        console.warn( message: "Your refresh token is invalid. Try to relogin");
        console.error(err);
    }
)
```

Рисунок 3.14 - Декларація методу оновлення токену доступу до запитів;

```
const authMiddleware = new ApolloLink( request: (operation : Operation , forward : NextLink ) => {
    const refreshToken = AppSettings.getToken()
    operation.setContext( context: ({headers : 0  = {}}) => ({
        headers: {
            ...headers,
            authorization: `Bearer ${refreshToken}`,
        }
    }));
    return forward(operation);
})
```

Рисунок 3.15 – Проміжний обробник запитів для додання автентифікації

```
const client = new ApolloClient( options: {
    cache,
    link: from( links: [
        refreshLink,
        authMiddleware,
        onError( errorHandler: ({graphQLErrors : readonly GraphQLError[] | undefined , networkError : Error | … , operation : Operation , forward}) => {
            console.log(graphQLErrors);
            if (graphQLErrors) {
                toast.error(graphQLErrors[0].message)
            }
            console.log(networkError);
        }),
        httpLink
    ])
});
```

Рисунок 3.16 – Ініціалізація задекларованих методів та функцій у обробник

```
ReactDOM.render(
    <BrowserRouter>
        <ApolloProvider client={client}>
            <AuthProvider>
                <App/>
                <ToastContainer
                    position="top-right"
                    autoClose={5000}
                    hideProgressBar={false}
                    newestOnTop={false}
                    closeOnClick
                    rtl={false}
                    pauseOnFocusLoss
                    draggable
                    pauseOnHover
                />
            </AuthProvider>
        </ApolloProvider>
    </BrowserRouter>,
    document.getElementById( elementId: 'root'));
```

Рисунок 3.17 – Створення контексту додатку

На початку файлу відбувається декларування змінної для збереження кеш даних, далі створюється HTTP лінк(див. рис. 3.12) який буде використовуватись для посилання запитів з клієнта на сервер за допомогою

GraphQL, наступним створюється refreshLink(див. рис. 3.13-3.14), який є необхідним для оновлення refreshToken, щоб користувачу не доводилось перезавантажувати сторінку для його оновлення, а це відбувалось автоматично перед запитом за необхідності, слідом за ним декларується обробник авторизації який включає до заголовків запиту наш token, який далі перевіряється на сервері, деталі роботи якого описані в розділі 3.2.4, потім ці всі змінні складаються в об'єкт client(див. рис. 3.16) який далі передаються в ApolloProvider(див. рис. 3.17) для подальшої можливості використання GraphQL запитів у всьому додатку. Також у самій ієрахії компонентів ми відмалюємо наш головний компонент App, обгортаємо його в AuthProvider(див. рис. 3.17) опис роботи якого буде трохи нижче, та додаємо компонент для відображення вспливаючих вікон в нашому додатку, оскільки ми хочемо мати змогу викликати їх з любого місця в додатку то краще винести їх на самий високий рівень ієрархії компонентів. У файлі app.tsx(див. рис. 3.18) ми описуємо головний компонент :

```
const refreshTokenRequest = async () => {
    const {ok, accessToken} = await AuthClient.getRefreshToken()
    if (ok) {
        AppSettings.setToken(accessToken)
        setIsAuth( val: true)
        setLoading( value: false)
    } else {
        setIsAuth( val: false)
        setLoading( value: false)
    }

}

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

if (loading) {
    return <div>Loading...</div>
}

return (
    <>
        {router}
    </>
)
```

Рисунок 3.18 – Декларація компоненту додатку

Оскільки він буде першим виконуватись кожного разу при перепуску сторінки, то в ньому ми будемо викликати REST кол для оновлення refreshToken, та в залежності від успішності цього запиту будемо віддавати користувачу сторінки, при успішному оновлені користувач отримує головні сторінки а при не успішному сторінки автентифікації, ця логіка описана у файлі routes.tsx(див. рис. 3.19) :

```
export const useRoutes = ({isAuth}: UseRoutesProps) => {

    if (isAuth) {
        return (
            <Routes>
                <Route path={RoutesPath.Home} element={<Home/>}/>
                <Route path={RoutesPath.Main} element={<Main/>}/>
                <Route path={RoutesPath.Statistic} element={<Statistic/>}/>
                <Route path={RoutesPath.Logout} element={<Exit/>}/>
                <Route path={RoutesPath.CreateWheel} element={<CreateWheel/>}/>
                <Route path={"*"} element={<NoFound isAuth={true}/>}/>
            </Routes>
        );
    }
    if (!isAuth) {
        return (
            <Routes>
                <Route path={RoutesPath.Login} element={<Login/>}/>
                <Route path={RoutesPath.Register} element={<Register/>}/>
                <Route path={RoutesPath.ForgotPassword} element={<ForgotPassword/>}/>
                <Route path={RoutesPath.ChangePassword} element={<ChangePassword/>}/>
                <Route path={"*"} element={<NoFound isAuth={false}/>}/>
            </Routes>
        )
    }
    return null
```

Рисунок 3.19 – Ініціалізація сторінок додатку

В залежності від стану користувача отримується різний список доступних сторінок.

Сам стан аунтентифікації користувача було винесено в AuthContext і обгорнуто ним весь додаток, це не обхідно для того, якщо користувач вийде з сесії або його токен завершить свою дію, то контекст оновиться і увесь додаток буде вимушений відмальовуватись заново, з правильним контекстом виконання, опис файлу authContext.tsx(див. рис. 3.20) :

```
const AuthContext = createContext<AuthContext>( defaultValue: {
    setState: {
        setIsAuth: (val : boolean ) => {
        }
    },
    state: {
        isAuth: false
    }
})

const AuthProvider: React.FC = ({children : ReactElement<any, string | JSXElementConstructor<any>> | ... }) => {
    const [isAuth, setIsAuth] = useState( initialState: false)
    return (
        <AuthContext.Provider value={{
            state: {isAuth},
            setState: {setIsAuth}
        }}>
            {children}
        </AuthContext.Provider>
    )
}

const useAuth = () => useContext(AuthContext)

export {AuthProvider, useAuth}
```

Рисунок 3.20 – Створення власного контексту

Тут все доволі просто, необхідно лише створити стан компоненту який буде слідкувати за статусом автентифікації користувач, і всі дочірні елементи

які ми в нього передали, також необхідно зберегти тому ми їх кладемо в середу обгортки, таким чином вони будуть знати цей контекст.

3.3.3 Реалізація компоненту Kanban card

Основна задача цього компоненту, відобразити задачу у вигляді карточки зі статусом, пріоритетом та іменем :

```
const KanbanCard: React.FC<KanbanCardProps> = ({data : Objective , colors : CardColors ,updateState : … }) => {

    const [showPriorityTooltip, setShowTooltip] = useState<boolean>( initialState: false)
    const [showModal, setShowModal] = useState<boolean>( initialState: false)
    const getPriorityIconByIdName = useMemo( factory: () => {
        switch (data.priority.name) {
            case PriorityEnum.Low:
                return {icon: "bi-diamond", color: "text-blue-500", bg: "bg-blue-500"}
            case PriorityEnum.Medium:
                return {icon: "bi-diamond-half", color: "text-yellow-500", bg: "bg-yellow-500"}
            case PriorityEnum.High:
                return {icon: "bi-diamond-fill", color: "text-red-500", bg: "bg-red-500"}
            default:
                return {icon: "", color: "", bg: "text-blue-500"}
        }
    }, deps: [data])
```

Рисунок 3.21 – Функціональна частина компоненту KanbanCard

Рисунок 3.22 – Частина компоненту KanbanCard яка повертає розмітку

При ініціалізації компонента ми чекаємо, що до нього буде передано 3 основні властивості, це інформація про карточку, кольори її відображення та функція-хендлер оновлення стану додатку, оскільки ми 1 користувач у своєму робочу просторі, немає необхідності оновлювати данні з бази кожного разу, достатньо отримати позитивний результат збереження карточки не сервері, і додати до списку карточок на клієнті тим самим провівши, не складну але значну оптимізацію.

Також в середні карточки окрім відмалювання розмітки, зберігається стан відкриття модального вікна оновлення даних карточки, у випадку натискання на іконку оновлення стан зміниться і модальне вікно буде показано користувачеві, також у саме модальне вікно ми предаємо данні які потрібно там показати, і функцію-хендлер закриття карточки.

3.3.4 Реалізація головної сторінки

Головна сторінка додатку займається відображенням, нагального результату роботи користувача з лівого боку, та 2-х способів відображення списку завдань з правого боку екрану:

```
enum Tabs {
    KANBAN = "kanban",
    LIST = "list"
}

const Main = () => {
    const {data: me} = useMeQuery( baseOptions: {fetchPolicy: "network-only"})
    const navigate = useNavigate()
    const [wheelState, setWheelState] = useState<SphereOfLife[]>( initialState: [])
    const [objectivesState, setObjectivesState] = useState<Objective[]>( initialState: [])
    const [showModal, setShowModal] = useState<boolean>( initialState: false)
    const [activeTab, setActiveTab] = useState<Tabs>(Tabs.KANBAN)
    const userId = me?.me?.id
    const {loading: wheelLoading} = useGetWheelByUserIdQuery( baseOptions: {
        variables: {
            userId: userId as string
        },
        skip: userId == undefined,
        onCompleted: (data => setNewWheelState(data))
    })

    const {data: priorities} = useAllPrioritiesQuery()
    const {data: statuses} = useAllStatusesQuery()
```

Рисунок 3.23 – Декларація компоненту головної сторінки

```
const {loading: objectivesLoading} = useGetObjectivesByUserQuery( baseOptions: {
    onCompleted: (data => {
        if (data?.getObjectivesByUser) {
            setObjectivesState(data.getObjectivesByUser)
        }
    })
})

const setNewWheelState = (data: any) => {
    if (data.getWheelByUserId?.wheelStructure) {
        setWheelState(JSON.parse(data.getWheelByUserId?.wheelStructure))
    }
}

const saveItemHandler = (data: Objective) => {
    setObjectivesState( value: prevState => [...prevState, data])
}

useEffect( effect: () => {
    if (me?.me) {
        if (!me.me.initialWheel) {
            navigate(RoutesPath.CreateWheel)
            toast.info( content: "First off all you need create your own wheel!")
        }
    }
```

Рисунок 3.24 – Опис методу життєвого циклу для перевірки ініціалізації робочого

```
            <ObjectivesList data={wheelState}/>
                <Wheel radius={400} data={wheelState}/>

        </div> : <div
            className={"flex w-[500px] bg-zinc-50 p-4 justify-center items-center h-full border-2 rounded"}>
            <Loader/>
        </div>}
    <div className={"w-full"}>
        <div className={"flex w-full  justify-between"}>
            <div className={"flex"}>
                <div onClick={() => setActiveTab(Tabs.KANBAN)}
                    className={`p-2 hover:cursor-pointer ${activeTab === Tabs.KANBAN ? "bg-zinc-50 border-x-2 border-t-2 rounded-t" : ""}`}>Kanban
                </div>
                <div onClick={() => setActiveTab(Tabs.LIST)}
                    className={`p-2 hover:cursor-pointer ${activeTab === Tabs.LIST ? "bg-zinc-50 border-x-2 border-t-2 rounded-t" : ""}`}>Task
                   List
                </div>
            </div>
            <div className={"my-1"}>
                <FormBtn type={FormBtnType.Button} title={"Create Objective"}
                         styleType={FormBtnStyleType.Regular} onClick={() => setShowModal( value: true)}/>
            </div>
            <AddObjectiveModal saveItem={saveItemHandler} closeModal={() => setShowModal( value: false)}
                               showModal={showModal}/>
```

Рисунок 3.25 – Розмітка компоненту головної сторінки

На початку коду ми декларуємо enum(див. рис. 3.23) який буде необхідний для зберігання стану відображення списку задач, далі в самому компоненті ми бачимо використання hooks для отримання даних які нам сгенерував GrqphQL, на сторінці їх 5(див. рис. 3.23-3.24) :

- useMeQuery – для отримання даних користувача;
- useGetWheelByUserIdQuery – для отримання даних прогресу користувача;
- useAllPrioritiesQuery – для отримання всіх наявних пріоритетів;
- useAllStatusesQuery – для отримання всіх наявних статусів;
- useGetObjectivesByUserQuery – отримання всіх карток користувача, також приймає до виклику userId по якому ми будемо робити вибірку в БД, хендлер завершення загрузи, і індикатор початку запиту;

Також компонент має стан типу відображення який ми можемо перемикати і в залежності від типу, відображати або Kanban або TaskList(див. рис. 3.25).

3.4 Огляд функціонуючих сторінок додатку

Перші сторінки з якими в додатку стикається користувач майже завжди є сторінки аутентифікації, загалом най необхіднішими є лише 2 Логін та Реєстрація але інколи вони об’єднуються в одну сторінку, також майже всі додатки мають сторінки для відновлення та зміни паролю нижче на малюнках буде представлено 4 автентифікаційні сторінки :

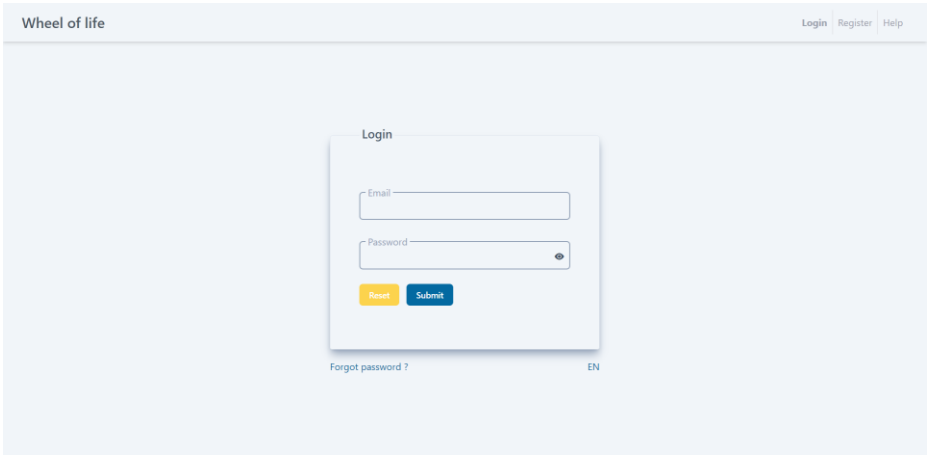


Рисунок 3.26 – сторінка логіну.

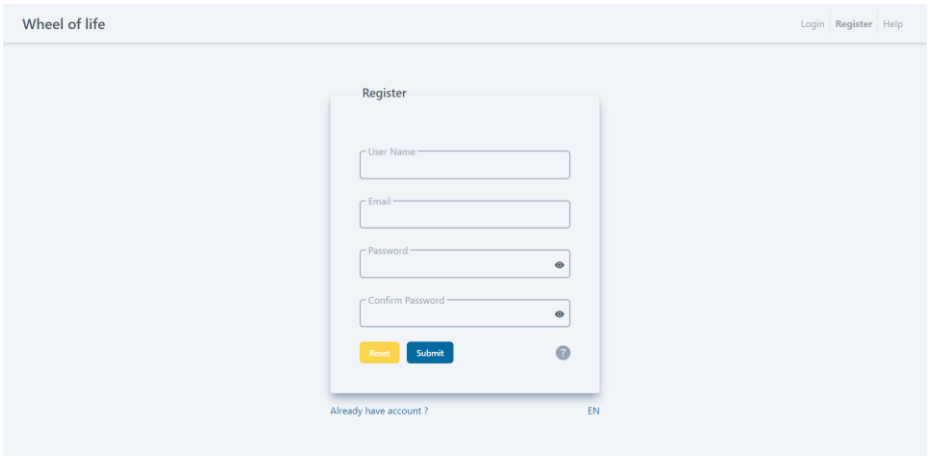


Рисунок 3.27 – сторінка реєстрації.

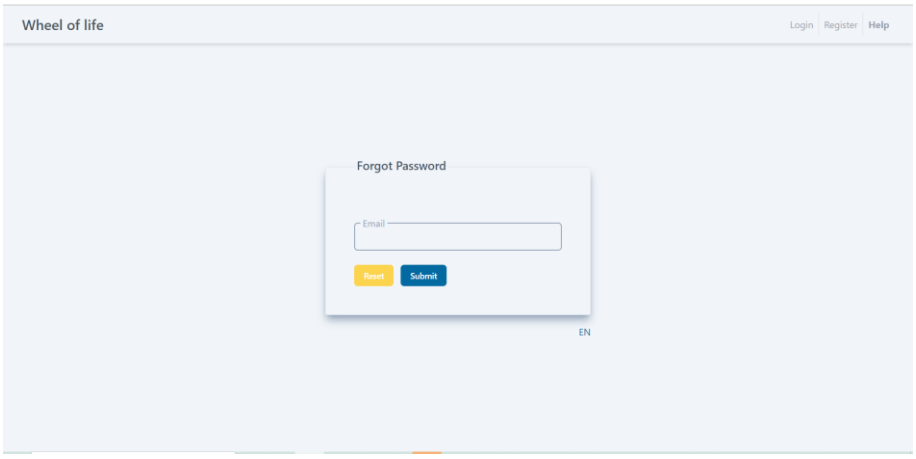


Рисунок 3.28 – сторінка з запитом на зміну паролю.

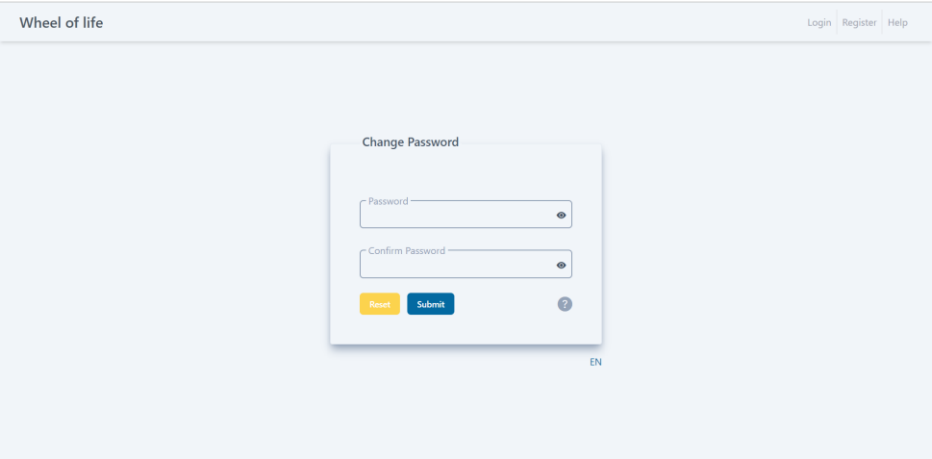


Рисунок 3.29 – сторінка зі зміною паролю.

Всі аутентифікаційні сторінки мають спільну шапку для більш зручної навігації та невеликі лінки під формами, також перехід до сторінки зміну паролю відбувається через електронний лист надісланий на пошту яку користувач обрав при реєстрації, приклад листа наведено на рисунку 2.13.

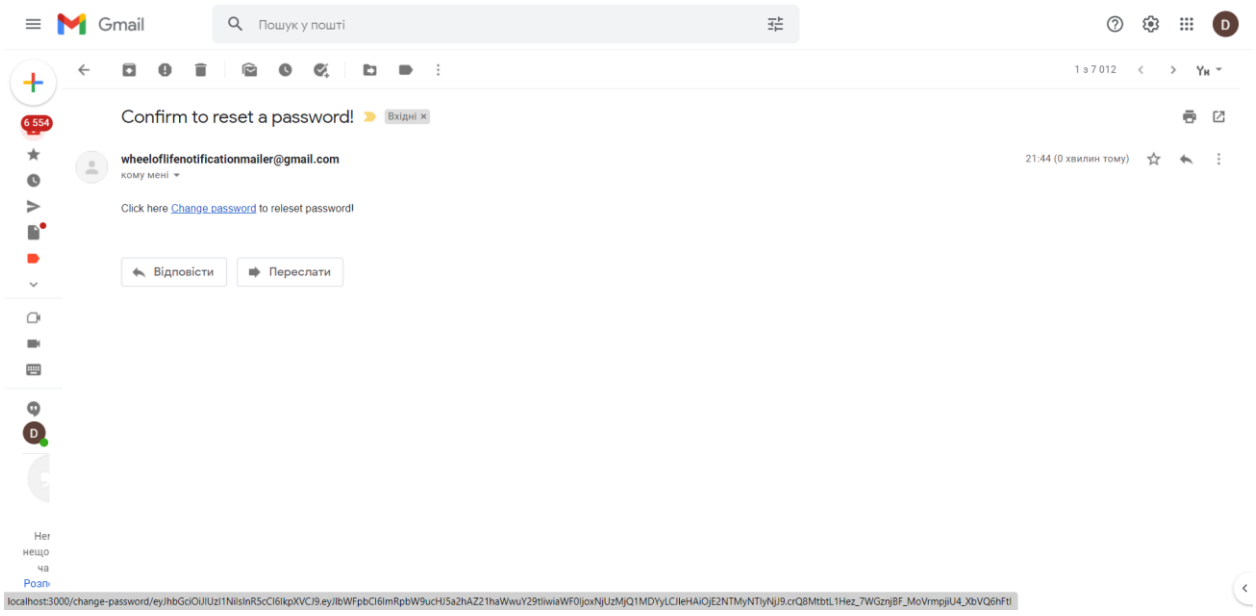


Рисунок 3.30 – приклад надісланого на пошту письма для зміни паролю, має лінк з токеном який потрібен для більш захищеної зміни паролю.

Після аутентицікації при першому вході до системи користувач автоматично направляється на сторінку ініціалізації колеса, якщо колесо вже ініціалізоване то потрапляє на головну сторінку з відображенням наявних задач.

Рисунок 3.31 – Сторінка заповнення колеса

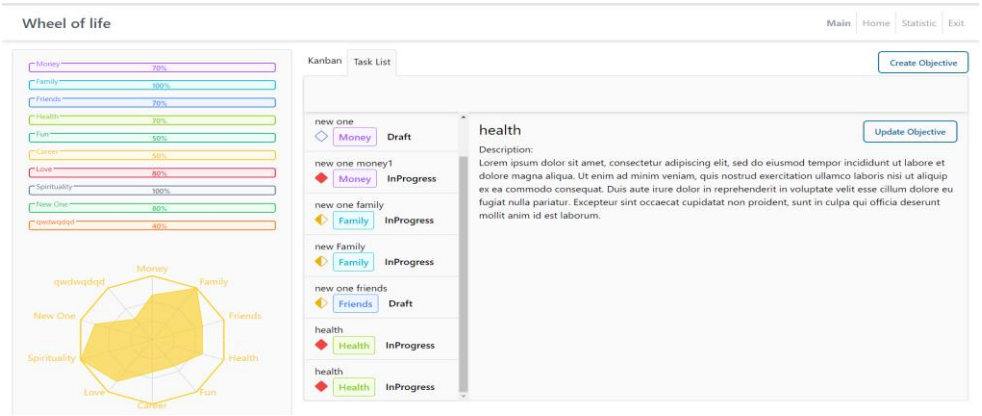


Рисунок 3.32 – Головна сторінка з увімкнутим типом відображення “List”

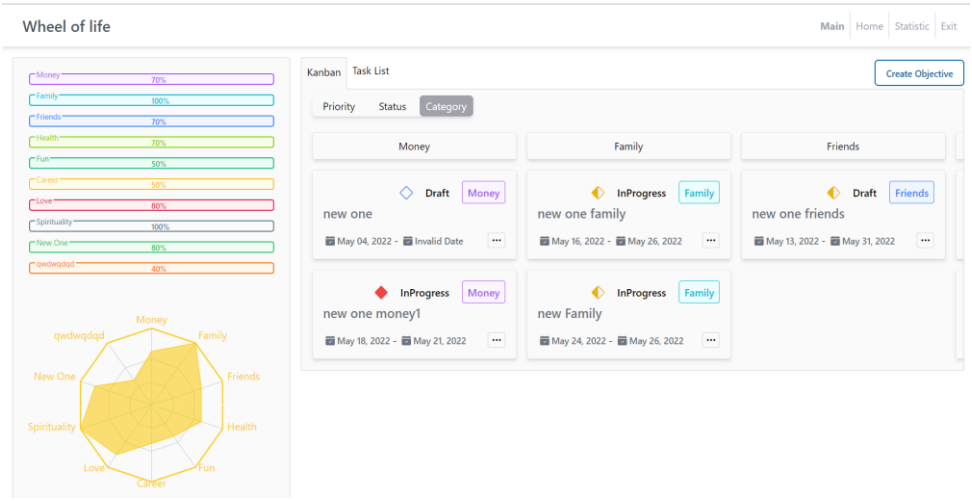


Рисунок 3.33 – Головна сторінка з увімкнутим типом відображення “Kanban”

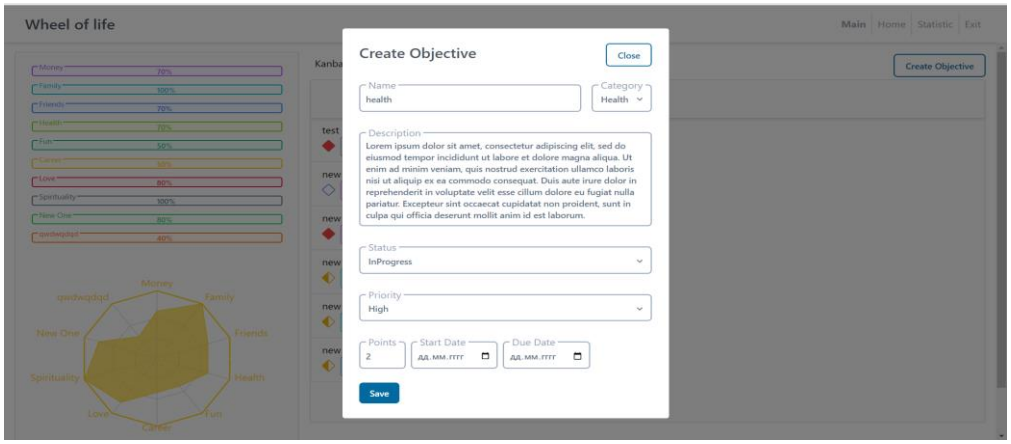


Рисунок 3.34 – Модальне вікно додання задач

ВИСНОВКИ

Додатки самоменеджменту все більше займають місця у нашому житті, як на роботі так і у особистому житті, вони надають користувачам змогу, швидко та зручно організовувати визначення цілей та шляхи їх виконання, та взаємодіяти у певних робочих групах.

Зважаючи на вище описані фактори, було прийнято рішення розробити власний додаток самоменеджменту у напрямку лайф-менеджмент, головними перевагами якого над аналогами є :

- Можливість портувати додаток на більшість існуючих та популярних платформ;
- Швидка ініціалізація робочого простору;
- Декілька варіантів відображення даних;
- Швидка та надійна автентифікація;
- Новий стек технологій який забезпечую більшу оптимізацію і можливості ;
- Зручний та швидкий спосіб відображення наявного результату у всіх обраних сферах діяльності;

На базі таких технологій як мова програмування TypeScript, біблотека для створення слакдних односторінкових веб-додатків React.js, платформа для виконання JavaScript на сервері Node.js, та бази даних PostgreSQL.

ПЕРЕЛІК ПОСИЛАНЬ

1. So what`s GraphQL thing I keep hearing about? [Електронний ресурс]. – Режим доступу до ресурсу: https://www.freecodecamp.org/news/so-whats-this-graphql-thing-i-keep-hearing-about-baf4d36c20cf
2. Що таке колесо життя і як воно допоможе в особистому розвитку [Електронний ресурс]. – Режим доступу до ресурсу: https://blog.agrokebety.com/shcho-take-koleso-zhyttya
3. Сучасні погляди на сутність самоменеджменту [Електронний ресурс]. – Режим доступу до ресурсу: http://www.km.lviv.ua/wp-content/uploads/2016/04/Samomenedzhment.pdf
4. What is PostgreSQL [Електронний ресурс]. – Режим доступу до ресурсу: https://www.postgresqltutorial.com/postgresql-getting-started/what-is-postgresql/
5. Introduction to Node.js [Електронний ресурс]. – Режим доступу до ресурсу: https://nodejs.dev/learn
6. The Best Guide to Know What Is React [Електронний ресурс]. – Режим доступу до ресурсу: https://www.simplilearn.com/tutorials/reactjs-tutorial/what-is-reactjs
7. How to Use TypeScript – Begginer-Friendly TS Tutorial [Електронний ресурс]. – Режим доступу до ресурсу: https://www.freecodecamp.org/news/an-introduction-to-typescript/
8. Introduction to REST API - RESTful Web Services [Електронний ресурс]. – Режим доступу до ресурсу: https://www.springboottutorial.com/introduction-to-rest-api
9. How to Create a React App with a Node Backend: The Complete Guide [Електронний ресурс]. – Режим доступу до ресурсу: https://www.freecodecamp.org/news/how-to-create-a-react-app-with-a-node-backend-the-complete-guide/

ДОДАТОК А

ДЕРЖАВНИЙ УНІВЕРСИТЕТ ТЕЛЕКОМУНІКАЦІЙ
НАВЧАЛЬНО-НАУКОВИЙ ІНСТИТУТ ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ
КАФЕДРА ІНЖЕНЕРІЇ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

Розробка web-додатку самоменеджменту засобами React.js, Node.js

Виконав студент 4 курсу
групи ПД-44
Приходько Дмитрій Анатолійович
Керівник роботи
К.т.н., доц., доцент кафедри ІПЗ Золотухіна Оксана Анатоліївна

Київ – 2022

Аналоги

Jira

LifeWheel

Trello

Таблиця порівнянь аналогів

Назва	Переваги	Недоліки
Jira	• Великий набір інструментів • Багато способів відображення даних та статистики • Високий рівень захищеності даних • Зручний для роботи в команді • Має інтеграцію з багатьма іншими програмами	• Складний у використанні та вивченні інфраструктури • Довгий процес створення робочого простору • Не зручний для лайф-менеджменту
Trello	• Зручний для самоменеджменту та менеджменту цілей команд • Легкий та зрозумілий інтерфейс • Можна легко та швидко почати роботу з додатком • Має інтеграцію з багатьма іншими програмами	• Відносно низький рівень захищеності даних • Обмежений набір інструментів • Лише один спосіб відображення даних
LifeWheel	• Зручний у використані з телефону • Зручний для ведення самоменеджменту та лайф-менеджменту • Зручний та гарний спосіб відображення наявного прогресу	• Доступний лише на платформі IOS • Більшість частина функцій платна; • Не зручний спосіб роботи з наявними задачами • Відсутність ведення статистки виконаних задач

3

Мета, об’єкт та предмет дослідження

- **Мета роботи** – налагодження та спрощення процесів управління власним часом та ресурсами за рахунок впровадження програмного додатку самомененджменту.
- **Об’єкти досліджень** - процеси самоменеджменту.
- **Предмет дослідження** – додатки для самоменеджменту.

Завдання бакалаврської роботи

1. Аналіз існуючих додатків самоменеджменту, з метою виявлення недоліків та дослідженням існуючих патернів побудови користувацьких інтерфейсів за цим напрямком;
2. Розробка архітектури додатку для самоменеджменту;
3. Розробка архітектури бази даних;
4. Розробка серверної частини за “багато-слойоною” архітектурою;
5. Розробка архітектури клієнтської частини на основі компонентного підходу;
6. Реалізація головних модулів додатку :
   - Авторизація користувача;
   - Створення початкового робочого простору для подальшого його використання;
   - Додавання нових цілей та редагування вже існуючих;
   - Можливість перегляду даних у декількох видах відображення;
   - Відображення поточного результату діяльності користувача у обраних сферах;

Програмні засоби реалізації

TailwindCSS - фреймворк оснований на методології BEM, головною відмінністю від подібних йому є - модульна система створення власних динамічних CSS стилів та відсутність готових компонентів що робить більш зручною розробку додатків з не шаблонним дизайном.

React.Js – JavaScript бібліотека з відкритим кодом, використовується для створення односторінкових веб додатків, використовує компонентний підхід, та дозволяє використовувати мультипарадигменість для більш гнучкої розробки.

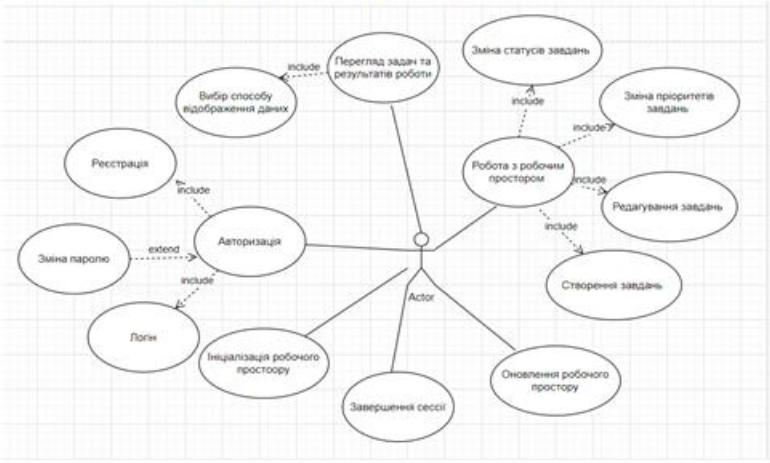
WebStorm – розумна IDEA для розробки : мобільних, веб та десктоп додатків, мовами програмування JavaScript та TypeScript.

PostgreSQL – реляційна база даних, була обрана через можливість зберігання даних в таблицях у форматі JSON.

GraphQL – мова запитів між сервером та клієнтом, є строго типізованою, має гарну інтеграцію з сучасними мовами програмування та надає великий багато готових рішень з “коробки”.

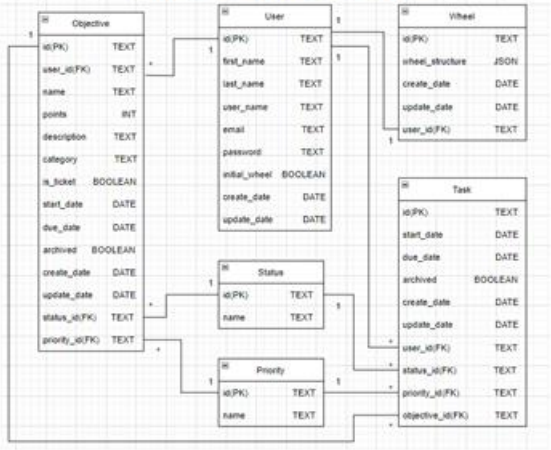
Node.Js – платформа з відкритим кодом для виконання JavaScript на сервері та створення оточення розробки для сучасних веб, десктоп та мобільних додатків.

Діаграма Прецедентів



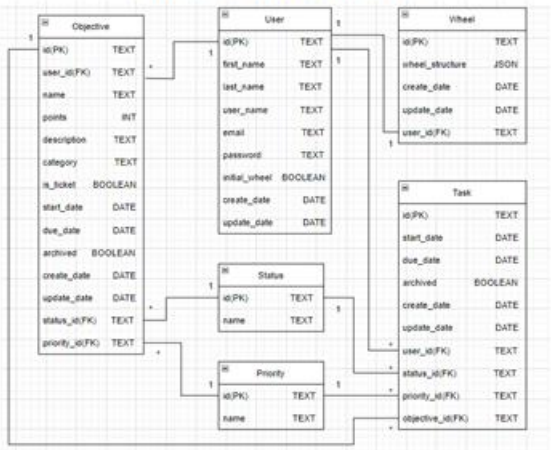
7

Схема бази даних



8

Схема бази даних



8

Розроблені форми авторизації

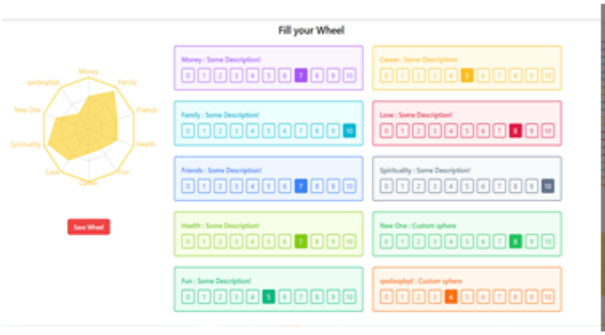
Лоігн

Реєстрація

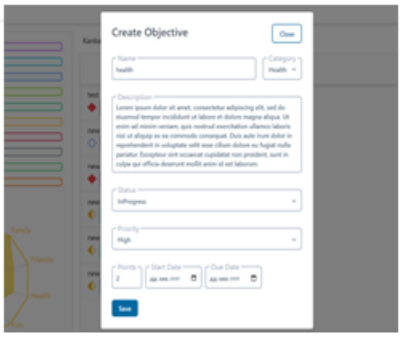
Запит на зміну паролю

Зміна паролю

Основні форми взаємодії з користувачем

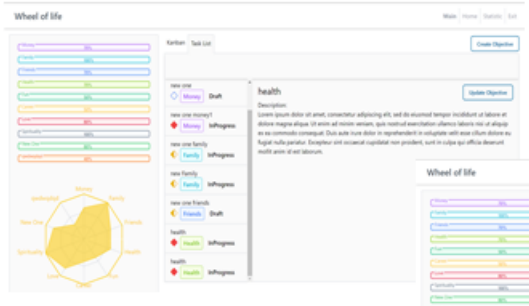


Сторінка ініціалізації робочого простору

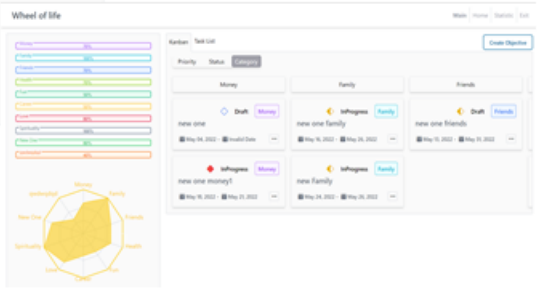


Модальне вікно створення/редагування завдань

Головна сторінка



Відображення даних у вигляді списку



Відображення даних у вигляді дошки

Апробація результатів дослідження

1. Приходько Д.А., "Wheel-of-life (колесо-життя) " найкращий спосіб самоменеджменту/ науково-Технічна конференція "Застосування програмного забезпечення в інфокомунікаційних технологіях" Збірник тез 20.04.2022, ДУТ, м. Київ – К.: ДУТ, 2022. С.14-15

2. Приходько Д.А., "GraphQL" як сучасний спосіб обміну даними / Науково-технічна конференція "застосування програмного забезпечення в інфокомунікаційних технологіях" Збірник тез 20.04.2022, ДУТ, м. Київ – К.: ДУТ, 2022. С.16-17

Висновки

1. Проведено аналіз додатків, які використовуються для організації самоменджменту. Ключовими недоліками аналогів є те, що вони мають в більшості тільки один спосіб відображення даних. При цьому сам процес налаштування робочого простору є досить довгим, а створення нових завдань та взаємодія з ними – складні. Значним недоліком також виступає відсутність кросс-платформеності.
2. Розроблено архітектуру додатку самоменеджменту особливостям якої є : багаторівнева архітектура з боку сервера – підхід при якому програма розбивається на певну кількість рівнів основними з яких можна виділити : controllers(слухачі подій), services(робота зі сторонніми сервісами), repositories(взаємодія з БД), middlewares(проміжні обробники подій), entities(декларація сутностей БД у коді), на стороні клієнта використано функціонально-компонентний підхід який є більш гнучким та швидшим у розробці на відміну від об'єктно орієнтованого, що досягається завдяки будуванню архітектури композицією, а не наслідуванням.
3. Розроблено діаграму класів та схему бази даних виходячи зі стеку технологій в який входить Node.js, TypeScript, PostrgeSQL та React.js завдяки використанню PostgreSQL було зменшено кількість таблиць завдяки винесенню деяких структур у JSON об'єкти. Також використання TypeScript який є мультипарадигменною мовою програмування, сутності з боку сервера було представлено як класи які зв'язуються з базою за допомогою ORM, а вже при обробці та відправці даних клієнт вони приймають вигляд компонентів з реалізацією у вигляді функцій.
4. Розроблено серверну частину додатку з використанням платформи Node.js та мови програмування TypeScript.
5. Налагоджено процес обміну даних між сервером та клієнтом за допомогою мови запитів GraphQL.
6. За допомогою таких інструментів як : JavaScript бібліотека React.js, tailwindCSS та мови програмування TypeScript було реалізовано основні модулі :
    1. Авторизація користувача;
    2. Створення початкового робочого простору для подальшого його використання;
    3. Додавання нових цілей та редагування вже існуючих;
    4. Можливість перегляду даних у декількох видах відображення;
    5. Відображення поточного результату діяльності користувача у обраних сферах;

ДЯКУЮ ЗА УВАГУ!