Міністерство освіти і науки України
НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ «КИЄВО-МОГИЛЯНСЬКА АКАДЕМІЯ»
Кафедра мережних технологій факультету інформатики

РОЗРОБКА ВЕБ-ЗАСТОСУВАННЯ ДЛЯ ПІДВИЩЕННЯ ПРОДУКТИВНОСТІ ОРГАНІЗАЦІЇ РОБОТИ

Текстова частина до курсової роботи
за спеціальністю «Комп'ютерні науки» 122

Керівник курсової роботи
кандидат фізико-математичних наук,
Гречко А. В.

_____

(підпис)

«____» _____ 2021 р.

Виконала студентка Лукічова С. О.

«____» _____ 2021 р.

Київ – 2021

Міністерство освіти і науки України

НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ «КИЄВО-МОГИЛЯНСЬКА АКАДЕМІЯ»

Кафедра мережних технологій факультету інформатики

ЗАТВЕРДЖУЮ

Завідувач кафедри мережних технологій,

Малашонок Геннадій Іванович

_____

(підпис)

«____» _____ 2021 р.

ІНДВИВІДУАЛЬНЕ ЗАВДАННЯ

на курсову роботу

студентки 3 курсу факультету інформатики

Лукічової Світлани Олександрівни

Тема: Розробка веб-застосування для підвищення продуктивності організації роботи

Зміст текстової частини до курсової роботи:

- Календарний план
- Анотація
- Вступ
- Розділ 1. Аналіз предметної області. Постановка завдання курсової роботи
- Розділ 2. Теоретичні відомості
- Розділ 3. Програмна реалізація
- Висновки
- Список використаних джерел
- Додатки

Дата видачі «____» _____ 2021 р.

Керівник _____

(підпис)

Завдання отримала _____

(підпис)

Тема: Розробка веб-застосування для підвищення продуктивності організації роботи

Календарний план виконання курсової роботи:

№	Назва етапу курсової роботи	Термін виконання етапу	Примітка
1	Отримання теми курсової роботи	10.11.2020	
2	Пошук літератури за темою роботи	21.11.2020	
3	Ознайомлення з літературою	29.11.2020	
4	Вивчення існуючих аналогів	12.12.2020	
5	Вивчення TypeScript	24.12.2020	
6	Вивчення Java Spring Boot	15.01.2021	
7	Вивчення React	05.02.2021	
8	Проектування бази даних	10.02.2021	
9	Початок створення практичної частини курсової роботи	01.03.2021	
10	Початок створення текстової частини курсової роботи	05.04.2021	
11	Завершення написання практичної частини курсової роботи	02.05.2021	
12	Завершення написання текстової частини курсової роботи	15.05.2021	
13	Створення презентації	16.05.2021	
14	Захист курсової роботи	Після 24.05.2021	

Зміст

Перелік умовних позначень

1. Id – ідентифікаційний код
2. Валідність – правильність, достовірність
3. БД – база даних
4. JS – JavaScript
5. HTTP - HyperText Transfer Protocol
6. MPA - Multi Page Application
7. СКБД – система керування базою даних
8. JWT – JSON Web Token

Анотація

Дана курсова робота присвячена створенню веб-застосування, що покращить продуктивність організації роботи користувачів. Вона базується на підході до організації справ із назвою «Getting Things Done», автор Девід Аллен. Застосунок розроблений за допомогою мови програмування TypeScript, із використанням бібліотек React, Redux та інших, з метою виконання різних функцій в системі. Серверна сторона застосунку створена із використанням Java Spring. Використовувалася база даних Postgres. Текстова частина курсової роботи містить три розділи: перший описує предменту область та постановку задачі, другий містить теоретичні відомості, третій – опис програмної розробки застосунку та результатів його роботи.

Вступ

Наше сьогодення диктує нові правила для успішного життя, і максимальна продуктивність – одне з них. Для досягнення успіхів у професійних чи особистих справах важливо бути свідомим щодо власної ефективності та розподілу ресурсів: часу, сил, енергії. Дійсно, для сучасної людини ефективно організовувати свою роботу, пам’ятати про всі справи та зберігати високий рівень продуктивності грають важливу роль. Але зі збільшенням кількості інформаційних ресурсів та завантаженості людей, зберігати всі необхідні проблеми, події, дзвінки тощо в пам’яті стає складно та енергозатратно. Нерідко під натиском великої кількості справ, люди втрачають увагу, їх ментальний стан та загальний настрій погіршується. Тому, корисним методом обробки та збереження інформації стає застосунок для підвищення продуктивності організації роботи людини. Із його допомогою стає можливо візуалізувати усі нагальні та майбутні справи, зберегти важливу інформацію, спланувати час та ефективно розподілити ресурси. Людині більше не потрібно пам’ятати та постійно обробляти великі обсяги даних в пам’яті – усе необхідне для організації роботи збережене в застосунку й може легко використовуватися в будь-який час.

Крім цього, популярність особливих технік та підходів по продуктивнії організації роботи постійно зростає. Сучасні люди можуть обрати спосіб підвищення продуктивності праці, що підходять саме їм, й тим самим знайти впевненість в собі, своїх можливостях, позбавитися страху братися за великі проєкти. У світ вийшло багато книжок, праць та методик відомих людей на тему особистого розвитку, тому кожен може обрати ідеальну техніку для себе. Одна з найвідоміших технік – Getting Things Done, опублікована в книзі бізнес-тренера Девіда Аллена. Вона

набула широкого розгласу й займає місце серед найефективніших методик. Саме цей підхід став підгрунтям та теоретичною основою даної роботи.

Метою даної роботи є вирішення проблеми недостатньої ефективності організації роботи людей та підвищення їх особистої продуктивності, а саме – забезпечення суспільство системою, де усі нагальні задачі, плани та цілі зможуть бути збережені та ефективно оброблені. Дана програма допоможе користувачу систематизувати та візуалізувати власне навантаження, ефективно розподілити задачі по категоріям, що сприятиме покращенню організації його життя. Застосунок має забезпечити користувача зручним та інтуїтивно зрозумілим інтерфейсом, усіма базовими необхідними функціями для збереження та обробки даних (задачі, нотатки, календар тощо), швидким та надійним доступом до інформації та необхідною надійністю системи.

Головними технологіями при виконанні роботи були Java Spring на серверній стороні та TypeScript на клієнтській стороні застосунку, адже вони забезпечують розробника широким спектром можливостей для зручної побудови програми. Для збереження даних була використана база даних Postgres.

Дана робота складається зі вступу, трьох розділів та висновків.

Перший розділ присвячений опису предметної області та аналізу існуючих аналогів застосунку. Визначена постановка завдання курсової роботи.

Другий розділ містить теоретичну інформацію, що була використана при розробці системи.

Третій розділ присвячено опису програмної реалізації застосування, а саме технічну завдання, структура та функціональність системи, практичне використання.

РОЗДІЛ 1. Опис предметної області. Постановка завдання курсової роботи.

1.1 Аналіз сучасного стану питання та обгрунтування теми

Продуктивність – здатність ефективно виконувати необхідну роботу та досягати поставлених цілей у вказані сроки. [1] Це якість, до якої прагнуть усі люди, що бажають бути успішними, адже вона визначає темп особистісного чи професійного зростання. У сучасному світі висвітлювати, якого людина досягла успіху, чого навчилася, які свої цілі досягла, стало нормою. Кожного дня у соціальних мережах ми дивимося на своїх друзів чи на відомих людей, що постійно досягають нових висот: нові онлайн-курси, тренінги, освіта тощо. Це мотивує не стояти на місці, постійно вдосконалюватися й ставити нові масштабні цілі. Але зі збільшенням задач, запланованих зустрічей, термінів виконання цілей стає все складніше утримувати цю кількість інформації в голові. Безліч справ не гарантує успішне їх виконання та високу продуктивність. Навпаки, втрачається фокус, нагальні справи забуваються, людина перебуває у постійному стресі через необхідність все пам’ятати та через страх забути щось надважливе.

Тож, у суспільства зростає необхідність у використання ефективного способу організації роботи. Потребують місце, де всі справи, задачі, цілі будуть надійно збережені та знаходитимуться у швидкому доступі. Також, є переваги у використанні конкретної, завідома ефективної системи, наприклад Getting Things Done, що зможе зробити процес організації роботи ще швидшим та зручнішим. Веб-застосування, що покращують продуктивність користувачів одні з найактуальніших та найнеобхідніших в сучасному світі.

1.2 Предметна область

Предметною областю даного веб-застосування є задачі, плани та цілі користувача, що можуть бути збережені та оброблені через процес їх сортування за категоріями, типами, термінами виконування, актуальністю тощо. Кожна задача повинна мати назву, категорію та критерій, чи виповнена вона. Опціональними ознаками можуть бути: опис задачі, термін виконання, належність до проєкту, відповідальна за задачу людина.

Крім цього, предметна область спирається на певний підхід до організації задач, відомий завдяки книги бізнес-консультанта Девіда Аллена – «Getting Things Done». Основні методи та функціональність підходу є висвітленими у веб-застосуванні, що є результатом даної курсової роботи. Користувачу застосунком надається можливість випробувати на собі ефективність даної методології організації роботи. Крім цього, у системі впроваджена можливість, за бажанням, відійти від чіткого алгоритму методології та створювати власні списки задач, що розширює варіанти використання застосунку та його переваги у покращенні продуктивності.

1.2.1 Підхід Девіда Аллена «Getting Things Done»

Getting Things Done – загальновідома методологія для організації та контролю задач. [2] Це практичний інструмент, що сприяє вільному від стресу, результативному росту особистості людини. Головна мета системи – звільнити думки людини від його задач та зберегти їх в одному місці на зовнішньому носії. Методика включає в себе 5 етапів: збір, обробка, організація, огляд та виконання.

1. Збір.

   На цьому етапі головна мета – зібрати якомога більше задач та зафіксувати в одному місці. Це дозволить не загубити чи забути потенційно важливі справи, а також надати важливість тим

задачам, що людина могла ігнорувати довгий час. У результаті, буде проведено мозговий штурм та зібрано усю інформацію про справи людини на фізичному носії.

2. Обробка.

   Цей крок передбачає уважний та ретельний перегляд кожної справи із загального списку для визначення її категорії. Автор методики пропонує відповісти на три питання: «Що це за задача?», «Чи можна з задачею щось зробити?» та «Чи є ця задача проектом?». У результаті, кожний елемент зі списку проходить обробку та для нього визначається необхідна категорія. Даний підхід було зображено у вигляді блок-схеми, її зображення можна побачити у додатку А, рис. А.1.

   Якщо на питання «Чи можна з задачею щось зробити?» відповідь є негативна, то є три варіанти категорії для неї:

   «Сміття» («Trash bin»). Потребує бути викресленою зі списку та забутою

   «Нотатка» («Note»). Категорія для задач, що не потребують виконання, але є важливі для користувача системою. Необхідно зберігати дані задачі в окремому місці, щоб за потребою, швидко звернутися до них для отримання інформації.

   «Знадобиться потім» («Someday-Maybe»). У дану категорію потрапляють задачі, які потенційно цікавлять користувача, але на даному життєвому етапі він не може нічого зробити для їх виконання. Прикладом можуть бути задачі «Вивчити іноземну мову» або «Поїхати закордон» - вони в певному сенсі важливі для людини, але в даний період часу вона не має достатньо коштів або можливостей для їх виконання. Немає жодної дії, яку б людина могла зробити для досягання цих цілей. Тому є сенс в тому, щоб зберегти дані задачі для подальшого перегляду.

Якщо на питання «Чи можна з задачею щось зробити?» відповідь позитивна, необхідно визначити наступну конкретну дію для її виповнення. Для наступного кроку існують категорії:

«Зробити». Сюди потрапляють задачі, що можуть бути виконані за термін менше, ніж 2 хвилини. Так як вони займають досить мало часу, є сенс не відкладати їх на потім, а виконати одразу.

«Делегувати» («Waiting-For»). Категорія для задач, які не потребують особистої присутності користувача, а можуть бути виконані іншою людиною. Справи даного типу необхідно делегувати надійній людині та вказати термін їх виконання.

«Відкласти» («Calendar»). Категорія, в яку попадають задачі, які неможливо виконати в даний момент часу або ж вони грають роль тільки у певний час. Прикладом такої задачі може бути «Подзвонити другу в понеділок». Виконати її можна тільки у вказаний день, тому є необхідність занести її в певний «Календар», де користувач згадає про неї в необхідний час.

Питання «Чи є ця задача проєктом?». Користувачу необхідно запитати себе, чи може дана задача бути виконана за одну дію. Якщо для цього необхідно виконати декілька послідовних кроків, задача отримує категорію «Проєкт» («Project»). Вона має зберігатися в окремому документі для проєктів.

Після цього користувач має визначити наступний крок для виконання проєку. Ця дія записується в окремий список, який має назву «Конкретні наступні кроки» («ASAP»). Задачі з цього списку мають виконуватися, як тільки у користувача з’являється вільний час. Вони мають однакову цінність й мають бути виконані почергово.

На етапі Обробки необхідно провести аналіз для всіх задач без виключення й не залишати задачі без категорії.

3. Організація. На цьому етапі необхідно помістити всі задачі певних категорій у відповідні їм місця. Таким чином задачі, які мають відбутися в певний день мають бути занесені в календар, задачі-проєкти поміщаються в окремий список проєктів, нотатки в список довідкової інформації тощо.
4. Огляд. Даний етап вимагає регулярного огляду нагальних справ та цілей. Користувачу пропонується переглядати усі списки раз в період, наприклад, кожного тижня. Таким чином він буде залишатися в курсі своїх задач, матиме змогу додавати та обробляти нову інформацію, а також організуватиме вже відомі цілі та проєкти.
5. Виконання. Важливий критерій успіху на цьому етапі – виконання усіх попередніх. Після вдалого збору інформації, її обробки, організації та регулярного огляду, необхідно виконати реальні дії для досягання своїх цілей. За допомогою організованих списків задач за категоріями, користувач залишатиметься свідомим щодо свого навантаження, зможе швидко братися за виконання своїх задач й досягати бажаних результатів.

Загалом, система «Getting Things Done» надає її користувачам ефективний спосіб візуалізації задач, цілей та планів, а також систему їх сортування задля результативного та швидкого виконання. Вона звільняє людину від обов’язку зберігати всі нагальні справи у пам’яті й тим самим допомагає створити комфортне, спокійне та продуктивне середовище. Із її допомогою, користувач концентрується безпосередньо на виконанні своїх справ, а не на постійному зберіганню їх в пам’яті та циклічній несвідомій обробці.

1.3 Аналіз існуючих аналогів

Як було описано в розділі «Аналіз сучасного стану питання та обгрунтування теми», продуктивність – актуальна тема сьогодення. Так як люди прагнуть покращити ефективність своєї роботи, із часом на ринку з’являється все більше різноманітних застосунків для цього. Для аналізу я обрала три застосунки, що найбільш частіше зустрічаються у списках типу «Топ найкращих застосунків для підвищення продуктивності».

Todoist

За моїми спостереженнями, Todoist (рис. 1.1) [3] – один із найпопулярніших застосунків для підвищення продуктивності.

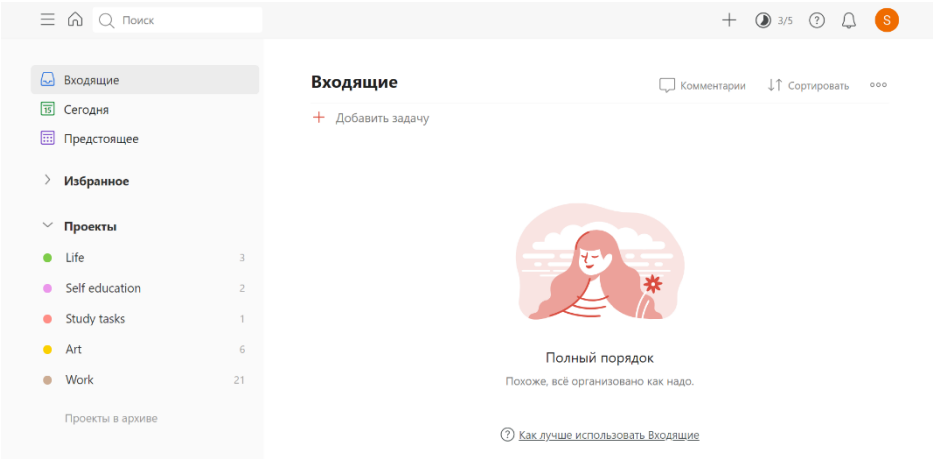

Рисунок 1.1 - Головна сторінка застосунку Todoist

У своїй статті «The 25+ Best Productivity Apps in 2021» [4] Ренсом Паттерсон називає його найкращим застосуванням для створення та роботи зі списками задач. Він виділяється простим інтерфейсом, зручним та базовим необхідним функціоналом. Дозволяє створювати задачі та проєкти з підзадачами, різноманітно редагувати їх. Серед переваг я виділила лаконічний та мінімалістичний дизайн, що є інтуїтивно зрозумілим користувачу, а також обмеженість функціоналу базовим набором, який зазвичай є найбільш використаним.

Серед недоліків:

1) Після виконання задачі, вона зникає зі списку. У разі випадкового натискання користувачем відмітки «виконано», немає можливості повернути задачу до списку.
2) Неможливо зобразити календар у вигляді «на місяць» та «на день». Також немає можливості передивитися список усіх задач із вказаним терміном виконання.
3) При створенні великої кількості «Проєктів» – списків задач, буде важко швидко знайти потрібний, адже всі вони зберігаються у боковому меню, немає окремої сторінки.
4) Немає статистики за інформацією, скільки всього задач виконано та скільки залишилося. Немає візуального представлення статистики.

Notion

Ще один застосунок, який користується великою популярністю – Notion (рис. 1.2) [5].

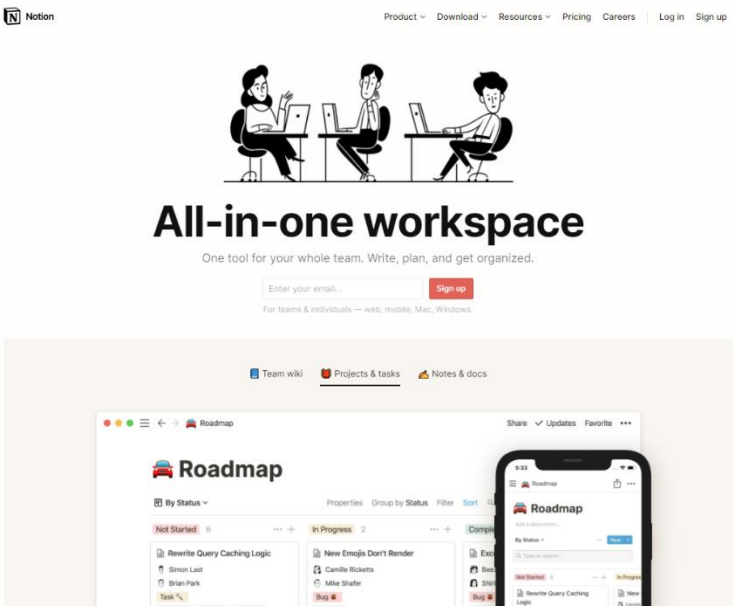

Рисунок 1.2 – Головна сторінка сайту Notion

У тій же статті [4] Ренсом Паттерсон відгукується про даний застосунок, як про систему, що має дуже багато функцій. І це справді так: у веб-застосуванні зібрані, певно, усі можливі функції, які користувач, що хоче організувати своє життя, міг тільки уявити. Наприклад, можна створювати нотатки, списки задач, списки книг, фільмів, цілей, записувати власні думки, планувати поїздки тощо. Проте я не впевнена, чи є це абсолютною перевагою застосунку. Із свого досвіду, хочу вказати, що Notion є дуже складним до опанування. Для того, щоб повністю зрозуміти функціонал системи, необхідно витратити час на дослідження кожної функції, а також вивчення документації. На рис. 1.3 зображений неповний список запропонованих функцій у даному застосуванні. Крім цього, через велику кількість іконок, різних стилів тексту, додаткових написів, різних типів контенту (текст, відео, вкладенні статті, посилання тощо), погіршується сприйняття системи, розуміння інформації та опанування застосунку. У своєму проєкті я не буду додавати велику кількість відволікаючих факторів, а також не буду перевантажувати інтерфейс, та систему загалом, надлишковим функціоналом.

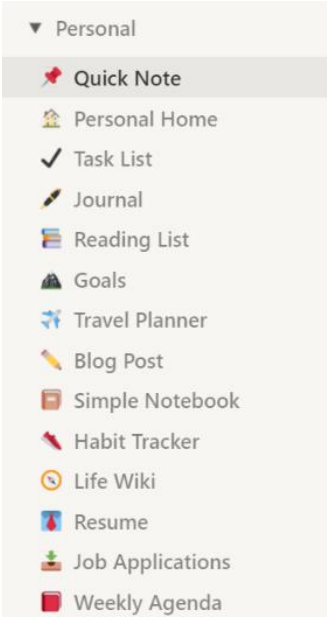

Рисунок 1.3 - Запропоновані функції застосунку Notion

Focuster

Focuster [6] – веб-застосунок для покращення продуктивності, що має приємний та досить мінімалістичний дизайн (рис. 1.4).

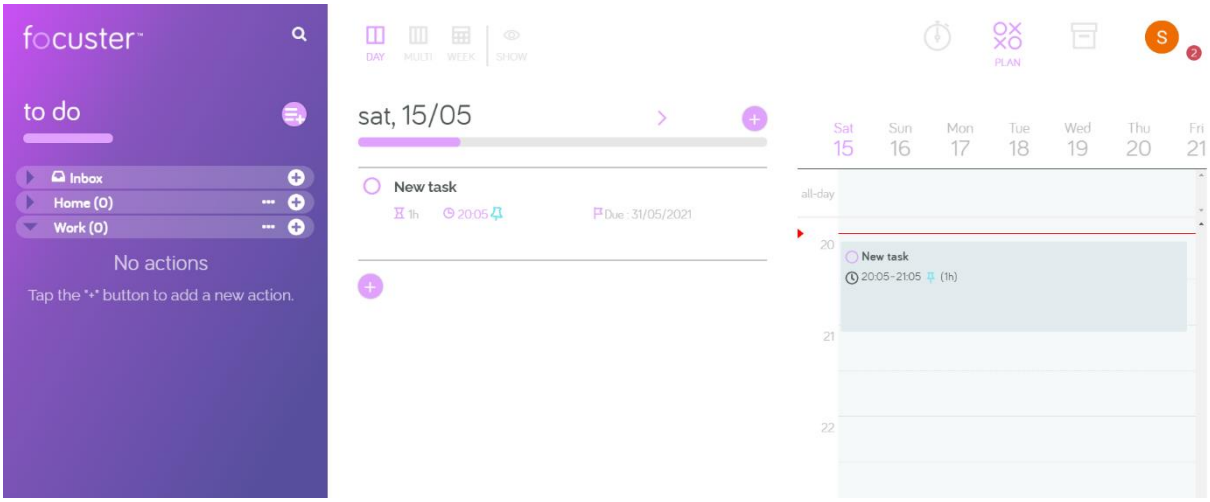

Рисунок 1.4 – Головна сторінка застосунку Focuster

Дане застосування може використовуватися для впровадження системи «Getting Things Done»: є можливість додавати задачі до проєктів, вказувати термін виконання, переглядати завдання у календарі тощо. Інтерфейс користувача розділений на три секції: зліва розташовані списки задач із підзадачами, по центру – нагальні задачі, та справа – календар. Загалом, таке розбиття екрану є незвичною практикою та сприяє гарному розподілу функціоналу. Із особливостей та переваг також можу визначити можливість додавати до кожної задачі час, за який вона може бути виконана.

Серед недоліків даного застосунку:

1) Після створення нової задачі одразу вказується термін її виконання – за годину після часу створення. Це не досить зручно, адже бувають випадки, коли необхідно створити задачу без терміну виконання. Для цього треба самостійно прибрати його з задачі.
2) Немає можливості переглядати календар у вигляді «на тиждень», «на місяць». Також немає можливості передивитися список усіх задач із вказаним терміном виконання.
3) Неможливо подивитися у календарі дати, попередні від поточної, а також після поточного тижня.
4) Неможливо додавати задачі безпосередньо до календаря.

Отже, зважаючи на переваги та недоліки існуючих аналогів, можу зробити висновок, що веб-застосунок даного типу повинен мати зручний, інтуїтивно зрозумілий інтерфейс, який легко опанувати; можливість перегляду календаря у різних варіантах та додавання задач безпосередньо до нього; не має бути перевантажений надлишковим функціоналом. Крім цього, при виконанні задачі, вона не повинна одразу зникати зі списку, щоб надати можливість користувачу відмінити дію. Має містити візуальне представлення статистики продуктивності.

1.4 Постановка завдання курсової роботи

Результатом виконання даної курсової роботи має бути проведена робота по створенню веб-застосування, що буде, за допомогою унікальної методології та додаткового функціоналу, підвищувати продуктивність організації роботи користувача.

Необхідно визначити функціональні вимоги, підготувати прототипи веб-застосування, а також провести аналіз існуючих систем на виявлення особливостей їх реалізації та недоліків.

Дослідити та визначити найефективніші та найдоцільніші методи та технології, що допоможуть впровадити у систему весь необхідний функціонал та вимоги.

Розробити веб-застосування, що задовільнятиме наступним вимогам.

Ключовими модулями системи є клієнтська та серверна частини клієнт-серверного застосунку.

Клієнтська частина надає зручний та інтуїтивно зрозумілий інтерфейс для швидкого опанування користувачем. Також, впроваджено функціонал для з’єднання та взаємодії із серверною стороною.

На серверній частині реалізовано автентифікацію та авторизацію користувачів, надійне зберігання їх даних у базі даних, функціонал для з’єднання та взаємодії із клієнтською стороною.

Функціональні вимоги до веб-застосунку:

1) Реєстрація та авторизація користувачів.
2) Перегляд, створення, редагування та видалення задач.
3) Перегляд, створення, редагування та видалення нотаток.
4) Перегляд, створення, редагування та видалення власних списків задач.
5) Сортування задач із категорії «Різне» («Things») до інших доступних категорій, в залежності від значення задачі.
6) Перегляд календаря, зміна типу перегляду: за місяць, тиждень, день.
7) Перегляд інформаційної довідки про систему.
8) Перегляд статистики про продуктивність користувача.

РОЗДІЛ 2. Теоретичні відомості

2.1 Різниця між сайтом та веб-застосуванням

Коли є мета створити деякий продукт, варто визначити, якого типу він має бути. Для початку, варто розрізняти сайт та веб-застосування.

Сайт – певний статичний ресурс, який складається з файлів, що повертаються користувачу у відповідь на запит. В основі лежить технологія HyperText Markup Language (HTML), яка дозволяє будувати розмітку даних на сторінках ресурсу. Для визначення зовнішнього вигляду використовуються Cascading Style Sheet (CSS). Також для забезпечення деякої анімації або реакції на дії користувача застосовують можливості мови JavaScript (JS). Окрім зазначених розширень, можна додавати зображення та інші файли.

Веб-застосування характеризується наявністю деякої логіки. В них так само застосовуються технології HTML, CSS, JS, із однією відмінністю – контент не є абсолютно статичним. Тобто сторінки можуть генеруватися динамічно, інформація про користувача може зберігатися до бази даних тощо. Веб-застосування характеризується більш складною структурою та більш розширеною функціональністю.

2.2 Multi Page Application

Одним із типів веб-застосувань є Multi Page Application (MPA) [7]. Головною особливістю його є те, що для різних частин проекту існують окремі сторінки. Наприклад, якщо при натисканні на кнопку меню або при підтвердження форми спостерігається завантаження браузером окремої сторінки із сервера – це ознака даного типу застосувань.

Серед переваг багатосторінкових застосувань є простота для користувачів. При використанні легко спостерігається успіх чи невдача якоїсь дії, очевидними є переходи між різними частинами ресурсу. Також

такі проекти легко оптимізуються для пошукових систем, адже їхній вміст придатний для сканування та наповнений необхідними ключовими словами.

Недоліком є сильна зв’язність логіки на стороні сервера та того, що отримує користувач в якості відповіді. В такому разі сервер не тільки відповідає за логіку, а ще й формує зовнішній вигляд сторінки та наповнює її вмістом. Звідси випливає неможливість перевикористання логіки сервера для інших платформ.

Для реалізації MPA часто використовуються шаблонізатори. Це спеціальні утиліти, які дозволяють динамічно генерувати відповідь для клієнта та надають певні абстракції для побудови компонентів сторінки. Крім них, у проектах зазвичай існує база даних для зберігання інформації про користувачів, а також інших даних, пов’язаних із застосуванням.

2.3 Single Page Application

Іншим сучасним підходом до проектування без-застосувань є Single Page Application [7]. Його особливість полягає в тому, що клієнт один раз отримує HTML сторінку, а весь контент динамічно генерується в її межах. Усі продукти зі складним інтерфейсом побудовані за даним принципом, адже потребують частих змін в зовнішньому вигляді.

Великою перевагою є гнучкість інтерфейсу. Можна визначати складні компоненти для роботи з даними. Також такі застосування характеризуються більшою швидкістю роботи, адже дані про сторінки завантажуються один раз.

Щоправда, постає більше вимог до обчислювальної здатності клієнта, адже такі сторінки виконують більше дій для коректного відображення інформації, замість того, аби просто показувати дані із серверу. Іншим

недоліком є погана пошукова оптимізація, оскільки дані генеруються динамічно та неможливо сканувати вміст компонентів проекту окремо.

Зазвичай, для застосувань такого типу використовуються фреймворки JS, які полегшують роботу із змінами стану сторінки. Спілкування із сервером відбувається у більшості випадків за протоколом HTTP для отримання та відправки даних.

2.4 Клієнт-серверна архітектура

Однією з найвідоміших та найпопулярніших архітектур для побудови веб-застосувань є клієнт-сервер [8]. Як відомо, дана концепція включає в себе наступні компоненти:

- Сервер – взаємодіє з базою даних, надає інформацію та послуги, якими користуються програми;
- Клієнт – надсилає запити на сервер, щоб отримати доступ та використати послуги, надані ним;
- Мережа – забезпечує взаємодію між клієнтом та сервером за допомогою запитів та відповідей.

Рисунок 2.1 - Архітектура застосунку

Архітектура застосунку зображена на рис. 2.1. Взаємодія між клієнтом та сервером відбувається за допомогою HTTP-протоколу. Клієнтська частина надсилає REST-запити GET для отримання даних з серверу, POST для додавання, PUT для оновлення та DELETE для видалення даних.

Використання клієнт-серверної архітектури, а саме – розділення інтерфейсів на клієнтів та серверів, має низку переваг для побудови застосування. По-перше, усі дані зберігаються в одному місці – на сервері,

що покращує їх захист та полегшує процеси управління наданням доступу до них. По-друге, зменшуються вимоги до комп’ютерів користувачів, так як усі необхідні обчислення відбуваються на стороні сервера. По-третє, відсутність дублювання коду серверних програм клієнтськими програмами. Сюди можна віднести також легкість масштабування систем та факт, що сервер можна використовувати для клієнтів різних платформ, тобто, є перспективи для модернізації застосунків під, наприклад, мобільні телефони.

РОЗДІЛ 3. Програмна реалізація

3.1 Аналіз технічного завдання

Головна роль у системі – користувача.

Технічні вимоги користувача до функціональності системи:

1. Реєстрація
    a. Наявність полів: логін, пароль, ім’я, прізвище, електронна пошта.
    b. Динамічна валідація полів при заповненні форми. Виведення підказок користувачу у разі неправильно заповнених полів. Наприклад, якщо пароль занадто короткий, користувач бачить відповідне повідомлення. Кнопка «Зареєструватися» має бути доступною тільки після введення коректних даних.
2. Автентифікація.
    a. Наявність власних логіну та паролю у зареєстрованого користувача для входу в систему та доступу до своїх даних.
3. Перегляд нагальних задач – тих, в яких термін виконання збігає на наступний день після поточної дати та раніше.
4. Перегляд задач за категоріями.
    a. Наявні категорії в системі: «Things», «ASAP», «Projects», «Calendar», «Someday-Maybe», «Notes», «Waiting-For».
5. Додавання задач у кожну категорію.
    a. В залежності від категорії, користувач бачить відповідні поля для створення нової задачі.

    Поля у категорії «Things»: назва задачі.

    Поля у категорії «ASAP»: назва задачі, назва проєкту, до якого відноситься задача.

Поля у категорії «Projects»: назва, термін виконання (необов’язково), опис (необов’язково).

Поля у категорії «Calendar»: назва, термін виконання.

Поля у категорії «Someday-Maybe»: назва.

Поля у категорії «Notes»: назва, опис.

Поля у категорії «Waiting-For»: назва, відповідальна людина, термін виконання (необов’язково), назва проєкту, до якого відноситься задача (необов’язково), опис (необов’язково).

6. Редагування задач у певній категорії.
    a. В залежності від категорії, користувач бачить відповідні поля для редагування задачі.
7. Видалення задач.
8. Відмітка задачі як виповненої.
9. Видалення всіх виповнених задач.
10. Сортування задач із загального списку – вибір певної категорії для кожної задачі.
    a. Користувачу надається можливість обрати одну з доступних категорій для кожної задачі. Після заповнення необхідних для обраної категорії, задача переміщується до відповідного списку.
11. Перегляд календарю на місяць, на тиждень, на день.
12. Перегляд усіх задач із терміном виконання.
13. Перегляд інформаційної довідки про систему.
14. Перегляд статистики продуктивності користувача.
    a. Кількість задач у кожній категорії.
    b. Кількість виповнених та невиповнених задач.
15. Створення власних списків задач.
    a. Поле: назва списку.

16. Перегляд, створення, редагування, видалення задач у власних списках.
17. Вихід із системи.

3.2 Обгрунтування вибору засобів розробки

3.2.1 Typescript

TypeScript – відома з 2012 року мова програмування, що є надбудовою над JavaScript [9]. Це означає, що код, написаний на даній мові, – в файлах .ts (або .tsx для JSX), – не може бути одразу виконаний в браузері, як JavaScript. Замість цього, він проходить транспіляцію – ще один крок компіляції, що перетворює код на JS-еквівалент. Валідний код, написаний на JavaScript, також буде виконаний у TypeScript [10].

Визначною особливістю TypeScript є строга типізація – визначення типу змінної при її оголошенні. Типи дають спосіб описати форму та інтерфейс об’єкта, забезпечуючи кращу документацію та дозволяючи TypeScript перевірити, чи правильно працює ваш код. Серед переваг строгої типізації: можливість знаходити помилки при написанні програми на етапі розробки; більш точна передача змісту функцій – типи даних, що вони приймають та які повертають; більша передбаченість виконання коду. При цьому, є можливість використовувати динамічну типізацію, як в JavaScript. У TypeScript існує декілька типів даних:

- Number – числові значення
- String – текстові значення
- Boolean – булеан
- Any – значення будь-якого типу
- T[] – масив значень заданого типу T
- Void — використовується лише для функцій і вказує, що вона не повертає ніякого значення

Крім цього, є можливість створювати власні типи. Для визначення структури власного типу необхідно користуватися інтерфейсами. У полях в інтерфейсі можна вказувати обов’язкові та необов’язкові поля. Інтерфейси можна розширювати, таким чином наслідуючи властивості іншого типу. Після компіляції в JS інтерфейси повністю зникають — вони створені лише для полегшення розробки.

Схожу роль грають перелічення (enums) – дозволяють визначати набір пов’язаних констант як частину одної сутності.

У TypeScript можна визначати загальні компоненти, стани та властивості (props), для того щоб повторно використовувати для різних типів даних.

Дані особливості мови TypeScript допомогають полегшити сприймання коду та перевикористання компонентів, підвищити швидкість розробки та допомогти визначати помилки на етапі розробки. Саме тому цю мову програмування було обрано для створення клієнтської частини веб-застосунку курсової роботи.

3.2.2 React

Бібліотека React використовується для створення компонентів користувацького інтерфейсу, придатних для ефективного перевикористання. [11] Ключовим є поняття реактивності, яке полягає у тому, що сторінка веб-застосунку у браузері відображає оновлення даних без перезавантаження. Це надає можливість застосуванню швидко реагувати на зміни, що вводить користувач, та реалізовувати функціональні вимоги будь-якого рівня складкості.

React є зручним рішенням при створенні веб-застосунків, придатних до масштабування. Його мета – мінімізувати помилки та надлишковий код, завдяки багаторазовому використанню компонентів. Компонент – окремий блок коду, що є логічно виокремленим та описує частину

користувальницького інтерфейсу. Він має певні параметри для відображення, а також власний стан, при зміні яких, викликається повторно процедура відображення з новими даними. Загальний інтерфейс застосування створюється з поєднання усіх компонентів та створення їхньої ієрархії: їх можна вкладати одне в одного.

При розробці на React використовується JSX (TSX) – розширення синтаксису JavaScript (TypeScript), який дозволяє визначити композицію компонентів через XML-подібний синтаксис.

Завдяки попередній компіляції коду, кінцеве застосування характеризується доволі високою ефективністю та забезпечує обернену сумісність браузерів, надаючи можливість використовувати сучасні можливості мов JavaScript або TypeScript. Наприклад, можна вільно використовувати анонімні функції для полегшення сприйняття там, де це доцільно, незважаючи на те, що не всі старі браузери підтримують їх, адже компілятор замінить їх на звичайні сигнатури, якщо вказати такі вимоги у конфігураціях.

Дані переваги використання повпливали на мій вибір бібліотеки для створення якісного й зручного у використанні та розробці користувальницького інтерфейсу.

3.2.3 Redux

Бібліотека Redux впливає на керування станом веб-застосунку. Він використовується при розробці складних застосувань, де керувати станом за допомогою стандартного менеджеру станів React неможливо. У Redux існують функції, що відповідають за оновлення загального стану – reducers. Після отримання запиту на оновлення стану, кожна з цих функцій обробляє його, формуючи новий стан. Таким чином можна перенести складну логіку із складних компонентів, в окремі файли. [12]

Дана бібліотека забезпечує глобальний стан застосунку та доступ до нього з будь-якого компоненту проекту. Це може бути корисним для кешування даних, коли компонент видаляється з контексту, а також коли інформація використовується в декількох частинах застосунку. В даному проекті бібліотека Redux використовується для зберігання інформації про поточного користувача.

3.2.4 База даних

Для збергіння інформації на сервері необхідно мати базу даних, яка буде надавати можливість виконувати вибірку, оновлення, створення та видалення інформації. Для виконання було обрано реляційну модель для підтримки цілісності та зручного виконання структурованих запитів за допомогою мови SQL.

Серед багатьох доступних систем керувань базою даних (СКБД) було обрано PostgreSQL, оскільки вона пропонує високу ефективність, виконання усіх стандартів та доступну документацію. Це серверна СКБД, тому для роботи потрібно запустити на цільовій операційній системі необхідний процес, а також вказати сокетний порт, ім’я та пароль користувача [13].

3.2.5 Java Spring Boot

В якості технології для реалізації серверної частини застосунку використовується об’єктно-орієнтована мова програмування Java разом із фреймворком Spring Boot, який надає сучасні можливості для створення програмних застосунків.

По-перше, він вмістить вбудований багатопоточний веб-сервер та утиліти для конвертації даних між JSON та Java об’єктами, що дозволяє створювати RESTful застосування з мінімальними зусиллями. По-друге, його архітектура побудована за принципом Inversion of Control (IoC) та

підтримує так званий контекст, де об’єкти створюються автоматично, таким чином, від програміста вимагається лише визначати інтерфейси та класи, а за їхню конфігурацію та зв’язування відповідає Spring Boot. По-третє, зручним є додавання залежностей та стартерів (певних наборів сконфігурованих залежностей), адже проєкти з його використанням підтримують популярні системи збирання проектів, як-от Maven чи Gradle.

В межах проекту застосовується принцип Object Relaction Mapping (ORM), який спрощує роботу із базою даних та надає зручний інтерфейс до неї через описані сутності та об’єкти замість прямої взаємодії з рядками бази даних. В якості реалізації даного принципу використовується Hibernate, який є одним із найбільш популярних рішень. Сутності можна описувати за допомогою анотацій, а запити до бази даних використовують особливий діалект SQL, який часто полегшує сприйняття та зменшує обсяг запитів, особливо на з’єднаннях таблиць.

Для мінімізації шаблонного коду використовується залежність Lombok, за допомогою якої можна анотаціями створювати конструктори, селектори, модифікатори та інші стандартні методи.

3.2.6 Бібліотеки

Бібліотека Moment надає інтерфейс для роботи із датами та часом [14]. Серед корисних функцій можна виділити створення об’єкту за рядком із датою та часом за заданим форматом або за числовим значенням Unix Timestamp, а також відповідний експорт.

Axios – це HTTP клієнт, за допомогою якого відбувається взаємодія із сервером, а саме надсилаються запити до нього. В нього дуже зручно передавати метод, параметри, тіло та заголовки запиту, а також отримувати результат у вигляді Promise [15].

Дана залежність служить обгорткою для компонентів форми та її частин в React. Вона позбавляє необхідності писати однотипні функції для

обробки та передачі значень, надаючи власний компонент із потрібними визначеннями.

Бібліотека Calendar пропонує визначений компонент для візуалізації даних по дням та годинам. У якості параметрів до нього можна передавати розклад із назвами подій, а також реагувати на натискання на конкретний проміжок часу.

Chart.js є бібліотекою для візуалізації даних у вигляді графіків. Вона містить у собі декілька їхніх типів, а також надає гнучкість налаштування відображення [16].

3.3 Опис розробки програми

3.3.1 Клієнт-серверне застосування

Веб-застосування побудоване за принципом клієнт-серверної архітектури. У програмній реалізації клієнтської сторони, взаємодія з сервером відбувається наступним чином (рис. 3.1):

```
editTask: async (task: IListItem): Promise<void> => {
    await apiClient.put( url: "/api/task", task);
},

deleteTask: async (id: number): Promise<void> => {
    await apiClient.delete( url: `/api/task/id/${id}` );
},
```

Рисунок 3.1 – Фрагмент коду файлу listItemService.ts

У файлі apiClient.ts використано JavaScript бібліотеку Axios для виконання HTTP-запитів (рис. 3.2). Цей варіант має переваги над методом fetch(), адже Axios автоматично перетворює JSON-дані – немає необхідності викликати метод json() при отриманні даних, а також надає зручний інтерфейс для формування запиту [17].

```
import axios from 'axios';

const apiClient = axios.create();
```

Рисунок 3.2 - Фрагмент коду файлу apiClient.ts

Налаштування шляху (рис. 3.3) для з'єднання з серверною частиною відбувається за допомогою змінних оточення, які можна вказати в .env файлі (рис. 3.4). Так само тут показано, як до запиту додається JWT-токен, опис якого наявний далі.

```
apiClient.interceptors.request.use( onFulfilled: request => {
    const token = tokenService.getAccessToken();
    if (token) {
        request.headers.Authorization = `Bearer ${token}`;
    }
    const prefix = `${env.backendProtocol}://${env.backendHost}:${env.backendPort}`;
    if (!request.url?.startsWith(prefix)) {
        request.url = prefix + request.url;
    }
    return request;
});
```

Рисунок 3. 3 - Фрагмент коду файлу apiClient.ts

```
REACT_APP_BASE_HOST=localhost
REACT_APP_BASE_PORT=3000
REACT_APP_BACKEND_PROTOCOL=http
REACT_APP_BACKEND_HOST=localhost
REACT_APP_BACKEND_PORT=8888
```

Рисунок 3. 4 - Вміст файлу .env

На серверній стороні запити обробляються всередині контролерів, які містять декларативний опис маршрутизації за допомогою анотацій, а саму логіку виконання запитів делегують сервісам (рис. 3.5):

```java
@DeleteMapping(value = "/id/{taskId}")
public ResponseEntity deleteTask(@PathVariable Long taskId) {
    Assert.notNull(taskId,  message: "Task id cannot be null");

    taskService.deleteTask(taskId);
    log.info("Deleted tasks with id {}", taskId);

    return ResponseEntity.ok().build();
}

@PutMapping
public ResponseEntity update(@RequestBody TaskEdit taskEdit) {
    taskService.update(taskEdit);
    log.info("Updated task id {} with data {}", taskEdit.getId(), taskEdit);

    return ResponseEntity.ok().build();
}
```

Рисунок 3. 5 - Фрагмент коду файлу TaskController.java

3.3.2 Авторизація з JWT токеном

У веб-застосунку для авторизації користувачів використовується технологія JSON Web Token (JWT) – JSON об’єкт, що визначен у відкритому стандарті RFC 7519. [18] Токени є засобом авторизації для кожного запиту взаємодії між клієнтом і сервером, а також зберіганням додаткової інформації про поточного користувача.

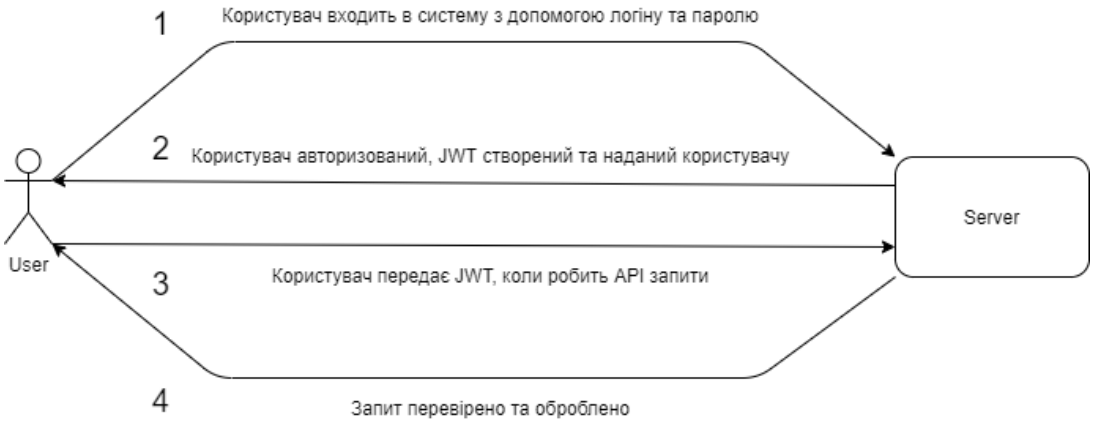

Рисунок 3. 6 - Процес використання JWT веб-застосуванням

Веб-застосування використовує JWT наступним чином:

1. Користувач входить в систему за допомогою його унікальних даних: логіну та паролю.
2. Сервер формує JWT та надає його користувачу системи.

3. Коли користувач надсилає API запити до системи, він додає до них отриманий JWT для підтвердження своєї особи та прав.
4. Серверна частина застосунку перевіряє отриманий JWT на валідність та, в разі підтвердження прав користувача на дані запити, оброблює запит. [19]

Використання JWT токенів дозволяє використовувати stateless підхід – кожен новий запит на сервер не залежить від попереднього й існує можливість ідентифікувати користувача за допомогою даних в кожному запиті [20].

Робота з токенами на клієнтській стороні представлена у файлах tokenService.ts та apiClient.ts. У першому файлі збережені функції для отримання, встановлення та видалення токену (рис. 3.7).

```
getAccessToken: (): string | null => localStorage.getItem(ACCESS_TOKEN_PROPERTY),

setAccessToken: (accessToken: string): void => {
    localStorage.setItem(ACCESS_TOKEN_PROPERTY, accessToken);
},

removeToken: (): void => {
    localStorage.removeItem(ACCESS_TOKEN_PROPERTY);
},
```

Рисунок 3. 7 - Фрагмент коду файлу tokenService.ts

Дані функції надалі використовуються:

getAccessToken – при перевірці наявності токену та додаванні його до headers запиту, у файлі apiClient.ts (рис. 3.8)

```
const token = tokenService.getAccessToken();
if (token) {
    request.headers.Authorization = `Bearer ${token}`;
}
const prefix = `${env.backendProtocol}://${env.backendHost}:${env.backendPort}`;
```

Рисунок 3. 8 - Фрагмент коду файлу apiClient.ts

setAccessToken – при входженні користувача у систему (рис. 3.9)

```
login: async (loginDto: ILoginRequest): Promise<void> => {
    const response = await apiClient.post( url: "/api/auth/signin", loginDto);
    const accessToken: string = response.data.token;
    tokenService.setAccessToken(accessToken);
},
```

Рисунок 3. 9 - Фрагмент коду файлу authService.ts

removeTokens – при визначенні токена не валідним або при виході користувача з системи (рис. 3.10 та рис. 3.11)

```
if (status !== 401) {
    return Promise.reject(new Error("Error!"));
}
tokenService.removeTokens();
```

Рисунок 3. 10 - Фрагмент коду файлу apiClient.ts

```
logout: async (): Promise<void> => {
    tokenService.removeTokens();
},
```

Рисунок 3. 11 - Фрагмент коду файлу authService.ts

На серверній частині застосунку JWT токен створюється за допомогою методу generateJwtToken() у файлі jwtUtils.java (рис. 3.12), також додається інформація про користувача, вказується кінцевий термін. Також токен шифрується секретним ключом за допомогою алгоритму HS512, що забезпезпечує надійність та унеможливлює підробку.

```
public String generateJwtToken(Authentication authentication) {

    UserDetailsImpl userPrincipal = (UserDetailsImpl) authentication.getPrincipal();

    return Jwts.builder()
            .setSubject((userPrincipal.getUsername()))
            .setIssuedAt(new Date())
            .setExpiration(new Date((new Date()).getTime() + jwtExpirationMs))
            .signWith(SignatureAlgorithm.HS512, jwtSecret)
            .compact();
}
```

Рисунок 3. 12 - Фрагмент коду файлу JwtUtils.java

Наступний метод (рис. 3.13) дозволяє взяти дані про користувача з токена, аби виконати його вибірку з бази даних.

```
public String getUserNameFromJwtToken(String token) {
    return Jwts.parser() JwtParser
            .setSigningKey(jwtSecret)
            .parseClaimsJws(token) Jws<Claims>
            .getBody() Claims
            .getSubject();
}
```

Рисунок 3. 13 - Фрагмент коду файлу JwtUtils.java

3.3.3 Клієнтська частина

Клієнтська частина веб-застосування розроблялася за допомогою мови програмування TypeScript, тому всі файли мають розширення .ts (.tsx).

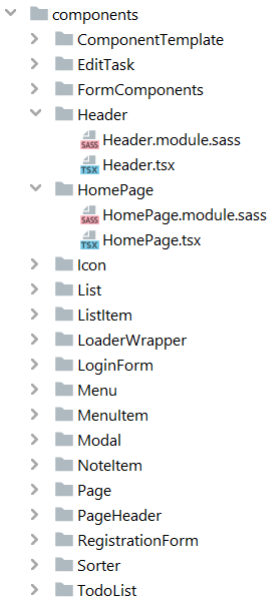

Рисунок 3. 14 - Фрагмент файлової структури клієнтської сторони

Кожен компонент розміщений у власній папці (рис. 3.14), яка містить безпосередньо файл .tsx та файл зі стилями для даного компоненту.

У проєкті використано препроцесор Sass, що полегшує роботу зі створеннями стилів для елементів. Файли мають розширення .module.sass з метою, щоб стилі елементів даного компоненту не конфліктували зі стилями інших. Таким чином, у кожному компоненті можуть бути створені елементи з однаковими класами чи ідентифікаторами.

Оскільки строга типізація є однією з особливостей мови TypeScript, в props та state кожного компоненту використовуються інтерфейси (рис. 3.15).

```
interface IOwnProps {
    item: IListItemInline
    handleEdit: (item: IInlineData) => void
    handleDelete: () => void
    handleChange: () => void
    parentEnabled: boolean
}

interface IState {
    modifying: boolean
}
```

Рисунок 3. 15 - Приклад інтерфейсу та стану

У папці **api** зберігається інформація про моделі даних та сервіси – способи взаємодії з серверною стороною (рис. 3.16).

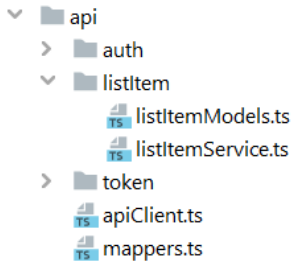

Рисунок 3. 16 - Вміст папки api

У файлах authModels.ts та listItemModels.ts знаходяться інтерфейси з даними про користувача та задачі, відповідно. Існують як звичайні інтерфейси, так і ті, що розширюють існуючі додатковими полями. Наприклад, на рис. 3.17 представлений інтерфейс IListItemInline, у якого поле deadline приймає значення числового типу або null. Інтерфейс IListItemDeadline, в свою чергу, розширює попередній, змінюючи тип поля deadline на числове, адже за логікою системи, у задач із типом «Deadline» обов’язково має бути присутнім термін виконання.

```
export interface IListItemInline {
    id: number
    name: string
    completed: boolean
    parentTask: IListItem | null
    deadline: number | null
}

export interface IListItemDeadline extends IListItemInline{
    deadline: number
}
```

Рисунок 3. 17 - Інтерфейс IListItemInline

У listItemModels.ts також зберігається перелічення з даними про типи задач (рис. 3.18). Кожній задачі в програмі має належати один з цих типів.

```
export enum MenuItemsEnum{
    THINGS= "THINGS",
    ASAP = "ASAP",
    PROJECTS = "PROJECTS",
    SOMEDAY = "SOMEDAY",
    NOTES = "NOTES",
    WAITING = "WAITING",
    DEADLINE = "DEADLINE",
    LIST = "LIST",
    TODO = "TODO"
}
```

Рисунок 3. 18 - Перелічення типів

У файлі mappers.ts знаходяться функції для обробки об’єктів. Вони використовуються з метою конвертації задачі конкретного типу до загального. Таким чином, на серверну сторону надсилаються об’єкти задач загального типу - IListItem. Наприклад, є задача типу «Project». Цей тип передбачає поля, зображені на рис. 3.19.

```
export interface IListItemProject {
    id: number
    name: string
    completed: boolean
    deadline: number | null
    description: string | null
}
```

Рисунок 3. 19 - Інтерфейс IListItemProject

У даному типі немає полей responsiblePerson та parentTask, але, щоб надіслати запит на сервер, необхідно мати об’єкт зі всіма полями: id, name, completed, deadline, description, responsiblePerson, parentTask. Отже, за допомогою наступної функції (рис. 3.20) можна звести даний об’єкт до загального типу – усі неіснуючі поля заповнюються null, а тип встановлюється відповідним значенням.

```
export const mapProjectItemToItem = (item: IListItemProject): IListItem => {
    return {
        ...nullableItem,
        ...item,
        type: MenuItemsEnum.PROJECTS,
    };
};
```

Рисунок 3. 20 - - Фрагмент коду файлу mappers.ts

Даний тип функцій використовується для того, щоб надіслати на сервер PUT запит для оновлення даних: з оновленого об’єкту задачі

інтерфейсу IListItemProject, створюється загальний об’єкт інтерфейсу IListItem.

У файлі reducer.ts зберігається інформація про користувача у глобальному Redux стані застосування, далі наведено приклад встановлення поточного користувача (рис. 3.21). Таким чином, можна отримати доступ до даних про нього з будь-якого компоненту проєкту, при відповідному запиті.

```
switch (action.type) {
    case SET_CURRENT_USER:
        return {
            ...state,
            currentUser: action.payload,
        };
```

Рисунок 3. 21 - Фрагмент коду файлу reducer.ts

Для перевірки правильності написання коду та виявлення помилок, у даному проєкті було використано інструмент Linter. Правила для нього збережені у файлі .eslintrc.json. Також, файл tsconfig.json дозволяє запобігти помилкам ще на етапі компіляції застосунку.

Для розробки клієнтської частини застосунку було обрано інтегроване середовище розробки Web Storm. Дане програмне забезпечення від компанії Jet Brains сприяло покращенню продуктивності написання програми завдяки зручним можливостям для управління багатофайловими проєктами, встановлення додаткових бібліотек та використання вбудованого контролю версій Git.

3.3.3 Серверна частина

Компоненти на стороні сервера, які відповідають за обробку запитів користувачів, діляться на три типи: контролери, сервіси та репозиторії.

Контролери містять методи, які відповідають маршрутам інтерфейсу. Для них вказується метод запиту, а також параметри і тіло, які очікуються. Кожен запит зберігає повідомлення в історії про себе, викликає метод сервісу та повертає відповідь (рис. 3.22). Ключовою особливістю є те, що

параметри в сигнатурі та тип результату є звичайними Java класами, а Spring Boot самостійно конвертує та валідує дані.

```
@PostMapping(value = "/complete")
public ResponseEntity complete(@RequestBody TaskId taskId) {
    taskService.complete(taskId);
    log.info("Completed task with id {}", taskId.getId());

    return ResponseEntity.ok().build();
}
```

Рисунок 3. 22 - Фрагмент коду файлу TaskController.java

Контролер у якості залежності має інтерфейс сервісу (рис. 3.23). Це зроблено для того, аби за принципом Dependency Inversion спиратися на інтерфейс, а не реалізацію. Немає необхідності вручну створювати об’єкти самих класів, адже Spring Boot сканує проект та автоматично надає екземпляр необхідного інтерфейсу.

```
public interface TaskService {

    List<Task> getAllTasks();

    void complete(TaskId taskId);
```

Рисунок 3. 23 - Фрагмент коду файлу TaskService.java

Що стосується самої реалізації сервісу, то вона містить основну бізнес-логіку застосунку (рис. 3.24). Тут виконуються усі перевірки на відстутність конфліктів з поточними даними, конвертація об’єктів, оновлення їхніх властивостей, а також викликаються методи репозиторію - інтерфейсу доступу до бази даних - для оновлення, створення, вибірки та видалення даних.

```
@Override
@Transactional
public void complete(TaskId taskId) {
    var id = taskId.getId();
    var task = getTaskById(id);
    task.setCompleted(!task.isCompleted());
    taskRepository.save(task);
}
```

Рисунок 3. 24 - Фрагмент коду файлу TaskServiceImpl.java

Репозиторій є інтерфейсом із описом усіх можливих операцій із даною сутністю, окрім стандартних. У більшості запитів використовується підхід, який називається Derived Query, який полягає у тому, що текст запиту будується самостійно, спираючись на ім'я методу, який має відповідати певним вимогам (рис. 3.25).

```
@Repository
public interface TaskRepository extends JpaRepository<Task, Long> {

    List<Task> findAllByTypeAndUser_Id(TaskType type, Long userId);
```

Рисунок 3. 25 - Фрагмент коду файлу TaskRepository.java

Також існують класи для опису сутностей – моделі. В них використовуються анотації Hibernate (рис. 3.26).

```
@Entity
@Table(name = "tasks")
@Where(clause = "deleted = false")
public class Task {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
```

Рисунок 3. 26 - Фрагмент коду файлу Task.java

Інші класи відповідають за конфігурацію інших частин застосунку: Spring Security (разом із JWT), обробку помилок, запуск тощо.

3.3.4 Розробка інтерфейсів

Також, для зручності розробки веб дизайну, були створені схематичні макети інтерфейсів. Вони демонструють орієнтовне розташування елементів, кольорову гамму, а також способи взаємодії користувача з системою. Зображені базові функції веб-застосунку. Переглянути макети можна у додатку Б.

3.4. Створення об’єктів і розробка головної програми

Усі маніпуляції з об’єктами протягом виконання запитів виконуються через бібліотеку Lombok, яка самостійно створює конструктори, селектори та модифікатори.

Більшість об’єктів моделей створюються за допомогою патерну Builder, реалізацію якого також надає бібліотека (рис. 3.27). Усі поля встановлюються окремими методами, а в кінці викликається термінальний метод, що повертає бажаний об’єкт. Такий підхід значно зручніший за виклик конструкторів, адже при великій кількості властивостей можна легко простежити, чому відповідає конкретне значення в будь-якому порядку, замість того, аби слідувати порядку аргументів.

```
var task = Task.builder()
        .name(taskCreate.getName())
        .type(taskCreate.getType())
        .completed(taskCreate.getCompleted())
        .responsiblePerson(taskCreate.getResponsiblePerson())
        .description(taskCreate.getDescription())
        .deadline(taskCreate.getDeadline())
        .parentTask(parentTask)
        .user(user)
        .build();
```

Рисунок 3. 27 - Фрагмент коду файлу TaskServiceImpl.java

Для встановлення нового та повернення поточного значення деякої властивості використовуються модифікатори та селектори відповідно (рис. 3.28). Вони так само генеруються бібліотекою Lombok, адже їхнє визначення є тривіальним, тому їхня явна реалізація лише збільшує обсяг програмного коду.

```
task.setDeadline(taskEdit.getDeadline());
task.setDescription(taskEdit.getDescription());
task.setName(taskEdit.getName());
task.setResponsiblePerson(taskEdit.getResponsiblePerson());
```

Рисунок 3. 28 - Фрагмент коду файлу TaskServiceImpl.java

Для створення деяких об’єктів, зазвичай із невеликою кількістю параметрів, застосовуються конструктори (рис. 3.29), визначення яких створюється відповідними анотаціями бібліотеки.

```
Long amount = taskRepository.countAllByUser_Id(userId);
Long completed = taskRepository.countAllByCompletedAndUser_Id( completed: true, userId);
var overall = new OverallStatisticsDto(amount, completed);
```

Рисунок 3. 29 - Фрагмент коду файлу TaskServiceImpl.java

Як зазначалося вище, відповідальність за створення об’єктів, які містять у собі логіку обробки запитів, або інших частин застосунку, покладається на фреймворк, а програміст декларативно визначає їхню ієрархію.

3.5 Опис бази даних та інтерфейсу програми

3.5.1 Схема бази даних

На рис. 3.30 представлена схема бази даних веб-застосування, відповідно до представлених функціональних вимог.

Рисунок 3. 30 - ER-модель

Опис таблиць та компонентів:

1. «Користувач» - таблиця, що відповідає даним про користувача системи. Містить атрибути:
   - код_користувача – первинний ключ, ідентифікаційний код користувача в базі даних, що є унікальним
   - логін – логін користувача, його унікальне ім’я в системі, обов’язковий атрибут

- пароль – пароль користувача у зашифрованому вигляді, обов’язковий атрибут
- ім’я – ім'я користувача, обов’язковий атрибут
- прізвище – прізвище користувача, обов’язковий атрибут
- ел_пошта – електронна пошта користувача, обов’язковий атрибут

2. «Задача» - таблиця, що відповідає даним про задачу користувача. Містить атрибути:
   - код_задачі – первинний ключ, ідентифікаційний код задачі в базі даних, що є унікальним
   - назва – назва задачі, обов’язковий атрибут
   - чи_виконана – чи є задача виконана, необов’язковий атрибут
   - термін_виконання – кінцевий термін виконання задачі, необов’язковий атрибут
   - батьківська_задача – батьківська задача для даної, відповідає за приналежність даної задачі до проєкту, необов’язковий атрибут
   - відповідальний – людина, що відповідальна за виконання даної задачі, необов’язковий атрибут
   - опис – опис задачі, необов’язковий атрибут

У таблиці «Задача» представлений необов’язковий рекурсивний (односторонній) зв’язок один до багатьох. Зв’язок сутності «Задача» з самою собою: один і той же тип сутності бере участь у зв’язку кілька разів, але у різних ролях. У даному випадку, сутність «Задача» може бути у ролі звичайної задачі, або ж відповідати за батьківську задачу, що може мати підзадачі. Тому присутній рекурсивний зв’язок «є батьківською задачею».

Крім цього, при створенні розширення ER-моделі (EER) було визначено суперклас та підкласи сутності «Задача» (рис). Суперклас – тип сутності, що включає одну або декілька розрізнених допоміжних угрупувань її екземплярів, які мають бути представлені в моделі даних.

Підклас – розрізнене допоміжне угрупування екземплярів типу сутності, яке має бути представлене в моделі даних.

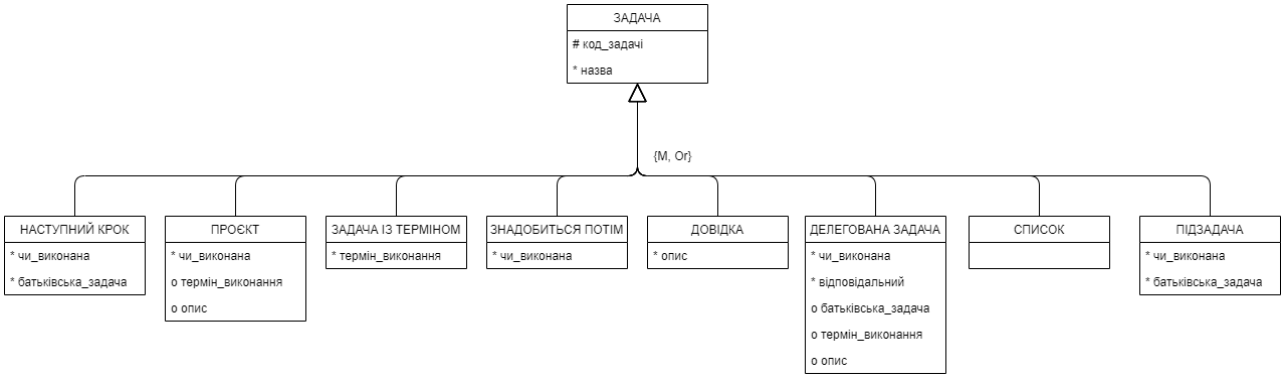

Рисунок 3. 31 - EER-модель

На даній схемі суперкласом є сутність «Задача», яка має 8 підкласів з різними атрибутами: «Наступний крок», «Проєкт», «Задача із терміном», «Знадобиться потім», «Нотатка», «Делегована задача», «Список», «Підзадача».

Тип наслідування визначений як «Mandatory, Or». Mandatory означає, що кожен об’єкт обов’язково є екземпляром одного з підкласів, Or - кожен об’єкт має відповідати лише одному підкласу.

Після побудови ER-моделі, було виконано перехід до реляційної моделі даних із дотриманням властивостей реляції.

Реляційна модель даних є логічна модель даних із відношеннями (позначають таблиці) та атрибутами (рядки).

Властивості реляції:

- унікальна назва кожного атрибуту та відношення
- порядок слідування атрибутів та кортежів у відношенні не має значення
- всі кортежі унікальні

Особливістю переходу було поєднання сутностей усіх підкласів у єдину таблицю, що містить усі доступні атрибути, а також додавання поля, яке позначає тип підкласу. Реляційна модель даних зображена у таблицях 3.32 та 3.33.

USER

PK	user_id	Autoincrement	NN	Атрибут простий обов’язковий «Ідентифікаційний номер користувача»
	username	Varchar	NN	Атрибут простий обов’язковий «Логін користувача»
	password	Varchar	NN	Атрибут простий обов’язковий «Пароль користувача»
	firstName	Varchar	NN	Атрибут простий обов’язковий «Ім'я користувача»
	lastName	Varchar	NN	Атрибут простий обов’язковий «Прізвище користувача»
	email	Varchar	NN	Атрибут простий обов’язковий «Електронна пошта користувача»

Таблиця 3.32 - Сутність "User"

TASK

PK	task_id	Autoincrement	NN	Атрибут простий обов’язковий «Ідентифікаційний номер задачі»
	task_name	Varchar	NN	Атрибут простий обов’язковий «Назва задачі»
	completed	Boolean	N	Атрибут простий необов’язковий «Чи виконана задача»
	deadline	DateTime	N	Атрибут простий необов’язковий «Термін виконання задачі»
FK	parentTaskId	Integer	N	Рекурсивний зв’язок сутності «Task» Атрибут простий необов’язковий «Ідентифікаційний номер батьківської задачі» ON DELETE CASCADE ON UPDATE CASCADE При видаленні чи оновленні батьківської задачі, видаляти чи оновлювати дану задачу.
	responsiblePerson	Varhcar	N	Атрибут простий необов’язковий «Ім'я людини, що відповідальна за виконання даної задачі»
	description	Varhcar	N	Атрибут простий необов’язковий «Опис задачі»
	task_type	Varchar	NN	Атрибут простий обов’язковий «Тип задачі»

Таблиця 3.33 - Сутність "Task"

Корпоративні обмеження цілісності:

- атрибут «username» є унікальним, неможливо створити декілька користувачів з однаковим логіном
- атрибут «email» є унікальним, неможливо створити декілька користувачів з однаковою електронною поштою
- для задач з типом «Project» та «List» не існує рекурсивного зв’язку «Є батьківською задачею»
- задачі типу «Наступний крок» мають обов’язкові атрибути: task_id, task_name, completed, parentTaskId
- задачі типу «Проєкт» мають обов’язкові атрибути: task_id, task_name, completed; необов'язкові атрибути: deadline, description
- задачі типу «Задача із терміном» мають обов’язкові атрибути: task_id, task_name, deadline
- задачі типу «Знадобиться потім» мають обов’язкові атрибути: task_id, task_name, completed
- задачі типу «Нотатка» мають обов’язкові атрибути: task_id, task_name; необов’язковий атрибути: description
- задачі типу «Делегована задача» мають обов’язкові атрибути: task_id, task_name, completed, respondiblePerson; необов’язкові атрибути: parentTaskId, deadline, description
- задачі типу «Список» мають обов’язкові атрибути: task_id, task_name
- задачі типу «Підзадача» мають обов’язкові атрибути: task_id, task_name, completed, parentTaskId

3.5.2 Інтерфейс програми

Інтерфейс програми розроблявся з метою створити інтуїтивно зрозумілий, легкий до опанування та зручний для використання зовнішній вигляд застосунку.

На рис. 3.34 зображена перша сторінка, яку бачить користувач. На ній присутня форма, що пропонує увійти в систему. У випадку, якщо користувач ще не має аккаунту, можна перейти до форми реєстрації.

Рисунок 3.34 – Сторінка входу в систему

Після аутентифікації, користувач потрапляє на головну сторінку застосування (рис. 3.35). На ній зображено привітання, поточна дата, нагальні задачі (ті, в яких термін виконання збігає наступного дня і раніше) та статистика щодо продуктивності користувача в системі. Якщо нагальних задач немає, користувачу пропонується перейти до списку задач, де це можна буде зробити.

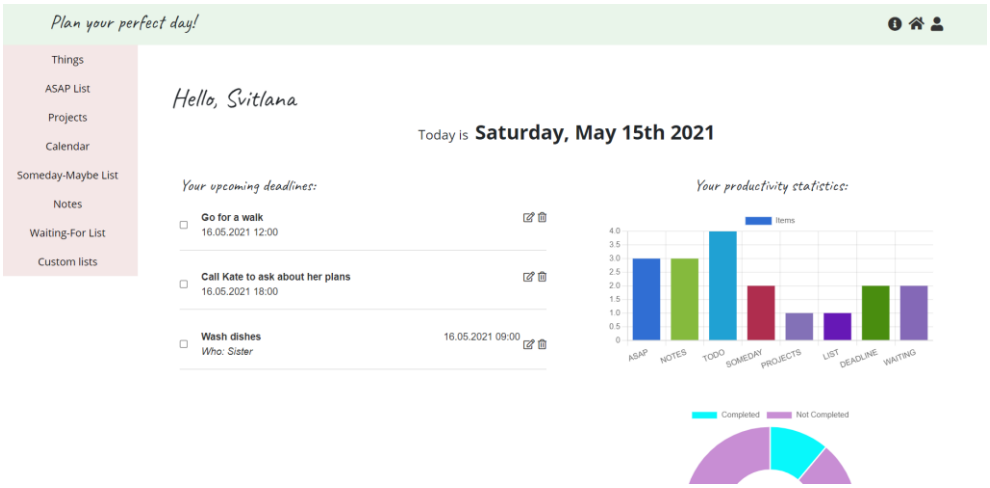

Рисунок 3.35 - Головна сторінка застосунку

Список задач зображений на рис. 3.36. Користувачу надається можливість додавати нові задачі, редагувати, видаляти поточні.

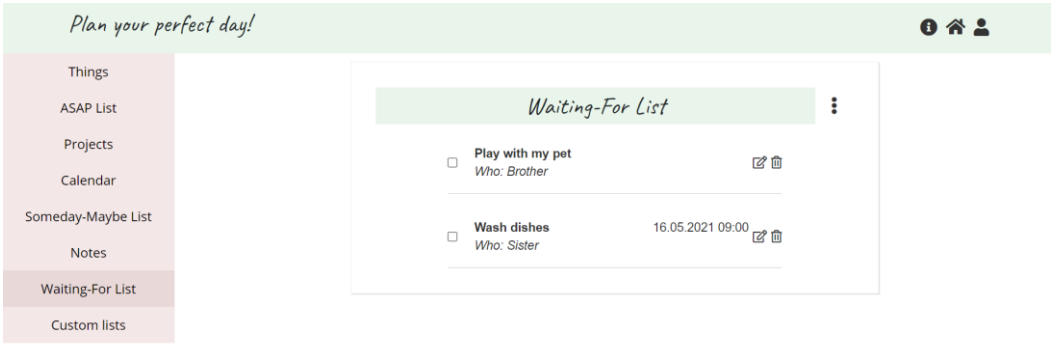

Рисунок 3.36 - Список задач

У пункті меню «Things», у випадаючому вікні можна обрати кнопку «Sort» – тоді користувач попадає на сторінку сортування задач (рис. 3.37). На ній зображена назва поточної задачі, кнопки категорій, до яких можна віднести задачу, а також блок схема алгоритму визначення категорії, запропонований автором системи «Getting Things Done» (додаток А, рис. А.1).

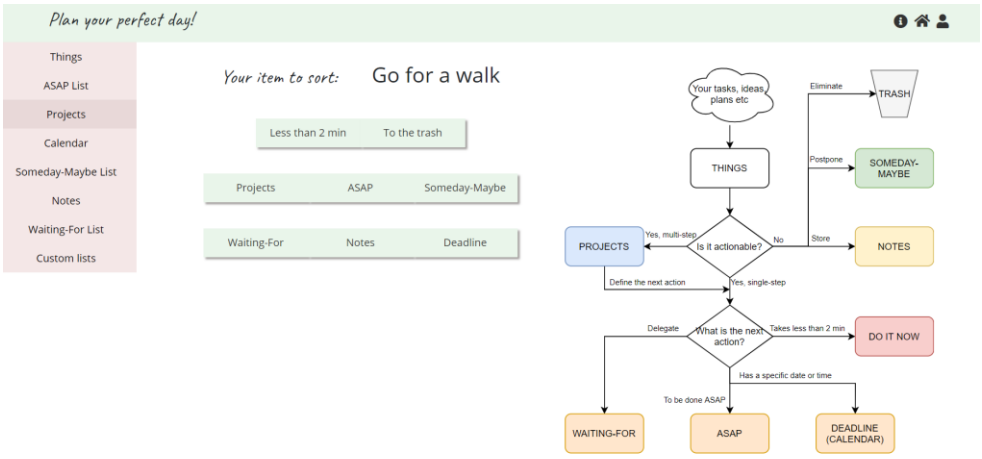

Рисунок 3.37 - Сортування задач

Інтерфейс списку нотатків та власних списків задач виглядає наступним чином – рис. 3.38. Користувач так само може додавати нові елементи, передивлятися, редагувати та видаляти наявні.

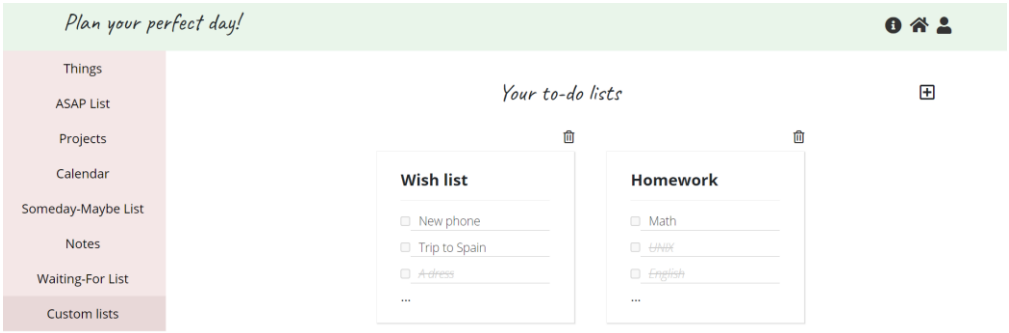

Рисунок 3.38 - Власні списки задач

Редагування задач певних категорій, інформаційна довідка та взаємодія з користувачем відбуваються за допомогою модальних вікон (рис. 3.39).

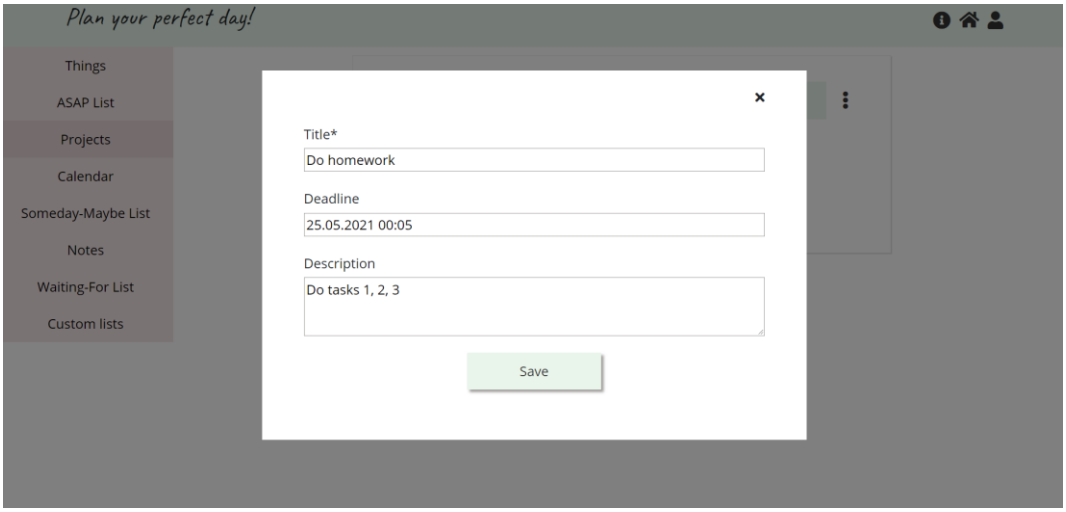

Рисунок 3.39 - Модальне вікно

3.6 Тестування програми і результати її виконання

Програма отримує вхідні дані від користувача у вигляді його запитів та дій, вихідними даними є результати виконання цих запитів, що відображуються на сторінці браузера.

Для початку користування системою, необхідно увійти в існуючий аккаун чи заєструвати новий. При реєстрації, усі поля перевіряються на валідність, що базується на вказаних у системі правилах: усі поля не можуть бути пустими, логін та пошта користувача унікальні – не можуть повторюватися з вже існуючими, логін має бути в межах 4-16 символів, пароль має бути в межах 6-16 символів (рис. 3.40). Допоки вимоги не виконані, кнопка «Зареєструватися» лишається неактивною.

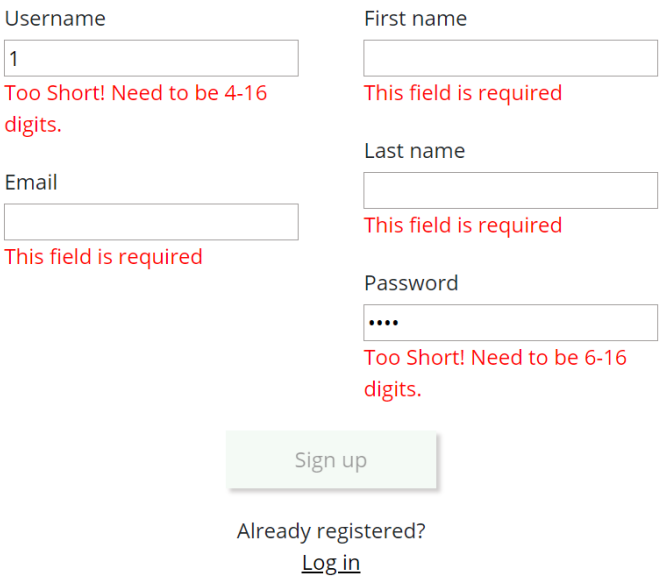

Рисунок 3.40 - Вимоги до форми реєстрації

При використанні вже існуючого логіну чи електроної пошти, з'являється повідомлення про помилку (рис. 3.41).

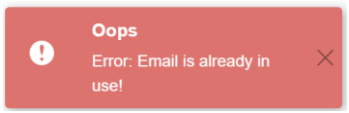

Рисунок 3.41 - Повідомлення про помилку

На головній сторінці користувача відображається статистика про його продуктивність: перша діаграма описує кількість задач в кожній категорії, друга – співвідношення виконаних задач до не виконаних (рис. 3.42). Також, на даному екрані зображені поточні задачі, є можливість їх відредагувати, відмітити виконаними чи видалити (рис. 3.43).

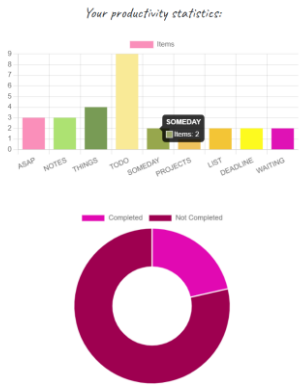

Рисунок 3.42 - Статистика продуктивності користувача

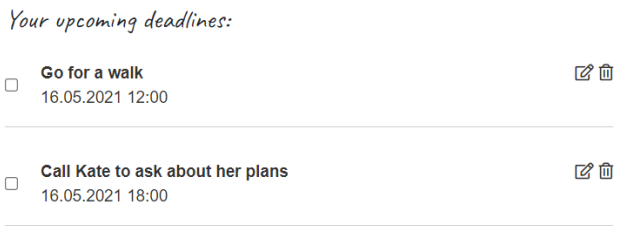

Рисунок 3.43 - Список поточних задач

При додаванні нової задачі, іконка «Підтвердити» стає активною тільки після заповнення усіх необхідних полів. На рис. 4.44 зображено додавання задачі в категорію «Things», на рис. 3.45 – «Project».

Рисунок 3.44 - Додавання задачі до "Things"

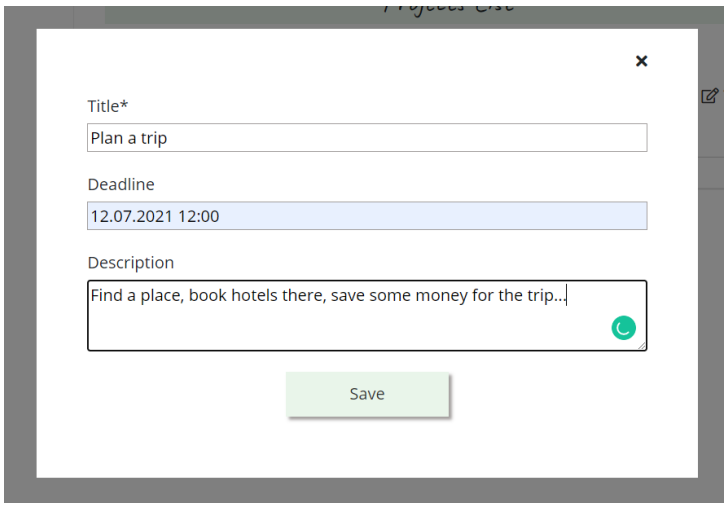

Рисунок 3.45 - Додавання задачі до "Projects"

Після створення, редагування чи видалення задачі, користувач бачить повідомлення про успішність виконаної операції (рис. 3.46).

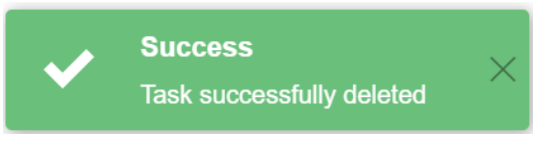

Рисунок 3.46 - Повідомлення про успішність операції

При сортуванні задач на категорії, користувач може обрати одну із запропонованих, залежно від сенсу даної задачі. Він обирає одну з кнопок, з’являється вікно для редагування задачі та додавання нових даних (рис. 3.47).

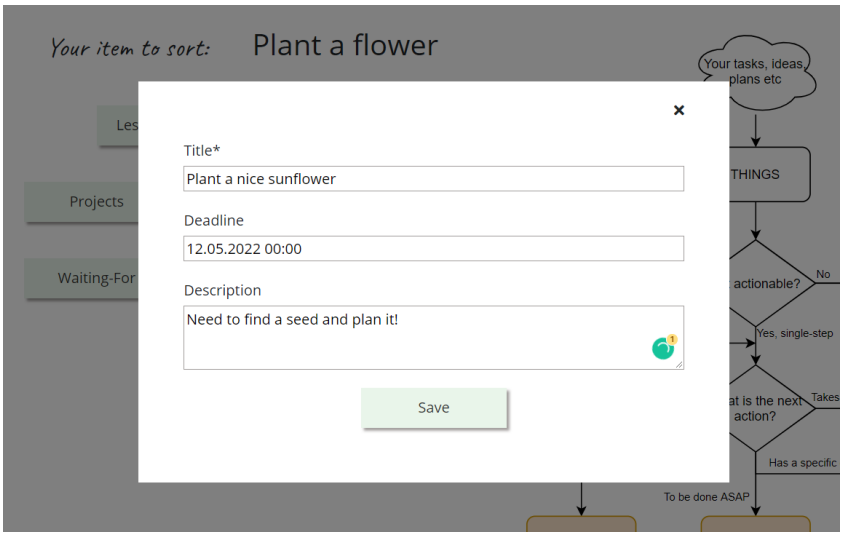

Рисунок 3.47 - Редагування задачі після обрання її категорії

Після збереження інформації про задачу, вона потрапляє у список обраної категорії. У випадку на рис. 3.47, задача опиняється у списку проєктів – рис. 3.48.

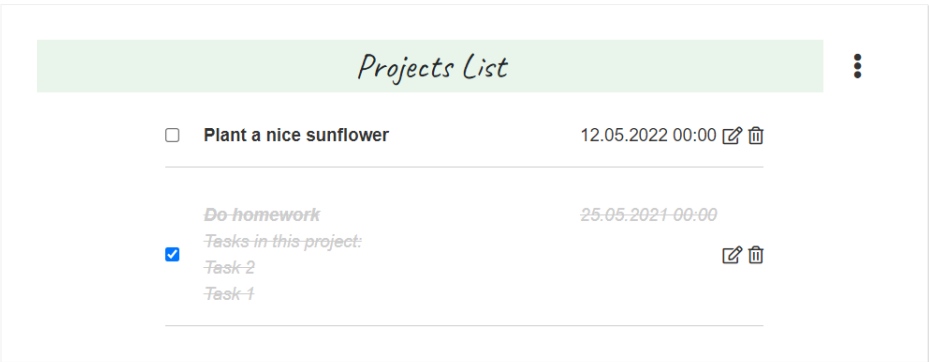

Рисунок 3.48 - Задача у списку "Projects"

Крім цього, у користувача є можливість створити власні списки задач. Він додає бажаний список, вводить його ім'я, може додавати до нього задачі (рис. 3.49 та рис. 3.50).

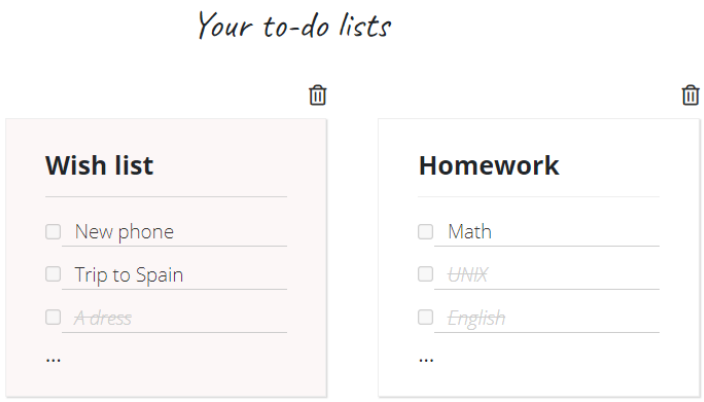

Рисунок 3.49 - Власні списки задач

Рисунок 3.50 - Вміст власного списку задач

Висновки

У результаті виконання курсової роботи була вирішена проблема недостатньо ефективної організації роботи людей, шляхом розроблення веб-застосування для її покращення. Підгрунтям для предметної області став відомий підхід до організації задач «Getting Things Done» автора Девіда Аллена. Дана методологія використовувалася для класифікації задач на категорії, з метою візуалізації та обробки власного навантаження, розподілу задач за їх сенсом та цінністю. Було розглянуто та проаналізовано існуючі на ринку аналоги, зроблено висновки на основі їх переваг та недоліків. За допомогою розглянутих теоретичних відомостей, було проаналізовано типи веб-застосувань, розглянута клієнт-серверна архітектура.

Описані засоби розробки програми (мови програмування TypeScript та Java, додаткові бібліотеки, фреймворк, база даних Postgres) сприяли успішному виконанню всіх поставлених функціональних вимог. У результаті, було створено веб-застосування, що вирішує поставлену проблему, має зручний функціонал та лаконічний, легкий до опанування дизайн.

Крім цього, застосунок має напрямки для подальшого вдосконалення: розширення функціоналу іншими відомими процесами організації роботи з методології «Getting Things Done», наприклад, впровадження методу «43 папки» чи «Шестирівнева модель огляду роботи». Вони є більш просунутими функціями, проте сприятимуть покращенню організації задач користувача. Також у застосунку може бути вдосконалений та урізноманітнений дизайн.

Загалом, зважаючи на результати виконання курсової роботи, можу визначити, що поставлені задачі були виконані, а мета роботи – досягнута.

Список використаних джерел

1. 10 додатків для підвищення продуктивності у 2021 році. [Електронний ресурс]. Режим доступу: https://scadalv.medium.com/10-todo-list-apps-to-help-you-improve-productivity-in-2021-eed8d94105bf
2. Allen D. Getting Things Done / David Allen., 2001. – 267 с. – (1).
3. Застосунок-аналог: Todoist. [Електронний ресурс]. Режим доступу: https://todoist.com
4. 25+ найкращих програм для продуктивності в 2021 році. Todoist. [Електронний ресурс]. Режим доступу: https://collegeinfogeek.com/productivity-apps/
5. Застосунок-аналог: Notion. [Електронний ресурс]. Режим доступу: https://www.notion.so/
6. Застосунок-аналог: Focuster. [Електронний ресурс]. Режим доступу: https://www.focuster.com/
7. Односторінкові (spa) і багатосторінкові (mpa) веб-додатки. [Електронний ресурс]. Режим доступу: https://vc.ru/seo/108149-odnostranichnye-spa-i-mnogostranichnye-pwa-veb-prilozheniya
8. Клієнт-серверна архітектура. [Електронний ресурс]. Режим доступу: https://www.omnisci.com/technical-glossary/client-server
9. TypeScript. Що, навіщо і як? [Електронний ресурс]. Режим доступу: https://medium.com/nuances-of-programming/typescript-%D1%87%D1%82%D0%BE-

%D0%B7%D0%B0%D1%87%D0%B5%D0%BC-%D0%B8-%D0%BA%D0%B0%D0%BA-c3226e693f4

10. TypeScript. [Електронний ресурс]. Режим доступу: https://www.typescriptlang.org/
11. React. [Електронний ресурс]. Режим доступу: https://reactjs.org/
12. Redux. [Електронний ресурс]. Режим доступу: https://redux.js.org/
13. PostgreSQL. [Електронний ресурс]. Режим доступу: https://www.postgresql.org/
14. Moment.js. [Електронний ресурс]. Режим доступу: https://momentjs.com/
15. Axios. [Електронний ресурс]. Режим доступу: https://www.npmjs.com/package/axios
16. Chart.js. [Електронний ресурс]. Режим доступу: https://www.chartjs.org/
17. Fetch чи Axios.js для створення http запитів. [Електронний ресурс]. Режим доступу: https://medium.com/@thejasonfile/fetch-vs-axios-js-for-making-http-requests-2b261cdd3af5
18. Стандарт RFC 7519. [Електронний ресурс]. Режим доступу: https://datatracker.ietf.org/doc/html/rfc7519
19. П’ять простих кроків, щоб зрозуміти JSON Web Tokens. [Електронний ресурс]. Режим доступу: https://medium.com/cyberverse/five-easy-steps-to-understand-json-web-tokens-jwt-7665d2ddf4d5
20. Визначення Stateful та Stateless веб сервісів. [Електронний ресурс]. Режим доступу: https://nordicapis.com/defining-stateful-vs-stateless-web-services/

Додаток А. Алгоритм визначення категорії задачі за методологією «Getting Things Done»

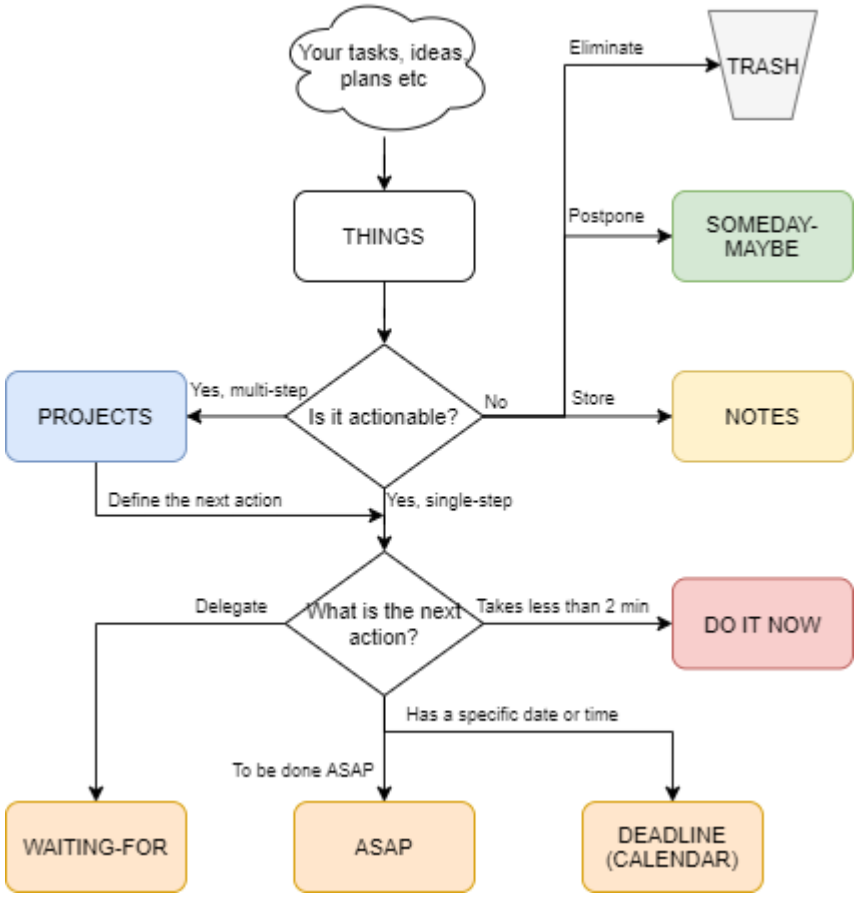

Рисунок А.1 - Алгоритм визначення категорії задачі

Додаток Б. Макети інтерфейсів

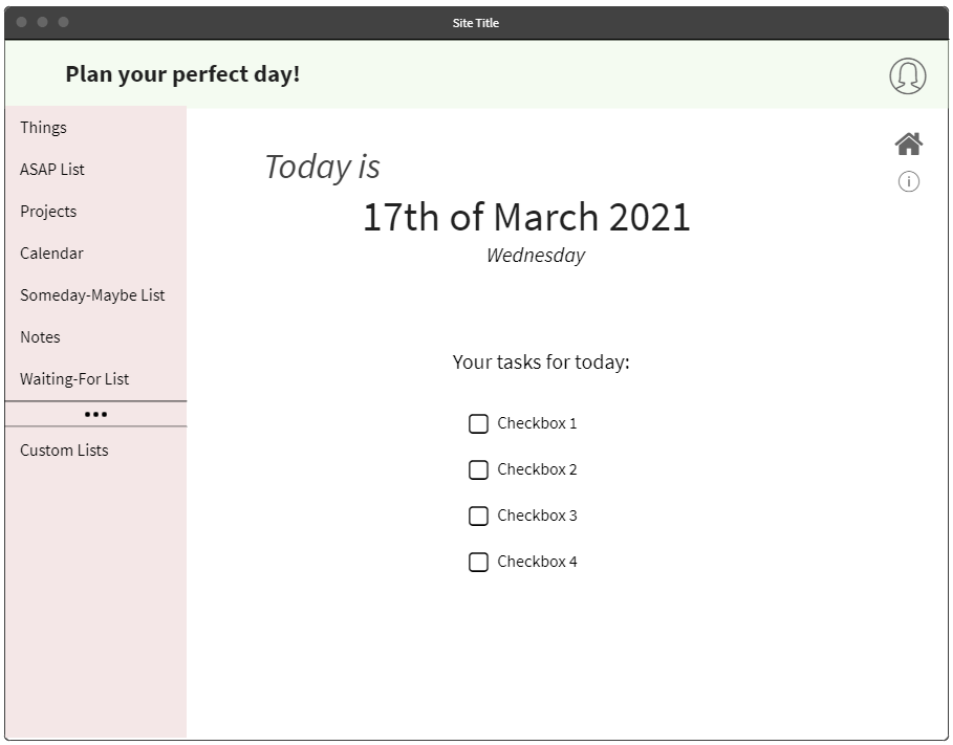

Рисунок Б.1 - Головна сторінка застосунку

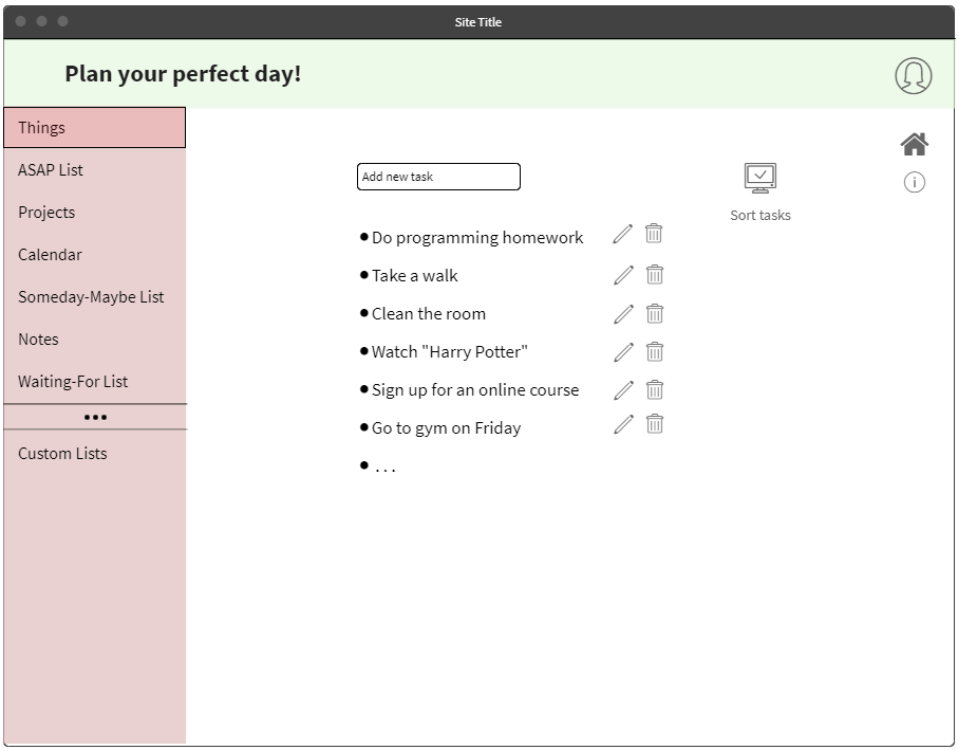

Рисунок Б.2 - Список задач

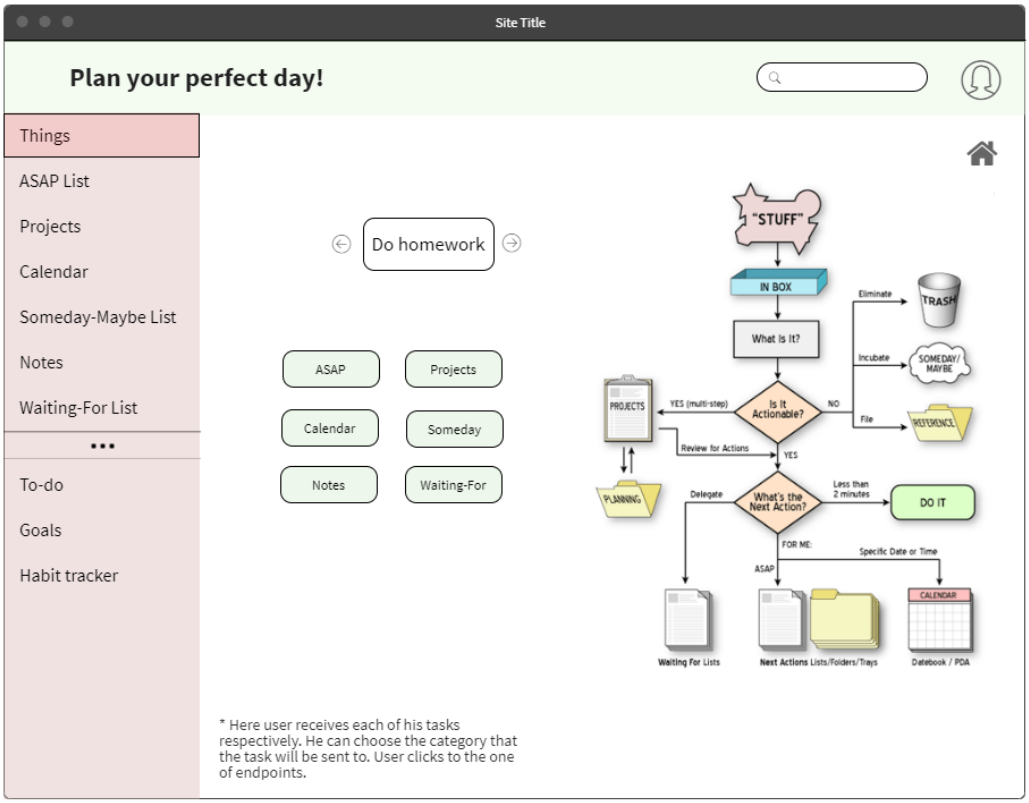

Рисунок Б.3 - Сторінка сортування

* If the user chooses Projects

He sets the name, description for the task, then the date and time for the task to be done before.

When Save button is clicked, the task goes to the Projects List

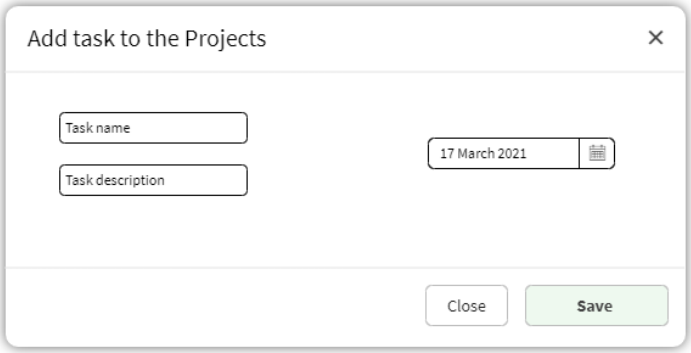

Рисунок Б.4 - Модальне вікно

Додаток В. Код програми серверної частини

Контроллер аутентифікації – authController.java

```
package org.lukichova.tasklist.controller;

import lombok.RequiredArgsConstructor;
import org.lukichova.tasklist.controller.dto.request.LoginRequest;
import org.lukichova.tasklist.controller.dto.request.SignupRequest;
import org.lukichova.tasklist.controller.dto.response.JwtResponse;
import org.lukichova.tasklist.controller.dto.response.MessageResponse;
import org.lukichova.tasklist.model.Role;
import org.lukichova.tasklist.model.User;
import org.lukichova.tasklist.repository.UserRepository;
import org.lukichova.tasklist.security.UserDetailsImpl;
import org.lukichova.tasklist.security.jwt.JwtUtils;
import org.springframework.http.ResponseEntity;
import org.springframework.security.authentication.AuthenticationManager;
import org.springframework.security.authentication.UsernamePasswordAuthenticationToken;
import org.springframework.security.core.Authentication;
import org.springframework.security.core.context.SecurityContextHolder;
import org.springframework.security.crypto.password.PasswordEncoder;
import org.springframework.util.StringUtils;
import org.springframework.web.bind.annotation.*;

import javax.validation.Valid;
import java.util.List;
import java.util.stream.Collectors;

@CrossOrigin(origins = "*", maxAge = 3600)
@RestController
@RequestMapping("/api/auth")
@RequiredArgsConstructor
public class AuthController {

    private final AuthenticationManager authenticationManager;
    private final UserRepository userRepository;
    private final PasswordEncoder encoder;
    private final JwtUtils jwtUtils;

    @PostMapping("/signin")
    public ResponseEntity<?> authenticateUser(@Valid @RequestBody LoginRequest loginRequest) {

        Authentication authentication = authenticationManager.authenticate(
                new UsernamePasswordAuthenticationToken(loginRequest.getUsername(),
                        loginRequest.getPassword()));

        SecurityContextHolder.getContext().setAuthentication(authentication);
        String jwt = jwtUtils.generateJwtToken(authentication);

        UserDetailsImpl userDetails = (UserDetailsImpl) authentication.getPrincipal();
        List<String> roles = userDetails.getAuthorities().stream()
                .map(item -> item.getAuthority())
                .collect(Collectors.toList());

        return ResponseEntity.ok(new JwtResponse(jwt,
                userDetails.getId(),
                userDetails.getUsername(),
                userDetails.getFirstName(),
                userDetails.getLastName(),
                userDetails.getEmail(),
                roles));
    }
```

```
    @PostMapping("/signup")
    public ResponseEntity<?> registerUser(@Valid @RequestBody SignupRequest signUpRequest) {
        if (userRepository.existsByUsername(signUpRequest.getLogin())) {
            return ResponseEntity
                    .badRequest()
                    .body(new MessageResponse("Error: Username is already taken!"));
        }
        if (userRepository.existsByEmail(signUpRequest.getEmail())) {
            return ResponseEntity
                    .badRequest()
                    .body(new MessageResponse("Error: Email is already in use!"));
        }
        User user = User.builder()
                .username(signUpRequest.getLogin())
                .firstName(signUpRequest.getFirstName())
                .lastName(signUpRequest.getLastName())
                .email(signUpRequest.getEmail())
                .password(encoder.encode(signUpRequest.getPassword()))
                .role(Role.ROLE_USER)
                .build();
        userRepository.save(user);
        return ResponseEntity.ok(new MessageResponse("User successfully registered"));
    }

    @GetMapping("/info")
    public ResponseEntity getUserInfo(@RequestHeader("Authorization") String authorization) {

        if (StringUtils.hasText(authorization) && authorization.startsWith("Bearer ")) {
            var token = authorization.substring(7, authorization.length());
            var userName = jwtUtils.getUserNameFromJwtToken(token);
            var user = userRepository.findByUsername(userName)
                    .orElseThrow(() -> new RuntimeException("User not found"));
            return ResponseEntity.ok(new JwtResponse(token,
                    user.getId(),
                    user.getUsername(),
                    user.getFirstName(),
                    user.getLastName(),
                    user.getEmail(),
                    List.of(user.getRole().name())));
        }
        return ResponseEntity.badRequest().body(new MessageResponse("Error: User not logged in!"));

    }
}
```

Контроллер роботи із задачами – taskController.java

```
package org.lukichova.tasklist.controller;

import io.jsonwebtoken.lang.Assert;
import lombok.RequiredArgsConstructor;
import lombok.extern.slf4j.Slf4j;
import org.lukichova.tasklist.controller.dto.request.TaskCreate;
import org.lukichova.tasklist.controller.dto.request.TaskEdit;
import org.lukichova.tasklist.controller.dto.request.TaskId;
import org.lukichova.tasklist.controller.dto.response.StatisticsDto;
import org.lukichova.tasklist.model.Task;
import org.lukichova.tasklist.service.TaskService;
import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.*;

import java.util.List;
```

```java
@CrossOrigin(origins = "*", maxAge = 3600)
@RestController
@RequiredArgsConstructor
@Slf4j
@RequestMapping("/api/task")
public class TaskController {

    private final TaskService taskService;

    @GetMapping
    public ResponseEntity getAllTasks() {
        log.info("Executed request get all task");
        return ResponseEntity.ok(taskService.getAllTasks());
    }

    @GetMapping(value = "/id/{id}")
    public ResponseEntity<Task> getTasksById(@PathVariable Long id) {
        Assert.notNull(id, "Id cannot be null");

        log.info("Executed request get task by id {}", id);
        return ResponseEntity.ok(taskService.getTaskById(id));
    }

    @GetMapping(value = "/{taskType}")
    public ResponseEntity<List<Task>> getAllTasksByType(@PathVariable String taskType) {
        Assert.notNull(taskType, "Task type cannot be null");

        log.info("Executed request get all task of type {}", taskType);
        return ResponseEntity.ok(taskService.getAllByType(taskType));
    }

    @GetMapping(value = "/with-deadline")
    public ResponseEntity<List<Task>> getAllTasksWithDeadline() {
        log.info("Executed request get all task with deadline");
        return ResponseEntity.ok(taskService.getAllWithDeadline());
    }

    @GetMapping(value = "/near-deadline")
    public ResponseEntity<List<Task>> getAllTasksNearDeadline() {
        log.info("Executed request get all task with deadline tomorrow");
        return ResponseEntity.ok(taskService.getAllNearDeadline());
    }

    @GetMapping(value = "/{parentTaskId}/children")
    public ResponseEntity<List<Task>> getAllChildTasks(@PathVariable Long parentTaskId) {
        Assert.notNull(parentTaskId, "Parent task Id cannot be null");

        log.info("Executed request get all child tasks for parent task Id {}", parentTaskId);
        return ResponseEntity.ok(taskService.getChildTasks(parentTaskId));
    }

    @PostMapping
    public ResponseEntity create(@RequestBody TaskCreate taskCreate) {
        Long id = taskService.create(taskCreate);
        log.info("Created task with id {} and name {}", id, taskCreate.getName());

        return ResponseEntity.status(HttpStatus.CREATED).header("id", String.valueOf(id)).build();
    }

    @PostMapping(value = "/complete")
    public ResponseEntity complete(@RequestBody TaskId taskId) {
        taskService.complete(taskId);
        log.info("Completed task with id {}", taskId.getId());

        return ResponseEntity.ok().build();
    }
```

```
    @DeleteMapping(value = "/{taskType}")
    public ResponseEntity deleteCompleted(@PathVariable String taskType) {
        Assert.notNull(taskType, "Task type cannot be null");

        taskService.deleteCompleted(taskType);
        log.info("Deleted tasks of {} type", taskType);

        return ResponseEntity.ok().build();
    }

    @DeleteMapping(value = "/id/{taskId}")
    public ResponseEntity deleteTask(@PathVariable Long taskId) {
        Assert.notNull(taskId, "Task id cannot be null");

        taskService.deleteTask(taskId);
        log.info("Deleted tasks with id {}", taskId);

        return ResponseEntity.ok().build();
    }

    @PutMapping
    public ResponseEntity update(@RequestBody TaskEdit taskEdit) {
        taskService.update(taskEdit);
        log.info("Updated task id {} with data {}", taskEdit.getId(), taskEdit);

        return ResponseEntity.ok().build();
    }

    @GetMapping("statistics")
    public ResponseEntity<StatisticsDto> statistics() {
        var stat = taskService.statistics();
        log.info("Statistics request");

        return ResponseEntity.ok(stat);
    }
}
```

Додаток Г. Код програми клієнтської сторони

Компонент, відповідальний за маршрутизацію – Page.tsx

```
import React from "react";
import styles from "./Page.module.sass";
import {ICurrentUser} from "../../api/auth/authModels";
import {Route, RouteComponentProps, Switch, withRouter} from "react-router-dom";
import ThingsList from "components/List/ThingsList";
import ASAPList from "components/List/ASAPList";
import ProjectsList from "components/List/ProjectsList";
import SomedayList from "components/List/SomedayList";
import WaitingList from "components/List/WaitingList";
import NotesList from "components/List/NotesList";
import CalendarList from "../List/CalendarList";
import Sorter from "../Sorter/Sorter";
import HomePage from "../HomePage/HomePage";
import TodoList from "../TodoList/TodoList";
import TodoListItemContent from "../TodoListItemContent/TodoListItemContent";

interface IOwnProps {
    currentUser?: ICurrentUser;
}

class Page extends React.Component<RouteComponentProps & IOwnProps> {

    render() {

        return (
            <div className={styles.pageWrapper}>
                <Switch>
                    <Route exact path={"/home"} render={() => <HomePage
                        currentUser={this.props.currentUser}/>}/>
                    <Route path="/home/things" render={() => <ThingsList/>}/>
                    <Route path="/home/sort" render={() => <Sorter/>}/>
                    <Route path="/home/asap" render={() => <ASAPList/>}/>
                    <Route path="/home/projects" render={() => <ProjectsList/>}/>
                    <Route path="/home/calendar" render={() => <CalendarList/>}/>
                    <Route path="/home/someday" render={() => <SomedayList/>}/>
                    <Route path="/home/notes" render={() => <NotesList/>}/>
                    <Route path="/home/waiting" render={() => <WaitingList/>}/>
                    <Route path="/home/list" render={() => <TodoList/>}/>
                    <Route path="/home/list-item/:id" render={() =>
<TodoListItemContent/>}/>
                </Switch>
            </div>
        );
    }
}

export default withRouter(Page);
```

Головна сторінка корситувача веб-застосунком – HomePage.tsx

```
import React from "react";
import styles from "./HomePage.module.sass";
import {ICurrentUser} from "../../api/auth/authModels";
import moment from "moment";
import {
    IListItem, IListItemDeadline,
    IListItemInline, IListItemProject, IListItemWaiting, MenuItemsEnum,
} from "../../api/listItem/listItemModels";
```

```tsx
import listItemService from "../../api/listItem/listItemService";
import {toastr} from "react-redux-toastr";
import ListItemInline from "../ListItem/ListItemInline";
import {IInlineData} from "../EditTask/EditTaskInline";
import {
    mapDeadlineDataToItem,
    mapProjectItemToItem,
    mapWaitingItemToItem
} from "../../api/mappers";
import ListItemProjects from "../ListItem/ListItemProjects";
import ListItemWaiting from "../ListItem/ListItemWaiting";
import Modal from "../Modal/Modal";
import EditTaskWaiting, {IWaitingData} from "../EditTask/EditTaskWaiting";
import EditTaskProject, {IProjectData} from "../EditTask/EditTaskProject";
import {Link, RouteComponentProps, withRouter} from "react-router-dom";
import Statistics from "../Statistics/Statistics";

interface IOwnProps {
    currentUser?: ICurrentUser;
}

interface IState {
    tasksForToday?: IListItem[];
    editProject?: IListItemProject
    editWaiting?: IListItemWaiting
}

class HomePage extends React.Component<RouteComponentProps & IOwnProps, IState> {
    state = {} as IState;

    async componentDidMount() {
        await this.loadTodayTasks();
    }

    private async loadTodayTasks() {
        const tasksForToday = (
            await listItemService.getListItemsNearDeadline()) as any as IListItem[];

        this.setState({tasksForToday});
    }

    handleDelete = async (id: number) => {
        await listItemService.deleteTask(id);
        toastr.success('Success', 'Task successfully deleted');
        this.loadTodayTasks();
    };

    handleEditDeadline = async (request: IListItemDeadline, item: IInlineData) => {
        await listItemService.editProject({
            ...mapDeadlineDataToItem(request),
            ...{...item, parentTaskId: item.parentTaskId} as IInlineData,
        });
        toastr.success('Success', 'Task successfully updated');
        await this.loadTodayTasks();
    };

    handleChange = async (id: number) => {
        await listItemService.changeCompleted(id);
        this.setState(prevState => {
            const updatedTodos = prevState.tasksForToday?.map(item => {
                if (item.id === id) {
                    item.completed = !item.completed;
                }
                return item;
            });
            return {
```

```
                ...prevState, tasksForToday: updatedTodos
            };
        });
    };

    handleEditWaiting = async (task: IListItemWaiting, data: IWaitingData) => {
        const newTask: IListItemWaiting = {...task, ...data};
        await listItemService.editProject(mapWaitingItemToItem(newTask));
        toastr.success('Success', 'Task successfully updated');
        this.setState({editWaiting: undefined});
        await this.loadTodayTasks();
    };

    handleEditProject = async (task: IListItemProject, data: IProjectData) => {
        const newTask: IListItemProject = {...task, ...data};
        await listItemService.editProject(mapProjectItemToItem(newTask));
        toastr.success('Success', 'Task successfully updated');
        this.setState({editProject: undefined});
        await this.loadTodayTasks();
    };

    handleListItem = (item: IListItem): JSX.Element => {
        switch (item.type) {
            case MenuItemsEnum.DEADLINE:
                return <ListItemInline
                    item={item as IListItemInline}
                    parentEnabled={false}
                    handleEdit={(newTask: IInlineData) => this.handleEditDeadline(item as
IListItemDeadline, newTask)}
                    handleDelete={() => this.handleDelete(item.id)}
                    handleChange={() => this.handleChange(item.id)}
                />;
            case MenuItemsEnum.PROJECTS:
                return <ListItemProjects
                    id={item.id}
                    text={item.name}
                    type={MenuItemsEnum.PROJECTS}
                    completed={item.completed}
                    deadline={item.deadline}
                    description={item.description}
                    handleDelete={() => this.handleDelete(item.id)}
                    handleChange={() => this.handleChange(item.id)}
                    openModal={() => this.setState({editProject: item as
IListItemProject})}
                />;
            case MenuItemsEnum.WAITING:
                return <ListItemWaiting
                    id={item.id}
                    text={item.name}
                    type={MenuItemsEnum.WAITING}
                    completed={item.completed}
                    deadline={item.deadline}
                    responsiblePerson={item.responsiblePerson}
                    openModal={() => this.setState({editWaiting: item as
IListItemWaiting})}
                    handleChange={() => this.handleChange(item.id)}
                    handleDelete={() => this.handleDelete(item.id)}
                />;
            default:
                throw new Error("Error!");
        }
    };

    render() {
        const {currentUser} = this.props;
        const {tasksForToday, editProject, editWaiting} = this.state;
```

```
        const listItems = tasksForToday?.map(item =>
            this.handleListItem(item));

        return (
            <div className={styles.wrapper}>
                {editProject && (
                    <Modal
                        close={() => this.setState({editProject: undefined})}>
                        <EditTaskProject initTask={editProject}
                                         saveTask={data =>
this.handleEditProject(editProject, data)}/>
                    </Modal>
                )}
                {editWaiting && (
                    <Modal
                        close={() => this.setState({editWaiting: undefined})}>
                        <EditTaskWaiting initTask={editWaiting}
                                         saveTask={data =>
this.handleEditWaiting(editWaiting, data)}/>
                    </Modal>
                )}

                <div className={styles.greeting}>
                    Hello, {currentUser?.firstName}
                </div>
                <div className={styles.today}>
                    Today is  
                    <span className={styles.date}>
                         {moment().format("dddd, MMMM Do YYYY")}
                    </span>
                </div>
                <div className={styles.row}>
                    <div className={styles.col}>
                        {!!tasksForToday?.length &&
                        <div className={styles.todayTasksDiv}>
                            <p className={styles.p}>Your upcoming deadlines:</p>
                            <div>{listItems}</div>
                        </div>
                        }
                        {!tasksForToday?.length &&
                        <div className={styles.todayTasksDiv}>
                            <p className={styles.p}>You have no tasks with upcoming
deadline.</p>
                            <Link className={"text-decoration-none"} to="/home/things">
                                <p className={styles.bold}>Add more tasks.</p>
                            </Link>
                        </div>
                        }
                    </div>
                    <div className={styles.col}>
                        <Statistics/>
                    </div>
                </div>

            </div>
        );
    }
}

export default withRouter(HomePage);
```