МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ДЕРЖАВНИЙ УНІВЕРСИТЕТ ЕКОНОМІКИ І ТЕХНОЛОГІЙ

ННІ/факультет Навчально-науковий інститут економіки та бізнес-освіти

Кафедра Економіки та цифрового бізнесу

Спеціальність 122 «Комп'ютерні науки»

Форма навчання Денна

КВАЛІФІКАЦІЙНА РОБОТА БАКАЛАВРА

Даць Софії Володимирівни
(прізвище, ім'я, по батькові здобувача)

на тему Розробка Web-застосунку для моніторингу та аналізу продуктивності життєдіяльності
(повна назва теми)

за матеріалами

(повна назва бази дослідження)

науковий керівник к.т.н. *(наук. ступінь, вчене звання)* _____ *(підпис)* Селезньов М.Є. *(прізвище, ініціали)*

Робота допущена до захисту в ЕК
Протокол засідання кафедри
від 09 червня 2025р. № 12
Завідувач кафедри _____
(підпис)
к.е.н., доцент *наук. ступень, вчене звання* Радько В.М. прізвище, ініціали

Кривий Ріг – 2025

ЗАТВЕРДЖЕНО
Наказ Міністерства освіти і науки, молоді та спорту України
29 березня 2012 року № 384

Форма № Н-9.01

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ДЕРЖАВНИЙ УНІВЕРСИТЕТ ЕКОНОМІКИ І ТЕХНОЛОГІЙ
НАВЧАЛЬНО-НАУКОВИЙ ІНСТИТУТ ЕКОНОМІКИ ТА БІЗНЕС-ОСВІТИ
(повне найменування вищого навчального закладу)

Кафедра економіки та цифрового бізнесу
Освітній ступінь бакалавр
Спеціальність 122 «Комп'ютерні науки»

ЗАТВЕРДЖУЮ
Завідувач кафедри _____В.М. Радько

"07" квітня 2025 року

З А В Д А Н Н Я
НА КВАЛІФІКАЦІЙНУ РОБОТУ БАКАЛАВРА ЗДОБУВАЧУ

Даць Софії Володимирівні

1. Тема роботи «Розробка Web-застосунку для моніторингу та аналізу продуктивності життєдіяльності»

науковий керівник роботи Селезньов Максим Євгенович,
затверджені наказом вищого навчального закладу від «04» квітня 2025 р. № 224-ст (д/ф)
№ 151-ст (з/ф)

2. Строк подання здобувачем роботи 31.05.2025р.

3. Зміст кваліфікаційної роботи бакалавра, об'єкт, предмет та мета дослідження:

Розділ 1 ТЕОРЕТИЧНІ АСПЕКТИ РОЗРОБКИ WEB-ЗАСТОСУНКУ ДЛЯ МОНІТОРИНГУ ПРОДУКТИВНОСТІ ТА ОГЛЯД ІСНУЮЧИХ РІШЕНЬ
Розділ 2 ПРОЄКТУВАННЯ ТА РОЗРОБКА WEB-ЗАСТОСУНКУ
Розділ 3 ФУНКЦІОНУВАННЯ ТА ЕЛЕМЕНТИ ІНТЕРФЕЙСУ WEB-ЗАСТОСУНКУ

Об'єкт дослідження - процес створення інформаційної системи для моніторингу та аналізу продуктивності життєдіяльності у вигляді Web-застосунку

Предмет дослідження - функціональні та технічні аспекти розробки Web-застосунку для моніторингу та аналізу продуктивності життєдіяльності

Мета кваліфікаційної роботи бакалавра – створення повнофункціонального Web-застосунку, який дозволяє здійснювати моніторинг, аналіз та візуалізацію показників продуктивності життєдіяльності користувача на основі його даних, та результатів їх обробки

4. Дата видачі завдання 04.04.2025р.

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів кваліфікаційної роботи бакалавра	Строк виконання етапів роботи	Відмітка керівника про виконання етапів (дата, підпис)
1	Підготовка розділу 1	до 28.04.2025р.	26.04.2025
2	Підготовка розділу 2	до 16.05.2025р.	15.05.2025
3	Підготовка розділу 3	до 30.05.2025р.	28.05.2025
4	Реєстрація завершеної дипломної роботи	до 31.05.2025р.	30.05.2025
5	Отримання відгуку від наукового керівника	03-04.06.2025р.	04.06.2025
6	Отримання зовнішньої рецензії	05-06.06.2025р.	06.06.2025
7	Перевірка кваліфікаційної роботи на плагіат	02-09.06.2025р.	04.06.2025
8	Попередній захист кваліфікаційної роботи на кафедрі	03.06.2025р.	03.06.2025
9	Допуск кафедрою кваліфікаційної роботи до захисту	09.06.2025р.	09.06.2025
10	Підготовка студента до захисту в ЕК	до 17.06.2025р.	17.06.2025

Завдання підготував науковий керівник _____ **Селезньов М. Є**
(підпис) (прізвище та ініціали)

Завдання одержав здобувач _____ **Даць С. В.**
(підпис) (прізвище та ініціали)

Примітки:

1. Форму призначено для видачі завдання здобувачу на виконання кваліфікаційної роботи бакалавра і контролю за ходом роботи з боку кафедри.
2. Розробляється керівником кваліфікаційної роботи. Видається кафедрою.
3. Формат бланка А4 (210×297 мм), 2 сторінки.

РЕФЕРАТ

Робота містить 78 сторінок, 10 рисунків, 50 джерел, 1 таблицю і 2 додатки.

Об'єкт дослідження: процес створення інформаційної системи для моніторингу та аналізу продуктивності життєдіяльності у вигляді Web-застосунку.

Предмет дослідження: функціональні та технічні аспекти розробки Web-застосунку для моніторингу та аналізу продуктивності життєдіяльності.

Мета дослідження: створення повнофункціонального Web-застосунку, який дозволяє здійснювати моніторинг, аналіз та візуалізацію показників продуктивності життєдіяльності користувача на основі його даних, та результатів їх обробки.

Область застосування: розроблений Web-застосунок може слугувати інструментом для особистісного моніторингу, щоденного планування, виявлення і аналізу поведінкових патернів, а також розробки індивідуальних порад щодо підвищення продуктивності. Запропоноване рішення має потенціал для адаптації в освітньому середовищі, практиці психологічної підтримки та при реалізації інших ініціатив з розвитку ментального добробуту.

ІНФОРМАЦІЙНА СИСТЕМА, МОНІТОРИНГ, WEB-ЗАСТОСУНОК, ПРОДУКТИВНІСТЬ, ЖИТТЄДІЯЛЬНІСТЬ, WEB-РОЗРОБКА, WEB-ІНТЕРФЕЙС, АВТОМАТИЗАЦІЯ.

ЗМІСТ

ПЕРЕЛІК УМОВНИХ СКОРОЧЕНЬ

СКБД - Система Керування Базами Даних;

API - Application Programming Interface;

CI/CD - Continuous Integration / Continuous Deployment

HTTPS - HyperText Transfer Protocol Secure

JWT - JSON Web Token;

ORM - Object-Relational Mapping

REST - Representational State Transfer.

SPA - Single Page Application

UI - User Interface

UX - User Experience

ВСТУП

У сучасному інформаційному суспільстві, де темп життя невпинно зростає, а цифрові технології проникають у всі сфери людської діяльності, проблема ефективного використання особистого часу, ресурсів і життєвої енергії набуває особливої актуальності. Зростання обсягів завдань, інформаційне перевантаження, мультизадачність, а також потреба у збалансуванні професійної, соціальної та особистої сфер зумовлюють потребу в нових засобах організації та аналізу продуктивності життєдіяльності. В таких умовах актуальним стає створення інструментів, які не лише фіксують дії користувача, але й дають змогу проводити глибоку аналітику, визначати закономірності, прогнозувати ефективність життєдіяльності та пропонувати шляхи її підвищення.

Розробка Web-застосунку для моніторингу та аналізу продуктивності життєдіяльності є важливою складовою в системі цифрової трансформації особистого самоменеджменту. Web-платформи мають переваги доступності, гнучкості, масштабованості та інтеграційної сумісності з іншими сервісами. Завдяки застосуванню сучасних технологій Web-розробки, таких як React, Node.js, MongoDB, REST API тощо, з’являється можливість створити багатофункціональний, інтуїтивно зрозумілий інструмент, який буде не лише зберігати дані користувача, а й інтерпретувати їх у контексті динаміки, мотиваційної ефективності, енергетичного балансу та когнітивного навантаження.

Об’єкт дослідження: процес створення інформаційної системи для моніторингу та аналізу продуктивності життєдіяльності у вигляді Web-застосунку.

Предмет дослідження: функціональні та технічні аспекти розробки Web-застосунку для моніторингу та аналізу продуктивності життєдіяльності.

Метою роботи є створення повнофункціонального Web-застосунку, який дозволяє здійснювати моніторинг, аналіз та візуалізацію показників

продуктивності життєдіяльності користувача на основі його даних, та результатів їх обробки.

В процесі дослідження вирішувалися наступні завдання::

- Проаналізувати теоретичні основи поняття «продуктивність життєдіяльності» в сучасному інформаційному середовищі.
- Оглянути існуючі програмні рішення для моніторингу особистої ефективності та визначити їхні переваги і недоліки.
- Розробити концептуальну модель функціонування системи моніторингу.
- Побудувати структуру бази даних, що дозволяє ефективно зберігати, обробляти та візуалізувати динаміку активності користувача.
- Реалізувати фронтенд та бекенд компоненти Web-застосунку з використанням сучасного стеку технологій.
- Провести тестування застосунку, оцінити його стабільність, функціональність та перспективи масштабування.

Використовувалися такі методики:

- аналіз та синтез;
- моделювання;
- порівняльний аналіз;
- емпіричне тестування.

Практична значущість результатів дослідження полягає в отриманому функціонуючому Web-застосунку, який можна використовувати для самоспостереження, планування дня, аналізу власних звичок, а також формування рекомендацій для підвищення особистої ефективності. Розробка може бути адаптована для використання в освітніх закладах, психологічному консультуванні, а також у програмах ментального добробуту.

РОЗДІЛ 1
ТЕОРЕТИЧНІ АСПЕКТИ РОЗРОБКИ WEB-ЗАСТОСУНКУ ДЛЯ МОНІТОРИНГУ ПРОДУКТИВНОСТІ ТА ОГЛЯД ІСНУЮЧИХ РІШЕНЬ

1.1 Огляд сучасних підходів до моніторингу продуктивності

У сучасному інформаційно-цифровому суспільстві поняття продуктивності життєдіяльності набуло якісно нового змісту. Традиційні підходи до продуктивності, які ґрунтувалися на кількісному вимірюванні витрат часу та обсягу виконаної роботи, поступаються місцем комплексним моделям, що включають когнітивні, емоційні, соціальні та технологічні чинники. Відтак, продуктивність життєдіяльності в цифровому середовищі розглядається як інтегральний показник ефективності взаємодії особистості з навколишнім середовищем, що формується в умовах глибокої диджиталізації та постійного інформаційного обміну [1].

До ключових ознак такого середовища належать: постійна доступність інформації, розмитість меж між роботою та особистим життям, багатозадачність, когнітивне перевантаження, зростаюча залежність від цифрових інтерфейсів та алгоритмізованих сервісів. У контексті зазначених змін, продуктивність перестає бути лише характеристикою індивідуальної праці, а набуває системного характеру, пов’язаного з адаптивністю, стресостійкістю, саморегуляцією, здатністю приймати обґрунтовані рішення в умовах швидкоплинної інформаційної динаміки.

Вона включає такі складові, як організація часу, самоспостереження, рефлексивне ставлення до результатів діяльності, грамотне використання цифрових інструментів, а також вміння балансувати між ефективністю та психоемоційним благополуччям. Особливого значення набуває поняття цифрової самодисципліни, що означає здатність свідомо обмежувати нераціональні цифрові активності, регулювати взаємодію з гаджетами, контролювати

інформаційне навантаження, впроваджувати практики цифрового відновлення (digital detox) тощо [9].

У процесі формування продуктивної життєдіяльності особистість повинна постійно оновлювати власні стратегії поведінки відповідно до мінливих умов цифрового середовища. Цей процес супроводжується необхідністю переосмислення таких понять, як «корисна діяльність», «ефективна зайнятість», «вимірюваний результат» і «реальна цінність витраченого часу». У науковому дискурсі останніх років все частіше простежується тенденція до розмежування поняття активності та результативності: не кожна активність у цифровому середовищі приводить до реальної продуктивності, натомість надмірна взаємодія з інформацією часто веде до зниження ефективності та формування ілюзії досягнення [15].

Відповідно, сучасне бачення продуктивності вимагає включення параметрів ментального виснаження, швидкості переключення між задачами, стійкості уваги, якості концентрації, часу на відновлення та емоційного комфорту. Усе це зумовлює зміну парадигми від кількісного обліку дій до якісного аналізу процесу і впливу активності на загальне функціонування особистості. Окрему увагу заслуговує теоретичне осмислення взаємозв'язку між продуктивністю та когнітивною екологією – дисципліною, що досліджує вплив середовища на структуру мислення. У цифровому контексті йдеться про організацію цифрового простору, який або сприяє, або перешкоджає продуктивності через візуальне перевантаження, фоновий шум, часті сповіщення, алгоритми відволікання уваги тощо.

Цей підхід набуває все більшої актуальності в зв'язку з концепціями гігієни уваги, мінімізації перешкод, архітектури вибору в інтерфейсному дизайні. Технології самі по собі не є нейтральними – вони модифікують структури діяльності людини, її часові уподобання, емоційний фон, ритм взаємодії з реальністю. Тому продуктивність у цифрову добу – це не тільки здатність до самоуправління, але й вміння адаптуватися до техносередовища, використовувати його можливості для розвитку та контролювати ризики

надмірного залучення. При цьому велике значення має здатність особистості розпізнавати механізми впливу цифрового середовища на власну поведінку, що потребує високого рівня цифрової грамотності, критичного мислення та розвитку метапізнавальних навичок.

У науковому контексті продуктивність життєдіяльності в цифровому середовищі дедалі частіше осмислюється через призму інтегрованих моделей, які поєднують елементи тайм-менеджменту, когнітивної науки, психології праці, соціальної екології та UX-дизайну. Так, в основі сучасного підходу лежить трикомпонентна модель продуктивності: когнітивна (спроможність до концентрації, мислення, планування), емоційна (регуляція емоцій, мотивація, уникнення вигорання) та технологічна (використання відповідних інструментів і середовищ) [20].

Згідно з цією моделлю, досягнення високої продуктивності можливе лише за умови одночасного розвитку всіх трьох напрямів. З практичної точки зору, це означає, що використання лише одного трекера задач або календаря без відповідної ментальної підготовки та регуляції емоційного стану не дає очікуваного ефекту. Натомість, інтегровані системи, які дозволяють не лише фіксувати події, а й оцінювати рівень концентрації, настрій, втому, мотиваційний стан – є більш ефективними у довгостроковій перспективі.

У цьому контексті набуває поширення застосування багатовимірних web-застосунків, що орієнтовані на аналітику життєвої активності користувача: час активностей, динаміка задач, емоційні позначки, рівень навантаження, графіки продуктивності тощо. Такі системи, що працюють на базі персоналізованих алгоритмів, вже не просто обчислюють витрачений час, а й надають рекомендації, прогнозують потенційні зниження ефективності, спрямовують на оптимізацію режиму роботи [37].

У свою чергу, розробка таких систем вимагає врахування теоретико-методологічних основ, серед яких ключовими є: системний підхід, теорія саморегуляції, модель виконавчих функцій, принципи UX-дизайну та етичні аспекти персоналізованих рішень. Важливо зазначити, що ефективність web-

застосунків значною мірою залежить від ступеня їхньої адаптивності до індивідуальних характеристик користувача – його хронотипу, ритму роботи, рівня цифрової компетентності, типу мотивації.

Саме тому сучасні концепції продуктивності вимагають формування динамічних моделей, що здатні змінюватися в залежності від накопиченого досвіду користувача, реагувати на зміни в його поведінці та пропонувати релевантні варіанти дій. Таким чином, у цифровому середовищі продуктивність життєдіяльності перестає бути універсальною категорією і набуває індивідуалізованого, адаптивного та змінного характеру. Водночас це створює передумови для розробки концептуальних рамок нової інформаційно-поведінкової екосистеми, у межах якої взаємодія людини з технологіями стає не лише інструментом управління, але й засобом самопізнання, розвитку та підтримання психофізіологічного балансу [5].

У результаті, сучасне тлумачення продуктивності життєдіяльності в цифровому середовищі базується на ідеї цілісної особистісної ефективності, що охоплює здатність управляти інформацією, емоціями, ресурсами часу та енергії в умовах безперервної цифрової взаємодії, змінного середовища та зростаючих когнітивних вимог. Це визначає теоретичну та практичну важливість даної теми, а також необхідність глибокого аналізу механізмів, що формують продуктивність на різних рівнях – індивідуальному, технологічному, середовищному – з метою побудови ефективних Web-застосунків, здатних не просто обліковувати активність, а й сприяти формуванню сталих стратегій успішної життєдіяльності у цифрову добу.

Моніторинг особистої ефективності в умовах цифрового середовища є складовим елементом процесу самоменеджменту, спрямованого на підвищення результативності особистості в професійній, навчальній, побутовій та соціальній сферах. Під цим поняттям розуміють систематичне спостереження, фіксацію, аналіз і інтерпретацію дій, рішень, психофізіологічних та емоційних станів людини, які впливають на досягнення поставлених цілей. Моніторинг забезпечує об’єктивну картину реального використання часу, ресурсів і зусиль, дозволяє

виявити неефективні стратегії поведінки, сформулювати шляхи їх оптимізації, а також підтримувати мотиваційний рівень на основі зворотного зв’язку. З огляду на різноманіття підходів і методів, класифікація інструментів моніторингу особистої ефективності є необхідною для систематизації знань у цій сфері, вибору релевантних засобів вимірювання, а також для подальшої розробки інтегрованих цифрових рішень. Загалом класифікація методів моніторингу може здійснюватися за кількома ознаками: за формою збору інформації (суб’єктивні та об’єктивні), за способом представлення даних (кількісні та якісні), за рівнем автоматизації (ручні, напівавтоматизовані, повністю автоматизовані), за змістовим наповненням (поведінкові, когнітивні, емоційні, фізіологічні), за рівнем інтерактивності (пасивні та активні), а також за спрямованістю (ретроспективні, поточні, прогностичні) [6].

Суб’єктивні методи базуються на самооцінці та самоспостереженні. Вони включають ведення особистих щоденників, чек-листів, самоаналіз виконаних завдань, оцінку продуктивності за шкалами, заповнення форм зворотного зв’язку. До цієї категорії належать методи «Pomodoro», «Eisenhower Matrix», «Bullet Journal», техніки оцінки енергії та задоволеності, а також щоденна рефлексія. Перевага суб’єктивних методів полягає в їхній доступності, гнучкості та можливості персоналізації. Проте вони мають обмеження щодо об’єктивності, надійності оцінок, схильності до викривлення даних під впливом емоцій або втоми. Об’єктивні методи, навпаки, ґрунтуються на цифровій фіксації активності за допомогою програмного забезпечення, мобільних додатків, Web-розширень, сенсорів, трекерів. До них належать тайм-трекери (Toggl, Clockify, RescueTime), системи керування завданнями (Todoist, Asana, Trello), інструменти трекінгу фізичної активності (Fitbit, Apple Health, Google Fit), а також платформи, що здійснюють аналіз поведінкових патернів, рівня залучення, кількості перемикань між задачами тощо. Дані методи відзначаються високою точністю, деталізацією, можливістю автоматичного збору інформації, що мінімізує людський фактор. Проте їхній недолік – це обмеженість у фіксації нематеріальних, контекстуальних і емоційних аспектів діяльності [4].

За типом даних методи моніторингу поділяються на кількісні та якісні. Кількісні методи передбачають облік часових витрат, кількості завершених завдань, показників продуктивності, рівня виконання плану, швидкості реакції, частоти перерв, продуктивних/непродуктивних дій. На рисунку 1.1 наведена кількісна оцінка використання методів моніторингу особистої ефективності.

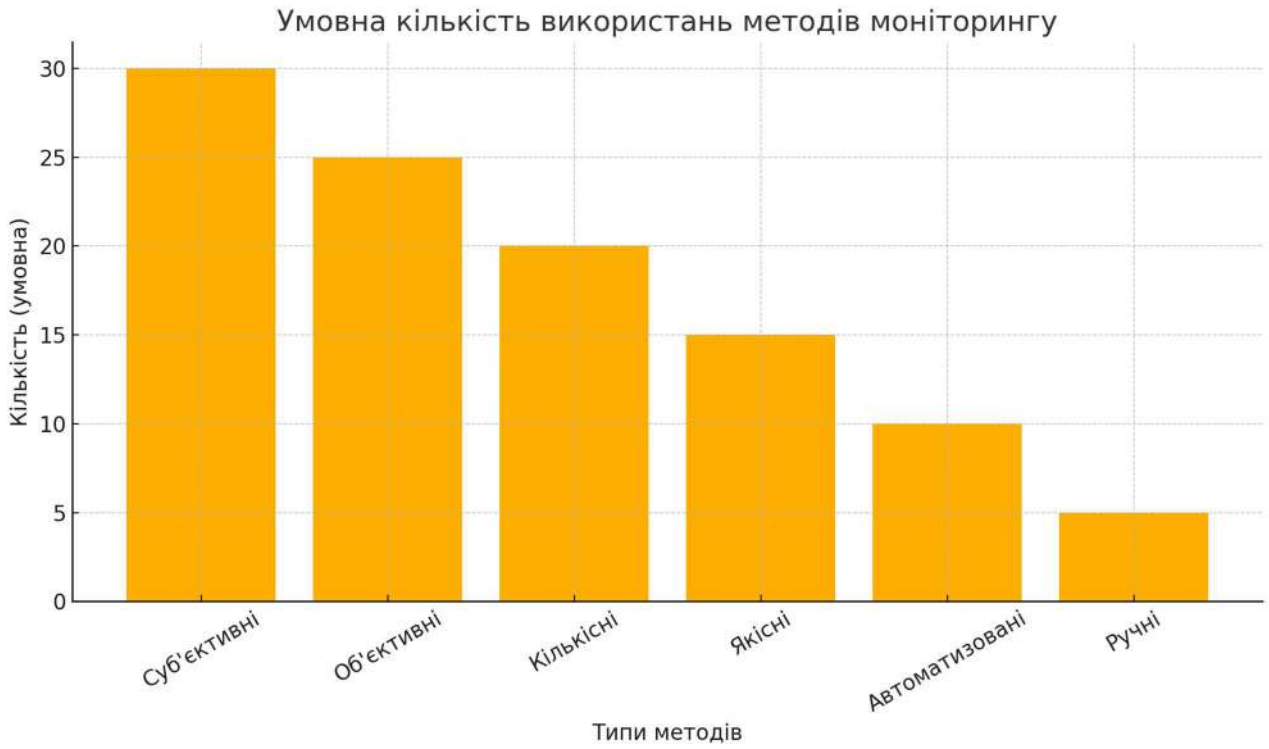

Рис. 1.1. Кількісна оцінка використання методів моніторингу особистої ефективності

Якісні методи орієнтовані на глибинний аналіз змісту діяльності, рівня задоволення результатом, впливу дій на загальний стан, відповідність завдань особистим цілям, структурність мислення та мотиваційну залученість. Важливою особливістю якісних методів є їхня здатність враховувати контекст, що є критичним у сучасних умовах динамічного навантаження та ментального багатозадачного середовища. За рівнем автоматизації методи моніторингу поділяються на ручні (заповнення таблиць, ведення записів, ручний підрахунок часу), напівавтоматизовані (використання шаблонів, форм, додатків із частковою

автоматизацією) та повністю автоматизовані (постійний фоновий збір даних, аналіз активностей без участі користувача). Останні забезпечують найвищий рівень об’єктивності та зручності, проте іноді можуть сприйматися як інвазивні або викликати психологічний дискомфорт через постійну фіксацію дій. За змістом інформації виокремлюють поведінкові (відстеження дій, темпу роботи, часу на завдання), когнітивні (оцінка ментальних процесів, фокусування, здатності до планування), емоційні (вимірювання настрою, стресу, мотивації) та фізіологічні (пульс, якість сну, втома). Кожен з наведених типів методів моніторингу дозволяє відстежувати певний аспект ефективності, проте тільки їхня комбінація може забезпечити цілісну картину продуктивності життєдіяльності особистості в цифровому середовищі [7].

1.2 Аналіз існуючих систем і технологій розробки

У сучасних умовах цифровізації суспільства Web-рішення, спрямовані на моніторинг, аналіз і підвищення особистої ефективності, посідають важливе місце в інфраструктурі самоменеджменту. Протягом останнього десятиліття відбувся стрімкий розвиток різноманітних онлайн-сервісів, застосунків і платформ, що забезпечують користувачам можливості для організації часу, контролю за виконанням завдань, аналізу поведінкових патернів, ведення особистої статистики, оцінки емоційного стану та індивідуального прогресу. Порівняльна характеристика наявних Web-рішень у сфері моніторингу особистої ефективності має на меті не лише виявлення відмінностей між окремими продуктами, але й формування цілісного уявлення про їхню функціональну повноту, гнучкість, адаптивність, масштабованість, інтеграційні можливості та загальний рівень користувацького досвіду (UX). Враховуючи широке різноманіття таких інструментів, доцільним є систематизований підхід до порівняння, який передбачає аналіз за ключовими параметрами: функціональне призначення, інтерфейсні рішення, глибина аналітики, можливість персоналізації, доступність для різних типів пристроїв, підтримка синхронізації

з іншими сервісами, рівень безпеки даних, наявність інтелектуальних підсистем. Для аналізу було обрано низку найбільш популярних і концептуально різних за підходом рішень: RescueTime, Toggl, Clockify, Todoist, Notion, Habitica, Google Calendar, Microsoft To Do та Forest. Ці сервіси, хоча й мають спільну мету - підвищення ефективності користувача, - суттєво відрізняються за функціоналом, логікою взаємодії та цільовою аудиторією [20].

RescueTime є прикладом повністю автоматизованого рішення, яке орієнтоване на пасивний моніторинг діяльності користувача. Система працює у фоновому режимі, відстежуючи активність у Web-браузері та додатках, аналізуючи продуктивність на основі класифікації відвідуваних сайтів. Основною перевагою цього інструменту є автоматизація збору даних, глибока аналітика, можливість побудови динамічних звітів, а також функція виявлення відволікаючих факторів. Проте обмеження полягають у невеликій гнучкості інтерфейсу, низькому рівні інтерактивності та відсутності повноцінного інструментарію для планування. Toggl, у свою чергу, є прикладом тайм-трекера з ручним керуванням. Користувач самостійно фіксує початок і завершення кожної активності, що дозволяє досягати високої точності, проте вимагає дисципліни. Toggl вирізняється простим, інтуїтивно зрозумілим інтерфейсом, функцією деталізації по проєктах, клієнтах і типах задач, підтримкою командної роботи та інтеграцією з більш ніж 100 сервісами. Основний недолік полягає в тому, що відсутність автоматизації створює ризик упущення даних. Clockify - ще один популярний сервіс трекінгу часу, який підтримує як ручний, так і автоматичний режим, а також має потужну звітність, можливість обліку годин, ставку за оплату праці, аналітику командної ефективності. Він є безкоштовним для базового функціоналу та забезпечує високу адаптивність до командних середовищ, проте не має глибокої персональної аналітики. Todoist - гнучка система управління завданнями, яка дозволяє створювати списки, нагадування, дедлайни, розподіляти задачі за пріоритетами, темами та днями. Вона підтримує систему «проект–підзавдання», інтерфейс із можливістю швидкого редагування, щоденну та тижневу статистику виконаного, а також має елементи гейміфікації.

Водночас Todoist не орієнтований на повноцінний моніторинг часу й не забезпечує глибокої поведінкової аналітики. Notion - універсальний інструмент для організації особистого і командного простору, який дозволяє створювати бази даних, дошки Kanban, вбудовані таблиці, документи, системи цілей і журналів. Його головна перевага - абсолютна кастомізація: користувач може створити власну систему обліку ефективності, поєднавши трекери, чек-листи, нотатки й динамічні таблиці. Проте Notion не пропонує вбудованого тайм-трекінгу й вимагає значних витрат часу на налаштування. Habitica є унікальним прикладом Web-застосунку, що реалізує концепцію гейміфікації продуктивності. Всі задачі перетворюються на ігрові дії, виконання яких приносить користувачу досвід, бонуси, предмети. Інструмент орієнтований на молодіжну аудиторію, поєднує моніторинг задач, мотиваційну підтримку та соціальну взаємодію через гільдії, місії, досягнення. Основний недолік - складність масштабування на складні завдання та вузький набір метрик. Forest - це приклад мобільного застосунку, що допомагає формувати навички фокусування: користувач «саджає дерево» на певний час фокусованої роботи, і якщо виходить із застосунку, дерево гине. Простий механізм стимулює зосередженість і формує візуальний ліс досягнень, однак Forest не підходить для комплексного трекінгу ефективності. Google Calendar і Microsoft To Do виступають як частина більш широких екосистем. Перший дозволяє планувати події, зустрічі, дедлайни, синхронізуючи дані між пристроями й користувачами. Другий дає змогу створювати списки задач, інтегруючи їх із поштою, календарем, сервісами Microsoft. Обидва інструменти зручні, доступні, мають мінімалістичний дизайн, проте не включають детального аналізу продуктивності й вимагають додаткових розширень для трекінгу [7].

Порівнюючи наведені Web-рішення, можна виділити кілька груп: сервіси моніторингу часу (RescueTime, Toggl, Clockify), менеджери завдань (Todoist, Microsoft To Do), гібридні простори (Notion), гейміфіковані застосунки (Habitica, Forest), органайзери-планувальники (Google Calendar). Вибір конкретного інструменту залежить від цілей користувача, його стилю мислення, потреб у візуалізації, аналітиці, мобільності, взаємодії з іншими. Сервіси, що підтримують

інтеграції, аналітику, автоматизацію та персоналізацію, мають вищий потенціал для довгострокового використання. При цьому варто враховувати такі критерії як: можливість адаптації під індивідуальні потреби, прозорість інтерфейсу, наявність зворотного зв'язку, захист персональних даних, підтримка багатоплатформеності, використання штучного інтелекту для прогнозування навантаження. У перспективі важливими стають комплексні рішення, які поєднують відстеження часу, керування задачами, емоційний моніторинг, біометричні показники та аналітику поведінкових патернів. Це означає, що розробка нових Web-рішень має базуватися на принципах мультифункціональності, контекстної адаптивності, UX-інтелектуальності та глибокої персоналізації. У результаті, порівняльна характеристика наявних інструментів дозволяє не лише здійснити обґрунтований вибір, але й формує теоретичну основу для створення нових моделей Web-застосунків, здатних задовольнити комплексні потреби сучасної людини в умовах інформаційного перевантаження, фрагментованої уваги та високої динаміки життєвих процесів [8].

Попри активний розвиток цифрових технологій і зростання кількості Web-застосунків, призначених для моніторингу й підвищення особистої ефективності, більшість із них має низку обмежень, які знижують загальну результативність їх використання. Ці недоліки мають різне походження - від архітектурних рішень і обмеженого функціоналу до психологічної несумісності з користувачем або неадекватності застосування в конкретному контексті. Типові недоліки, властиві існуючим Web-застосункам, доцільно розглядати у зв'язку з кількома групами: обмеження функціональних можливостей, недостатня адаптивність, проблеми з інтерфейсом користувача, неповнота аналітичної складової, низька інтеграція, обмежена персоналізація, перевантаження інформацією, недосконаліть мотиваційного супроводу, низький рівень конфіденційності даних та інші структурні або концептуальні обмеження [2].

Однією з найпоширеніших проблем є обмеженість функціонального покриття. Багато популярних застосунків зосереджуються лише на одному

аспекті продуктивності: трекінгу часу, створенні списків задач, формуванні календарів чи моніторингу фокусування. При цьому відсутнє комплексне охоплення різних параметрів життєдіяльності - когнітивних, емоційних, поведінкових і фізіологічних. Це створює ситуацію, коли користувачу доводиться одночасно використовувати кілька інструментів, кожен з яких працює у власній екосистемі, що не лише ускладнює щоденну практику, але й перешкоджає формуванню цілісної картини особистої ефективності. Крім того, багато Web-застосунків мають фіксований набір індикаторів та шаблонів, які не відповідають індивідуальним особливостям користувача. Відсутність можливості редагування, додавання власних метрик, кастомізації аналітики чи структури задач обмежує гнучкість інструментів і знижує мотивацію до їх регулярного використання.

Ще однією критичною проблемою є слабка адаптивність до контексту використання. У реальному житті продуктивність людини залежить від багатьох факторів: ритму доби, ментального стану, типу задач, зовнішнього оточення. Проте більшість застосунків не враховують контекстуальну змінність. Наприклад, трекери продуктивності не розрізняють, чи працює користувач у спокійній атмосфері чи в умовах багатозадачності; менеджери завдань не реагують на психологічну втому; інструменти з фіксації часу не зважають на рівень когнітивної складності роботи. Відсутність контекстуалізації призводить до зниження точності оцінок і, відповідно, до помилкових висновків. У цьому зв’язку актуальним є питання створення систем, здатних до динамічного налаштування на основі поведінкових патернів, самообучення та штучного інтелекту [4].

Недосконаліть інтерфейсу користувача також є поширеним фактором, який негативно впливає на взаємодію з цифровим інструментом. Часто спостерігається або надмірна складність навігації, або навпаки - надмірна спрощеність, що позбавляє користувача важливих функцій. Проблеми UX-дизайну включають перевантаження візуальними елементами, недружній мобільний інтерфейс, недостатньо продуману архітектуру даних, складність у пошуку необхідної інформації, неінтуїтивність кнопок і команд. Особливо це

стосується мультифункціональних платформ типу Notion, де гнучкість системи потребує високої технічної грамотності користувача, що не завжди виправдано в умовах щоденного застосування. Наявність таких бар'єрів знижує бажання постійно взаємодіяти з інструментом, а отже - нівелює потенційний ефект від його використання.

Ще один тип обмежень пов'язаний із поверхневістю аналітичного компонента. Чимало сервісів фіксують лише факт виконання завдання або витрати часу, не надаючи розширеної інтерпретації результатів. Відсутність механізмів виявлення закономірностей, динаміки змін, крос-аналізу між категоріями задач або днями тижня знижує ефективність роботи з даними. Користувач змушений самостійно інтерпретувати значення, будувати гіпотези щодо причин спаду продуктивності або порушення ритму. Деякі застосунки надають графіки або діаграми, проте не пояснюють, як ці показники співвідносяться з поведінкою людини чи зовнішніми обставинами. Тому постає необхідність у впровадженні інтелектуальних моделей аналізу, які б автоматично виявляли ризики емоційного вигорання, залежності від відволікаючих стимулів, когнітивного перевантаження тощо [7].

Окремим викликом є відсутність якісної інтеграції між платформами. Користувачі, які активно працюють з кількома застосунками, потребують зручного обміну даними між ними, єдиної синхронізації завдань, подій, нагадувань, пріоритетів. Проте навіть ті інструменти, що декларують інтеграцію, часто не реалізують її на повноцінному рівні: синхронізація є односторонньою, відбувається з затримкою або втрачає частину даних. Це створює фрагментацію цифрового простору особистості й вимагає дублювання дій, що суперечить самій ідеї ефективності. На цьому тлі актуальною стає і проблема обмеженої персоналізації. І хоча багато сучасних платформ пропонують налаштування теми, сповіщень або шаблонів задач, все ще рідкісними є системи, що враховують стиль мислення користувача, тип його мотивації (досягнення, уникнення, самореалізація), особливості сприйняття інформації (візуальне, текстове, комбіноване), ритмічну структуру дня (жайворонок чи сова). Персоналізовані

елементи, які мають прямий вплив на ефективність - наприклад, індивідуальний темп задач, рекомендовані інтервали відпочинку, типові фокус-блоки - залишаються поза увагою розробників [25].

Серйозною проблемою в контексті масового впровадження Web-застосунків є інформаційне перевантаження. У гонитві за функціональністю багато платформ пропонують десятки функцій, налаштувань, категорій, діаграм, які не лише не підвищують ефективність, а й створюють у користувача когнітивну втомлюваність, втрату орієнтації в інтерфейсі, зниження мотивації. Особливо небезпечним є надлишок сповіщень, які перебивають фокус уваги та не завжди відповідають реальній потребі користувача. У зв’язку з цим виникає потреба у впровадженні принципів інформаційної мінімізації та динамічного управління навантаженням - інструмент повинен не просто повідомляти, а прогнозувати доцільність взаємодії, адаптуючи інтенсивність залежно від стану користувача.

Останнім, але не менш важливим недоліком є питання конфіденційності та безпеки персональних даних. У багатьох застосунках обробляються чутливі дані - інформація про рутинні дії, графік роботи, місцеперебування, фізіологічний стан, навіть емоційні реакції. Проте не всі сервіси забезпечують належний рівень шифрування, прозорості політики зберігання, контролю за обміном інформацією. Деякі продукти передають дані третім сторонам для рекламних або маркетингових цілей, що суперечить очікуванням користувача про приватність і може знизити довіру до інструменту. Важливо, щоб цифрові рішення відповідали міжнародним стандартам захисту інформації (GDPR, ISO/IEC 27001), мали чітко сформульовані політики, гнучкі налаштування приватності, можливість вибору рівня доступу до даних [1]. На рисунку 1.2 наведений рівень вираженості типових недоліків існуючих Web-застосунків.

У підсумку, незважаючи на широкий вибір інструментів для моніторингу особистої ефективності, більшість наявних Web-застосунків має типові обмеження, які знижують їхню практичну цінність. До них належать вузькість функціоналу, відсутність контекстуального аналізу, слабка аналітика, низький

рівень персоналізації, перевантаження інформацією, недостатня інтеграція та ризики щодо безпеки даних.

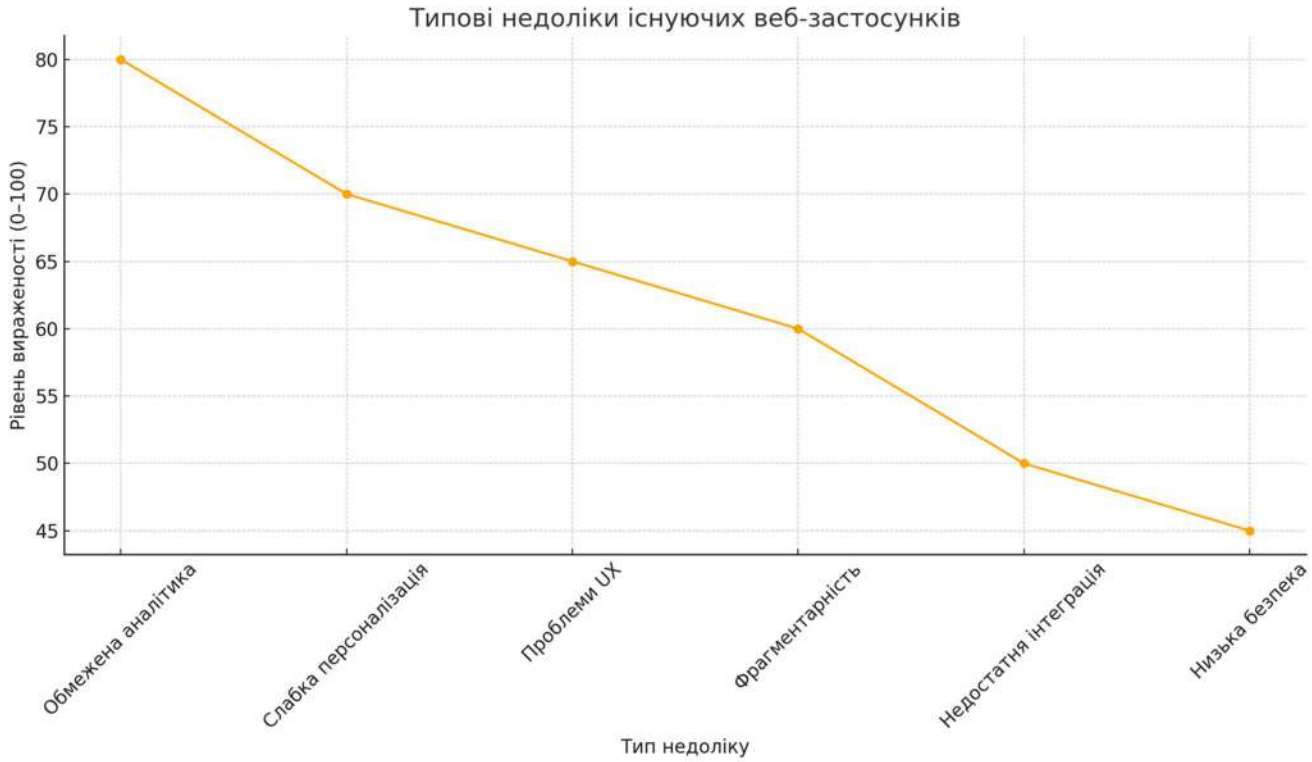

Рис. 1.2. Рівень вираженості типових недоліків існуючих Web-застосунків

Для подолання цих проблем необхідно розробляти нові рішення на основі принципів адаптивності, штучного інтелекту, індивідуалізованих моделей взаємодії, цифрової гігієни та етичної обробки даних. Це дозволить створити цифрові екосистеми, які не просто обліковують дії користувача, а комплексно сприяють підвищенню ефективності, добробуту й стійкості особистості у складному інформаційному середовищі.

1.3 Вибір технологічного стеку та архітектури

Побудова ефективного Web-застосунку потребує не лише чіткого визначення функціонального призначення продукту, але й ретельного підходу до вибору технологічного стеку та архітектурного рішення, які мають забезпечити

стабільність, масштабованість, продуктивність, безпеку та зручність розробки. У середовищі швидкоплинної цифрової трансформації, де вимоги до програмних рішень постійно зростають, саме технологічні та архітектурні засади виступають ключовими детермінантами якості кінцевого продукту. Вибір стеку технологій передбачає визначення оптимального набору мов програмування, фреймворків, бібліотек, баз даних, серверних технологій і хмарних сервісів, які забезпечують взаємодію між клієнтом і сервером, обробку даних, авторизацію, візуалізацію, логування та інші операції. При цьому необхідно враховувати як внутрішні параметри - досвід команди, цілі проєкту, очікувані навантаження, перспективи розширення, - так і зовнішні чинники: рівень підтримки обраних технологій, наявність документації, активність спільноти та відповідність сучасним стандартам галузі [6].

Для реалізації застосунку, що спрямований на моніторинг і аналіз особистої ефективності, доцільно застосовувати фронтенд-фреймворк, здатний забезпечити високу реактивність, адаптивність до мобільних пристроїв і підтримку сучасного JavaScript-стеку. Одним із найефективніших рішень у цьому контексті є React - бібліотека, яка дозволяє створювати інтерфейси користувача з гнучкою логікою станів і повторно використовуваними компонентами. Його підтримка візуалізаційних бібліотек (наприклад, Chart.js, Recharts) та інтеграція з глобальними менеджерами стану (Redux, Zustand) дає змогу реалізувати складні структури даних, інтерактивні графіки, аналітичні панелі. З боку бекенду доцільно використовувати Node.js - серверне середовище на основі JavaScript, що забезпечує високу продуктивність у роботі з великою кількістю одночасних запитів. Завдяки асинхронній обробці подій та доступу до багатої екосистеми бібліотек (Express.js, Prisma), Node.js дозволяє гнучко будувати RESTful API або реалізовувати GraphQL-архітектуру з підтримкою авторизації, валідації, обробки помилок і логування. У разі потреби реалізації складнішої бізнес-логіки або розширення аналітичних можливостей застосунку можна доповнити серверну частину фреймворками на Python (FastAPI або Django), що підвищує гнучкість і

дозволяє легко інтегрувати модулі машинного навчання або статистичного аналізу.

Щодо вибору бази даних, раціональним рішенням є використання реляційної СКБД PostgreSQL, яка підтримує складні запити, агрегацію, транзакції, індексацію та високий рівень надійності. Вона також добре інтегрується з ORM-інструментами (наприклад, Prisma або SQLAlchemy), що прискорює розробку, забезпечує захист від SQL-ін'єкцій і дозволяє централізовано керувати моделями даних. За потреби зберігання неструктурованих або слабоформалізованих даних, а також у випадку ведення журналів активності або історичних записів, можна додатково використати NoSQL-рішення - наприклад, MongoDB. Це створює умови для реалізації гібридної моделі даних, що дозволяє краще масштабувати систему відповідно до змін структури навантаження. Для покращення продуктивності системи та зниження навантаження на основну базу даних рекомендовано реалізувати систему кешування за допомогою Redis. Цей інструмент забезпечує зберігання часто використовуваних запитів і даних у пам'яті, що суттєво пришвидшує доступ до результатів без звернення до повного циклу обробки.

З боку інтеграцій і деплою важливим компонентом є система контролю версій Git із підтримкою CI/CD-процесів через GitHub Actions або GitLab CI/CD. Це дозволяє автоматизувати тестування, збірку та розгортання компонентів Web-застосунку, забезпечуючи безперервну інтеграцію нових змін без порушення стабільності. Хостинг застосунку можна реалізувати за допомогою хмарної інфраструктури типу Vercel або Netlify для фронтенду, а також Heroku, Render або AWS EC2 для бекенду, що дозволяє масштабувати ресурси відповідно до навантаження, забезпечити захищене з'єднання (HTTPS), резервне копіювання та моніторинг. Вибір саме такого технологічного стеку дозволяє не лише задовольнити функціональні потреби застосунку, а й створити міцну інженерну базу для його довготривалої підтримки й розвитку [36].

Архітектурна модель, яка використовується в межах даного проєкту, базується на концепції розділення клієнтської та серверної частин, що

реалізується через чітко визначений інтерфейс API. Такий підхід дозволяє забезпечити максимальну гнучкість у взаємодії між компонентами, сприяє модульності системи, а також дозволяє незалежно розробляти, тестувати та масштабувати кожен рівень. Клієнтська частина (frontend) функціонує як односторінковий застосунок (SPA), що знижує навантаження на сервер і підвищує швидкість взаємодії користувача з інтерфейсом. Вона обробляє маршрутизацію, відображає компоненти, керує станом додатку та здійснює запити до API, розміщеного на сервері. Серверна частина (backend) відповідає за обробку даних, логіку авторизації, перевірку прав доступу, маніпуляції з базами даних, обчислення показників та зберігання результатів моніторингу. У таблиці 1.1 наведені дані для порівняння популярних інструментів для створення Web-додатків.

Таблиця 1.1

Порівняння популярних інструментів для створення Web-додатків

Технологія	Мова	Продуктивність	Крива навчання	Підтримка спільноти	Гнучкість
React	JavaScript	Висока	Середня	Дуже висока	Висока
Vue.js	JavaScript	Висока	Низька	Висока	Висока
Angular	TypeScript	Середня	Висока	Висока	Середня
Node.js	JavaScript	Висока	Середня	Дуже висока	Висока
Django	Python	Середня	Середня	Висока	Середня
FastAPI	Python	Висока	Низька	Середня	Висока

Особливістю цієї архітектурної моделі є можливість горизонтального масштабування, тобто додавання нових екземплярів фронтенд- або бекенд-серверів у разі зростання кількості користувачів або обсягів даних. Це досягається завдяки контейнеризації з використанням Docker, що дозволяє запускати кожен компонент у відокремленому середовищі з передбачуваною конфігурацією. Управління контейнерами може здійснюватися через Kubernetes

або Docker Compose залежно від складності проєкту. Такий підхід також забезпечує гнучке розгортання в різних середовищах - локальному, тестовому, продуктивному - без необхідності внесення суттєвих змін у конфігурацію. Для підтримки безпеки реалізується система токен-автентифікації (JWT), контроль доступу на рівні ролей, валідація вхідних даних, шифрування паролів та захист API від типових атак [9].

У межах вибраної архітектури також передбачено можливість підключення сторонніх API для збору додаткових даних, таких як інформація про фізичну активність, тривалість сну, рівень стресу або зовнішні показники календарного планування. Це дозволяє розширити функціональність застосунку без повної перебудови його ядра. За допомогою реалізованого API-інтерфейсу можливе також створення мобільних або десктопних версій клієнтів, які взаємодіятимуть із тією ж логікою на бекенді, що забезпечує гнучкість розвитку проєкту. У структурі бекенду враховано можливість логування та моніторингу (наприклад, за допомогою LogRocket, Sentry або Prometheus), що дозволяє оперативно реагувати на збої, відстежувати поведінкові аномалії й виявляти потенційні вузькі місця.

Таким чином, обґрунтований вибір технологічного стеку та архітектури забезпечує відповідність ключовим вимогам до функціональності, надійності, масштабованості й зручності використання Web-застосунку.

Технології React, Node.js, PostgreSQL, Docker, CI/CD-процеси, гнучка архітектура на основі API - це перевірені рішення, які дозволяють ефективно реалізувати Web-продукт у динамічному цифровому середовищі, зберігаючи гнучкість для подальших оновлень та розширень. Обрана архітектурна модель створює оптимальні умови для впровадження системи моніторингу продуктивності життєдіяльності, яка здатна відповідати високим вимогам сучасного користувача як у сфері функціоналу, так і з точки зору досвіду взаємодії [19].

Висновки до розділу 1

Моніторинг продуктивності в умовах цифрового середовища передбачає цілісний підхід до аналізу особистої ефективності, що охоплює когнітивні, поведінкові та технологічні компоненти життєдіяльності. Продуктивність розглядається не лише як облік витраченого часу, а як динамічна система взаємодії між особистістю та цифровими інструментами, в основі якої лежить здатність самостійно планувати, контролювати, коригувати та аналізувати власну діяльність. Систематизація методів моніторингу дозволяє виокремити найбільш релевантні інструменти залежно від типу даних, рівня автоматизації та контексту використання.

Водночас аналіз існуючих Web-застосунків виявив низку обмежень, що перешкоджають формуванню комплексного уявлення про ефективність користувача. Серед таких обмежень найчастіше зустрічаються фрагментарність функціоналу, низький рівень аналітики, складність персоналізації, перевантаження інтерфейсу та обмежена масштабованість.

Вибір технологій для створення ефективного Web-застосунку має базуватись на критеріях продуктивності, безпеки, адаптивності, інтеграційних можливостей і стабільної підтримки з боку спільноти. Найбільш обґрунтованим підходом до архітектури є розмежована модель із чітким поділом клієнтської та серверної логіки, що забезпечує гнучкість масштабування, можливість автономного розвитку компонентів і високий рівень надійності системи в цілому. Сукупність теоретичних положень дає змогу визначити концептуальну платформу для подальшої реалізації Web-застосунку, адаптованого до сучасних вимог користувачів і динаміки цифрового середовища.

РОЗДІЛ 2
ПРОЄКТУВАННЯ ТА РОЗРОБКА WEB-ЗАСТОСУНКУ

2.1 Проєктування інформаційної моделі та бази даних

Формування логічної структури даних є ключовим етапом при розробці будь-якої інформаційної системи, зокрема Web-застосунку для моніторингу та аналізу продуктивності життєдіяльності. На цьому етапі відбувається концептуалізація об’єктів, з якими працює користувач, а також встановлення зв’язків між ними відповідно до логіки взаємодії в інтерфейсі та бекенді системи. У рамках практичної реалізації Web-застосунку було визначено, що основними сутностями є: користувачі (users), записи життєдіяльності (data_entries), налаштування відображення (user_settings). Обрану структуру бази даних ілюструє ER-діаграма, яка представлена на рисунку 2.1.

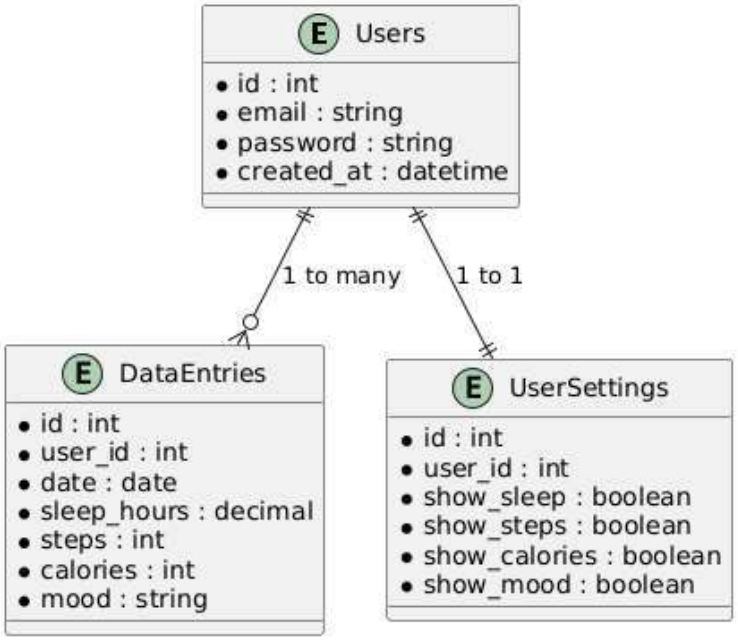

Рис. 2.1. Діаграма бази даних (ER-діаграма)

Кожна з цих сутностей виконує окрему функцію, але водночас взаємодіє з іншими через чітко задані зовнішні ключі (foreign keys), що забезпечує цілісність

даних, унеможливлює дублювання записів і дозволяє проводити агрегаційні операції на рівні бази даних. Таблиця users виступає базовою, оскільки містить унікальну інформацію про кожного зареєстрованого користувача, включаючи id, email, пароль (захешований через bcrypt), а також дату створення облікового запису. З огляду на те, що авторизація реалізована через токени (JWT), а токени зберігаються у браузері користувача (localStorage), кожна сесія взаємодії з клієнтським інтерфейсом асоціюється з ідентифікатором відповідного користувача, що уможливлює фільтрацію інформації безпосередньо в запитах SQL. Наступною важливою таблицею є data_entries, яка зберігає всі введені користувачами показники за різні дати. У структурі цієї таблиці передбачено такі поля: id запису (первинний ключ), user_id (зовнішній ключ, який посилається на таблицю users), date (дата внесення даних), sleep_hours (тривалість сну у годинах), steps (кількість кроків за день), calories (спожиті калорії), mood (настрій, виражений числом за шкалою або коментарем). Така структура дозволяє не лише зберігати показники для подальшої обробки, а й проводити сортування, групування, агрегацію та аналітичну обробку безпосередньо в MySQL. Важливою особливістю є включення до структури поля date, що є обов’язковим атрибутом для кожного запису й дозволяє вибудовувати динамічні графіки, фільтрувати записи за тиждень, місяць чи довільний період у межах аналітичного дашборду.

У процесі розробки логічної моделі даних було також враховано необхідність індивідуалізації відображення інформації в інтерфейсі користувача, що привело до створення ще однієї таблиці - user_settings. Ця таблиця дозволяє зберігати специфічні налаштування кожного користувача щодо того, які саме показники він хоче бачити на головному екрані дашборду: сон, настрій, кроки, калорії тощо. У структурі таблиці передбачено id, user_id, а також набір булевих значень, які вказують, чи відображати певний параметр. Це рішення реалізоване через маршрут /api/settings/:userId, що дозволяє через GET-запит отримати актуальні налаштування, а через POST-запит - оновити їх, забезпечуючи персоналізацію відображення у React-компонентах. Таким чином, структура

даних була побудована за принципами нормалізації, де кожна таблиця відповідає за окремий логічний аспект системи, що знижує надмірність інформації та полегшує обслуговування бази. З точки зору зв'язків між таблицями, модель є класичною: users (1) - (∞) data_entries, users (1) - (1) user_settings. Такі зв'язки чітко задаються у SQL через зовнішні ключі, що гарантує відповідність даних при кожному збереженні або читанні. У межах бекенд-реалізації структура даних відображена у вигляді моделей (наприклад, dataEntry.js, userModel.js), які інкапсулюють SQL-запити й надають контрольерам засоби для взаємодії з базою даних без дублювання коду. Водночас, для підтримки гнучкості системи передбачено можливість розширення таблиць - наприклад, додавання нових типів життєвих показників, таких як "рівень стресу", "якість сну", "тривалість фізичної активності", які можуть бути легко інтегровані в наявну структуру через розширення таблиці data_entries. Отже, логічна структура нашої системи була побудована відповідно до принципів модульності, нормалізації, масштабованості та узгодженості з архітектурою REST API, що створило фундамент для ефективної, безпечної та стабільної роботи Web-застосунку у реальних умовах використання.

База даних MySQL, розроблена для нашого web-застосунку з моніторингу та аналізу продуктивності життєдіяльності, була спроєктована як реляційна система, оптимізована під зберігання, агрегацію та фільтрацію персоніфікованих записів користувачів. Структура бази є централізованою та повністю адаптованою до REST-архітектури бекенду, розгорнутого на основі Node.js з Express. Основною метою створення цієї бази було забезпечення можливості довгострокового накопичення даних, їх обробки та формування статистичних звітів за індивідуальними параметрами, а також підтримка персоналізації інтерфейсу користувача. В межах реалізації використовувалися чотири ключові таблиці: users, data_entries, user_settings і допоміжна таблиця migrations(опціонально, при використанні систем ініціалізації схеми). Створення структури бази здійснювалося вручну через SQL-запити, використовуючи синтаксис, оптимізований під типи даних, які передбачено

зберігати, а також із урахуванням первинних і зовнішніх ключів для забезпечення логічної цілісності.

Таблиця users виконує роль центральної, оскільки всі інші сутності бази прямо чи опосередковано пов'язані з нею через поле user_id. Поля таблиці включають: id (INT, PRIMARY KEY, AUTO_INCREMENT), email(VARCHAR(255), UNIQUE, NOT NULL), password (VARCHAR(255), NOT NULL), created_at (DATETIME, DEFAULT CURRENT_TIMESTAMP). Поле password зберігає зашифрований хеш паролю, створений за допомогою бібліотеки bcrypt, а поле created_at фіксує момент створення облікового запису. Усі подальші звернення до бази через бекенд перевіряють відповідність електронної адреси й пароля користувача в цій таблиці, що є обов'язковою частиною процесу автентифікації. Кожному користувачу може відповідати довільна кількість записів у таблиці data_entries, яка є основним джерелом збереження інформації про життєві показники. Структура цієї таблиці включає: id (INT, PRIMARY KEY, AUTO_INCREMENT), user_id (INT, FOREIGN KEY, NOT NULL), date (DATE, NOT NULL), sleep_hours (DECIMAL(3,1)), steps (INT), calories (INT), mood(VARCHAR(50) або TINYTEXT). Поле user_id посилається на id із таблиці users, забезпечуючи цілісність посилань, а також дає змогу виконувати агрегатні запити на кшталт: "Середня кількість кроків за останні 7 днів для користувача N" або "Найнижчий настрій за поточний місяць". Всі поля в таблиці data_entries відповідають полям із форми введення в інтерфейсі сторінки DataEntry, а їх наявність у структурі дозволяє відразу після POST-запиту з React-застосунку зберігати дані без необхідності додаткового перетворення. Для цього в backend-контролері dataController.js реалізовано метод, що приймає ті самі поля, що й таблиця в базі даних, і виконує SQL-запит типу INSERT.

Окремо важливою є таблиця user_settings, яка дозволяє налаштовувати, які показники користувач хоче бачити в дашборді. У структурі передбачено: id (INT, PRIMARY KEY, AUTO_INCREMENT), user_id (INT, FOREIGN KEY, NOT NULL), show_sleep (BOOLEAN DEFAULT TRUE), show_steps (BOOLEAN DEFAULT TRUE), show_calories (BOOLEAN DEFAULT TRUE), show_mood

(BOOLEAN DEFAULT TRUE). Ці поля представляють булеві прапорці, які фронтенд (React) зчитує при завантаженні Dashboard-компоненту. Таким чином, користувач може через інтерфейс налаштувань (Settings.js) вибрати, які саме графіки або індикатори будуть відображатися. При збереженні цих налаштувань викликається POST-запит до /api/settings/:userId, який змінює значення в базі відповідної строки в user_settings. Логічна структура таблиці дозволяє легко масштабувати інтерфейс - наприклад, додати новий показник hydration_level, для чого достатньо додати нове булеве поле в цю таблицю і відповідний прапорець у формі на фронтенді. Завдяки чіткій відповідності між даними бази й візуальним представленням на клієнтській частині, структура user_settings дозволяє гнучко управляти виглядом інтерфейсу під потреби конкретного користувача, що є важливим елементом UX-адаптації.

Усі зв’язки в базі даних побудовані за принципом «один до багатьох» (1:N) або «один до одного» (1:1), при цьому використовуються зовнішні ключі для збереження консистентності. Наприклад, таблиця data_entries може містити тисячі записів для одного користувача, але кожен запис жорстко прив’язаний до існуючого user_id, без можливості створення «сирітського» запису. У разі спроби додати запис із неіснуючим user_id, MySQL відхилить транзакцію через порушення зовнішнього ключа. Це дозволяє уникати логічних помилок на рівні бекенду, зменшує потребу в надлишковій перевірці запитів і гарантує надійність операцій. У деяких випадках також використовуються індекси для прискорення вибірок - зокрема, на полях user_id і date у таблиці data_entries, оскільки вони найчастіше використовуються у WHERE-умовах запитів. Наприклад, при побудові графіків на Dashboard, SQL-запити до бекенду типу SELECT AVG(sleep_hours) WHERE user_id = X AND date BETWEEN Y AND Z виконуються дуже швидко саме завдяки правильно встановленим індексам.

Щодо технічного підключення до бази, воно реалізовано у backend-проєкті через бібліотеку mysql2, яка забезпечує обробку запитів у Node.js. Параметри підключення (host, user, password, database) винесені в .env файл, щоб захистити конфіденційні дані від публічного доступу. У файлі db.js виконується

ініціалізація підключення, після чого цей модуль імпортується у всі контролери, які працюють з базою. Крім того, для запобігання SQL-ін’єкціям у запитах використовуються підготовлені вирази (prepared statements), які дозволяють безпечно підставляти значення змінних у шаблон запиту. Це важливо для всіх точок введення, особливо при роботі з полями типу mood або calories, які заповнюються користувачами вручну. Варто також відзначити, що структура бази підтримує гнучке оновлення схеми без втрати даних - для цього використовуються міграції (опціонально), які можна реалізувати через сторонні бібліотеки типу Sequelize або Knex, хоча в нашому випадку створення й оновлення таблиць виконувалося вручну на етапі розробки. Такий підхід дозволяє повністю контролювати логіку структури й уникати неочікуваних конфліктів у продакшн-середовищі.

Реалізація сховища даних у рамках нашого web-застосунку для моніторингу продуктивності життєдіяльності передбачала створення надійного, масштабованого та безпечного середовища для зберігання персоніфікованих записів, які вводяться користувачем через інтерфейс. Як основну систему керування базами даних ми використали MySQL, що забезпечує гнучку роботу з реляційною структурою, високий рівень продуктивності та стабільну підтримку з боку спільноти. Обрана СКБД дозволила реалізувати всі вимоги до логічної моделі з точки зору зв’язків між таблицями, індексації, агрегування та захисту даних. Для підключення до MySQL з боку серверної частини, реалізованої на Node.js з Express, ми використали бібліотеку mysql2, яка дозволяє асинхронну взаємодію з базою, підтримує prepared statements для захисту від SQL-ін’єкцій і забезпечує стабільне з’єднання під час багатопотокових запитів. Весь функціонал підключення було винесено в окремий модуль, зазвичай db.js, де здійснюється ініціалізація пулу підключень з параметрами, взятими з файлу .env, що забезпечує гнучкість конфігурації та розділення коду й конфіденційних даних. У середовищі .env ми зберігали ключові змінні середовища, такі як DB_HOST, DB_USER, DB_PASSWORD, DB_NAME, які підвантажувалися через бібліотеку dotenv. Такий підхід дозволив уникнути прямого хардкоду

паролів і налаштувань у програмному коді, що є базовою практикою безпеки при роботі з production-ready середовищами.

Ініціалізація з’єднання з базою даних відбувалася один раз під час запуску сервера, а далі використання підключення передавалося до контролерів, які реалізовували логіку збереження, редагування або читання даних. Наприклад, при створенні нового запису в таблиці data_entries, контролер dataController.js виконував запит до бази з передачею змінних у підготовлений вираз, що забезпечувало і безпечність, і відповідність до схеми таблиці. Аналогічно, при запиті до дашборду (/api/dashboard/:userId) контролер dashboardController.jsвикористовував методи агрегації SQL (SUM, AVG, GROUP BY), що зчитували дані з data_entries, фільтруючи їх за user_id і діапазоном дат. Ще одним прикладом особливості реалізації є обробка налаштувань користувача через таблицю user_settings, з якої бекенд зчитував булеві параметри для кастомізації інтерфейсу. Дані запити надсилалися frontend-додатком через axios у форматі JSON, після чого зберігалися в MySQL без потреби у складній трансформації. Особливу увагу було приділено організації структури проєкту: ми розмежували логіку за каталогами routes, controllers, models, що дозволяє централізовано підтримувати доступ до сховища, забезпечуючи повторне використання SQL-запитів і мінімізацію дублювання коду. У моделях, наприклад у dataEntry.js, описувалися методи для збереження нових записів, вибірки по даті чи агрегації, а в контролерах відбувалося безпосереднє управління відповіддю на запити користувача. Такий підхід сприяв чіткій структуризації логіки доступу до сховища й забезпечив масштабованість системи.

Оскільки Web-застосунок передбачає постійний обмін даними між клієнтом і сервером, важливим аспектом стало ефективне управління підключенням, уникнення перевантаження бази та своєчасне закриття з’єднань. Для цього замість одиничного підключення було налаштовано пул з’єднань (createPool()), який дозволяє одночасно обслуговувати декілька клієнтських запитів без повторного створення нового з’єднання кожного разу. Усі запити до бази здійснювались через пул, що підвищило ефективність використання

ресурсів сервера та покращило продуктивність при одночасному надходженні запитів, наприклад, при відкритті дашборду одразу кількома користувачами. Для зменшення навантаження на базу при багаторазовому зверненні до одних і тих самих даних (наприклад, повторна побудова одного й того ж графіка) у подальшому планується використання систем кешування (наприклад, Redis), але на поточному етапі всі дані формуються в реальному часі на основі SQL-запитів. Це дозволяє гарантувати актуальність інформації, але водночас вимагає продуманої структури сховища, щоб запити були оптимізованими - наприклад, для щоденних даних були передбачені індекси по date і user_id, що суттєво знижує час вибірки. Загалом, використану структуру Web-додатку добре ілюструє діаграма компонентів, яка представлена на рисунку 2.2.

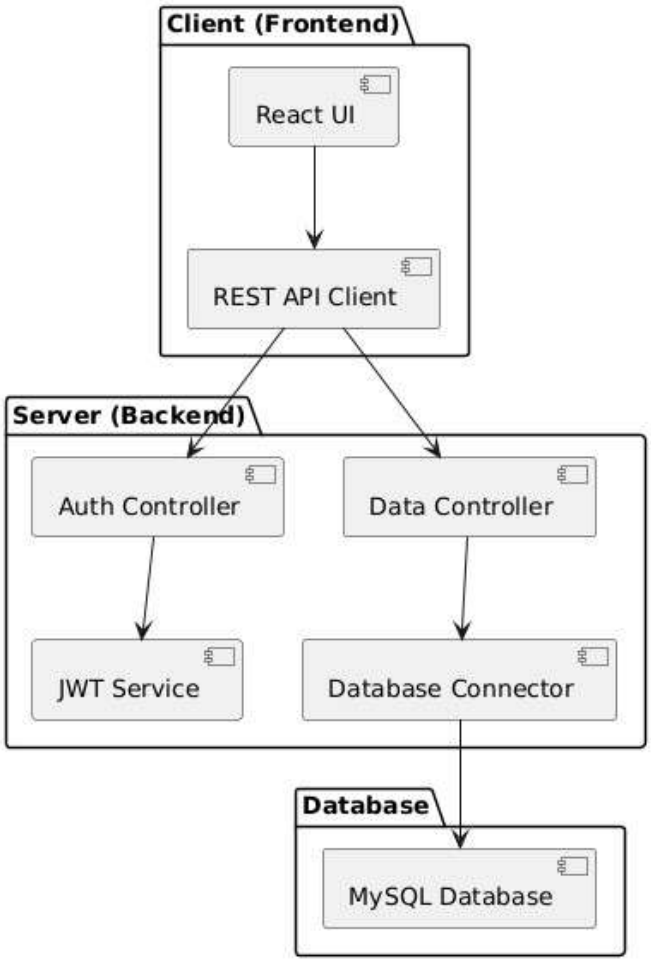

Рис. 2.2. Діаграма компонентів Web-додатку

Особливої уваги потребувала також безпека сховища. Крім збереження паролів у зашифрованому вигляді (bcrypt.hash()), були реалізовані заходи із захисту від спроб SQL-ін'єкцій: усі запити формувалися виключно як параметризовані, з явним визначенням типів полів і екрануванням значень. Авторизація запитів відбувалася через JWT (JSON Web Token), який додавався в заголовок кожного запиту з frontend-частини; бекенд перевіряв дійсність токена перед виконанням SQL-запитів до data_entries або user_settings. При цьому, розмежування прав доступу передбачало наявність двох ролей User та Admin, що ілюструє діаграма прецедентів, представлена на рисунку 2.3.

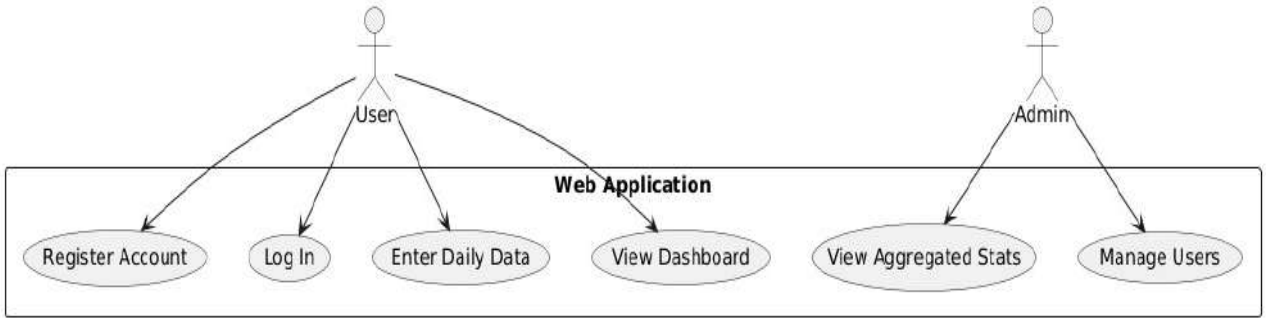

Рис. 2.3. Діаграма прецедентів Web-додатку

Таким чином, будь-який запит до сховища був не лише валідаційно перевірений, а й ідентифікований через унікального користувача. У разі відсутності токена або його недійсності запит не допускався до обробки. З точки зору розгортання, база даних на стадії розробки функціонувала на локальному сервері, але структура її підключення дозволяє легко перенести систему на хмарну платформу без зміни основної логіки підключення - достатньо замінити змінні в .env. Така гнучкість важлива для майбутнього масштабування, особливо якщо кількість користувачів перевищуватиме десятки або сотні, а також для впровадження резервного копіювання та реплікації бази на декількох вузлах.

Загалом, реалізація сховища даних і підключення до нього в нашому проєкті була побудована на основі принципів безпеки, ефективності, масштабованості та відповідності до сучасних архітектурних підходів.

Застосування MySQL як основної СКБД забезпечило структуроване зберігання життєвих показників користувачів і дозволило реалізувати аналітичну частину системи без додаткових шарів обробки. Завдяки використанню mysql2, dotenv, пулів з'єднань і централізованих моделей даних, серверна частина працює стабільно, обробляє запити за мілісекунди та гарантує точність обчислень, яка критично важлива при формуванні звітів, дашбордів і прогнозів на основі введених даних. Водночас гнучка структура підключення дозволяє інтегрувати додаткові сервіси в майбутньому, зокрема кешування, журналювання транзакцій або зовнішні API для отримання біометричних даних. Таким чином, сформована логіка роботи з базою даних та її підключенням є надійним фундаментом для подальшого розвитку функціоналу та переходу до розширених сценаріїв взаємодії користувача з системою.

2.2 Опис основних алгоритмів та методів аналізу даних

Алгоритми збору та зберігання даних у розроблюваному Web-застосунку для моніторингу продуктивності життєдіяльності були реалізовані з урахуванням вимог до простоти взаємодії для користувача, безперебійності передачі інформації між клієнтською і серверною частинами, а також з дотриманням принципів безпеки, структурованості та масштабованості. Основним джерелом даних у системі є сам користувач, який за допомогою форми введення на сторінці DataEntry вводить персональні щоденні показники: кількість годин сну, кількість зроблених кроків, кількість спожитих калорій, а також оцінку свого настрою. З технічної точки зору збір даних розпочинається з клієнтського рівня, де у React-застосунку реалізована форма з відповідними полями введення. Ці поля пов'язані зі станом компонента (через useState), і кожна зміна відображається у поточному об'єкті даних, що готується до відправки. Коли користувач натискає кнопку «Зберегти», усі введені значення збираються в JSON-об'єкт і передаються через axios.post на бекенд за маршрутом /api/data/, де обробляються сервером, перевіряються та зберігаються до бази MySQL.

На стороні бекенду запит із клієнтської частини обробляється відповідним маршрутом, реалізованим у dataRoutes.js, який спрямовує його до контролера dataController.js. Контролер у свою чергу виконує кілька ключових операцій: по-перше, валідацію вхідних даних (перевірка на наявність усіх обов'язкових полів, таких як user_id, date, sleep_hours, steps, calories, mood, а також правильність формату - наприклад, що кількість сну не може бути від'ємною, кроки та калорії мають бути цілими числами, а настрій - текстовим або числовим індикатором); по-друге, перетворення значень до форми, що відповідає типам полів у таблиці бази даних (наприклад, sleep_hours як DECIMAL, steps як INT); і по-третє, формування підготовленого SQL-запиту до таблиці data_entries, в якому підставляються відповідні значення. У випадку успішного збереження користувач отримує JSON-відповідь з підтвердженням, яка використовується на фронтенді для повідомлення про результат (наприклад, «Дані збережено успішно»).

Однією з ключових особливостей алгоритму збору даних є прив'язка кожного запису до конкретного користувача, що гарантує персоналізованість інформації й дозволяє уникати змішування записів між різними акаунтами. Коли користувач авторизується через форму входу (Login.js), він отримує токен авторизації (JWT), який зберігається у localStorage та додається до заголовків подальших запитів на бекенд. При кожному збереженні нового запису бекенд перевіряє цей токен, витягує з нього user_id і додає його до SQL-запиту. Завдяки такій структурі забезпечується безпека, а також автоматичний зв'язок між таблицею users і data_entries. Усі записи автоматично асоціюються з конкретним користувачем, без потреби вручну вказувати ID, що виключає можливість некоректного збереження чи підміни даних.

Варто підкреслити, що дата кожного запису визначається явно як окреме поле date у структурі бази, і вона може вводитися користувачем вручну або встановлюватися автоматично (якщо поле залишене порожнім). Це дозволяє системі будувати довготривалу хронологію показників, здійснювати вибірки за періодами (тиждень, місяць, конкретна дата) та формувати аналітичні графіки.

Саме це поле лежить в основі агрегацій у дашборді: при запиті до /api/dashboard/:userId бекенд здійснює SQL-запит з групуванням по датах або обчисленням середніх/сумарних значень, використовуючи WHERE user_id = ? AND date BETWEEN ? AND ?. Такий підхід дозволяє побудувати динамічну систему аналізу продуктивності, яка базується на реальних історичних даних і адаптується до вибраного інтервалу.

Для ефективності збору інформації важливим аспектом стало запобігання дублюванню записів. Щоб уникнути випадкового повторного збереження даних за ту саму дату, ми реалізували перевірку на унікальність запису на рівні бекенду: перед INSERT-запитом контролер перевіряє, чи вже існує в базі запис із заданою датою для поточного користувача. Якщо такий запис знайдено, система може запропонувати або оновити наявний (через UPDATE), або відхилити запит із відповідним повідомленням. Це дозволяє уникати накладень і втрати цілісності даних. У майбутньому така логіка може бути винесена на окремий рівень з можливістю редагування існуючих записів користувачем із фронтенду, однак навіть на поточному рівні система вже забезпечує контроль за структурою даних.

Ще однією важливою частиною алгоритму збору даних є обробка виключень і повідомлень про помилки. Якщо під час запиту трапляється помилка - наприклад, втрата з’єднання з базою даних, відсутність обов’язкових параметрів, помилка авторизації - користувач отримає відповідне повідомлення (400, 401 або 500 з описом). Це реалізовано через систему обробки помилок у Express.js, де у кожному контролері є try-catch блок із відповідним кодом статусу і JSON-відповіддю. Така реалізація дозволяє фронтенду грамотно реагувати на нештатні ситуації й формувати користувацький досвід із мінімальним рівнем фрустрації.

Методи агрегації та обробки показників у рамках реалізації Web-застосунку для моніторингу продуктивності життєдіяльності є одним із ключових функціональних компонентів системи, адже саме на їхній основі формується статистика, генеруються графіки та здійснюється подальший аналіз введених користувачем даних. Зібрані показники (тривалість сну, кількість

кроків, калорійність споживаної їжі, оцінка настрою) не є самоціллю - їх обробка дозволяє побачити динаміку, відслідковувати зміни стану здоров’я чи продуктивності з часом, а також здійснювати порівняльний аналіз періодів. У нашому проєкті всі ці дії реалізуються на рівні backend-серверу на базі Node.js із використанням Express та бази даних MySQL. Користувацькі запити надходять через маршрут /api/dashboard/:userId, і відповідний контролер (dashboardController.js) здійснює SQL-запити з агрегаційними функціями AVG, SUM, COUNT, MIN, MAX, які дозволяють обчислювати середні значення, сумарні показники, загальну кількість записів тощо. Наприклад, для побудови графіка динаміки сну за останній тиждень виконується запит, що групує значення sleep_hours за полем date і фільтрує їх за user_id, що дозволяє отримати масив об’єктів з датами та відповідними значеннями. Цей масив потім передається у фронтенд для відображення у вигляді лінійного графіка за допомогою бібліотеки Chart.js.

На стороні бази даних усі агрегаційні операції реалізовано безпосередньо в SQL-запитах, що дозволяє знизити навантаження на клієнтську частину та уникнути непотрібної передачі великого обсягу необробленої інформації. Наприклад, щоб обчислити середню кількість годин сну за останні 7 днів, запит виглядає так: SELECT AVG(sleep_hours) FROM data_entries WHERE user_id = ? AND date BETWEEN CURDATE() - INTERVAL 7 DAY AND CURDATE(). Аналогічним чином формується запит для сумарної кількості кроків або загальної кількості спожитих калорій за період. Підготовлені вирази у Node.js (через mysql2) дозволяють безпечно передати user_id та діапазон дат як параметри. Ці значення можуть задаватися динамічно залежно від налаштувань користувача або кнопок, обраних на панелі керування - наприклад, “за день”, “за тиждень”, “за місяць”. Оброблені результати з SQL перетворюються в зручний для React формат (JSON-масив), у якому кожен елемент має структуру {date: "...", sleep_hours: ..., steps: ..., mood: ...}. Завдяки цьому дані можуть бути легко інтерпретовані компонентами графічного виводу.

Особливої уваги потребує метод агрегації настрою, оскільки цей показник має не лише кількісне, а й якісне значення. У нашій реалізації настрій зберігається або як числове значення за шкалою від 1 до 10, або як короткий текстовий опис (“ok”, “bad”, “super”). Для першого варіанту можуть застосовуватись стандартні функції AVG(mood) у разі, якщо mood представлений числом. Для текстових варіантів реалізується інша логіка: проводиться класифікація введених значень за ключовими словами або їхнє кодування (наприклад, “bad” → 2, “ok” → 5, “good” → 8), що дозволяє виконати числову агрегацію навіть для мовленнєвих форм. У майбутньому це відкриває можливість побудови настроєвих карт або автоматичної оцінки психоемоційного стану на базі частотного аналізу. Поточна реалізація обмежується відображенням середнього або останнього значення настрою, яке передається на дашборд і виводиться у вигляді текстового індикатора, підсиленого кольором (зелене - позитивне, жовте - нейтральне, червоне - негативне).

Обробка агрегованих даних на фронтенді також відіграє суттєву роль, адже саме вона відповідає за кінцеву візуалізацію показників. Після отримання з бекенду агрегованих масивів даних, компонент Dashboard.js перетворює їх на формат, зручний для побудови графіків у Chart.js або react-chartjs-2. Для цього масиви дат і значень виділяються окремо: наприклад, labels = [“2024-04-01”, “2024-04-02”, …], data = [7.5, 6.8, 8.0, …]. Така структура дозволяє зберігати відповідність між датами і значеннями та створювати лінійні графіки, гістограми чи діаграми з кількома серіями одночасно. Крім того, інтерфейс має можливість адаптації: якщо користувач у налаштуваннях вимкнув відображення певного показника (наприклад, кроків), то ці дані не включаються у запити до бекенду і відповідні графіки не відображаються. Це стало можливим завдяки взаємодії з таблицею user_settings, у якій зберігаються булеві прапорці show_sleep, show_steps, show_calories, show_mood. Ці значення враховуються при формуванні запитів до /api/dashboard, і лише вибрані поля обробляються й передаються на фронтенд.

Крім стандартних середніх значень, у майбутньому реалізація може бути розширена до обчислення ковзних середніх, відхилень від норми, порівняння показників між тижнями або сезону, що дозволить користувачу отримати глибший рівень аналітики. Наприклад, формули типу SELECT AVG(sleep_hours) OVER (ORDER BY date ROWS 6 PRECEDING) дозволять побудувати ковзне середнє по 7 днях, що згладжує коливання та демонструє загальну тенденцію. Також можливо реалізувати побудову boxplot-діаграм із підрахунком квартилей і виявленням аномальних значень, особливо це корисно при аналізі розкиду настрою або калорій. Водночас, уже на поточному етапі застосунок забезпечує базову аналітичну функціональність, якої достатньо для користувача-початківця, що прагне відслідковувати свої показники у щоденному режимі.

Інтеграція аналітики у REST API є фундаментальним етапом реалізації повноцінної системи моніторингу життєдіяльності, яка не лише зберігає введені користувачем дані, але й забезпечує їх подальшу інтерпретацію та відображення у вигляді візуалізованої статистики. У рамках реалізованого нами web-застосунку аналітична логіка була інтегрована безпосередньо у backend-частину, побудовану на основі Express.js. Ключовим маршрутом, який відповідає за повернення агрегованих статистичних даних, є /api/dashboard/:userId, де :userId є динамічним параметром, що ідентифікує конкретного користувача. При надсиланні GET-запиту на цей маршрут, контролер dashboardController.js дійснює кілька важливих операцій: зчитування параметра userId, перевірка прав доступу (через JWT або cookie), формування SQL-запитів до бази MySQL з агрегаційними функціями, обробка отриманих результатів та повернення їх у форматі JSON у структурі, зручній для обробки на клієнтській частині. Саме цей процес і є прикладом повноцінної інтеграції аналітичних механізмів у REST API, оскільки відбувається не лише отримання даних, а й їхній попередній аналіз і агрегування безпосередньо на сервері.

Після надсилання запиту контролер формує відповідний SQL-запит, у якому за допомогою агрегатних функцій AVG, SUM, MAX, MIN, COUNT, а також GROUP BY date або ORDER BY date DESC LIMIT N витягуються

усереднені показники сну, кроків, калорій і настрою користувача за обраний період. Наприклад, щоб побудувати графік тривалості сну за тиждень, бекенд виконує запит на кшталт: SELECT date, sleep_hours FROM data_entries WHERE user_id = ? AND date BETWEEN CURDATE() - INTERVAL 7 DAY AND CURDATE(). Отриманий масив з датами та відповідними значеннями sleep_hours далі перетворюється в об’єкт, який надсилається на клієнт у такому форматі: { sleep: { labels: [...], data: [...] }, steps: { labels: [...], data: [...] }, calories: {...}, mood: {...} }. Завдяки такій структурі React-компоненти на фронтенді можуть без зайвих обчислень побудувати графіки за допомогою Chart.js. Інтеграція аналітики в API полягає саме в тому, що дані передаються вже агрегованими, очищеними та впорядкованими, що дозволяє знизити навантаження на клієнтську частину й прискорити візуалізацію.

Особливу увагу при реалізації аналітики в REST API ми приділили модульності й масштабованості контролера. Весь аналітичний функціонал винесено в окремі функції всередині dashboardController.js, де кожна з них відповідає за обробку одного з показників - наприклад, getAverageSleep, getTotalSteps, getCaloriesTrend, getMoodStats. Це дозволяє легко підтримувати код, додавати нові метрики або змінювати алгоритми без порушення логіки всього контролера. Якщо в майбутньому виникне необхідність додати, наприклад, рівень гідратації або пульс, достатньо буде створити нову функцію getHydrationTrend і додати її виклик у загальному обробнику запитів. Крім того, ми реалізували гнучкий підхід до обробки параметрів періоду: клієнтський інтерфейс може передати додаткові параметри, такі як ?period=7d або ?start=2024-04-01&end=2024-04-07, і бекенд відповідно скоригує SQL-запит. Це забезпечує кастомізацію запитів користувача та дозволяє створювати гнучкі аналітичні звіти.

Ще одним важливим аспектом інтеграції аналітики в REST API стало врахування налаштувань користувача, що зберігаються в таблиці user_settings. Коли користувач заходить на Dashboard, frontend спочатку надсилає запит до /api/settings/:userId, отримує перелік обраних метрик (наприклад, show_sleep:

true, show_steps: false) і вже на основі цього вирішує, які саме запити до аналітичного API надсилати. Бекенд, у свою чергу, адаптується до цих налаштувань, формуючи відповіді тільки за тими показниками, які активні. Це знижує навантаження на сервер, скорочує об’єм переданої інформації та підвищує швидкодію системи. Для користувача це виглядає як динамічна побудова дашборду відповідно до особистих вподобань - наприклад, якщо користувач цікавиться лише настроєм і сном, то він бачить лише відповідні графіки, а решта - приховані. Таким чином, REST API не лише передає дані, а адаптується до індивідуальних параметрів, що є ознакою високої інтеграції аналітичного рівня з персоналізацією.

Для побудови безпечної взаємодії між клієнтом і сервером при інтеграції аналітики було реалізовано авторизацію через JWT. Кожен запит до /api/dashboard/:userId повинен містити валідний токен у заголовку Authorization. Цей токен декодується за допомогою jsonwebtoken, і backend переконується, що запит справді належить тому користувачу, який має відповідний user_id. У разі відсутності токена або його недійсності запит не виконується, і система повертає статус 401. Це дозволяє захистити приватну аналітику користувача від стороннього доступу. Крім того, контролер перевіряє, чи user_id із токена збігається з user_id, переданим у URL, щоб виключити підміну ідентифікатора вручну.

У підсумку, реалізована інтеграція аналітики в REST API охоплює повний цикл: від надходження запиту з параметрами користувача і періоду - до формування обробленого масиву агрегованих значень, який відразу готовий до візуалізації. У розроблюваному застосунку аналітична логіка не є відокремленим модулем, а тісно пов’язана з усіма рівнями архітектури - базою MySQL, обробниками у Node.js, інтерфейсними компонентами React. Завдяки використанню SQL-агрегації, динамічних маршрутів, авторизації через JWT і підтримки користувацьких налаштувань, API став не просто засобом доступу до даних, а повноцінним інструментом аналітичної взаємодії між користувачем і системою. Така модель легко розширюється, масштабована та орієнтована на

майбутню інтеграцію зі сторонніми сервісами (наприклад, трекерами Fitbit або Apple Health) - для цього достатньо буде додати нові обробники в API, зберегти дані в data_entries і додати відповідні агрегаційні функції. Отже, інтеграція аналітики у REST API є не лише технічним рішенням, а й основою для цілісного цифрового досвіду користувача.

2.3 Реалізація серверної та клієнтської частини

Архітектура серверної частини у розробленому web-застосунку для моніторингу продуктивності життєдіяльності побудована на основі середовища виконання JavaScript-коду Node.js та каркасу Express.js, що забезпечує ефективну організацію маршрутизації, обробки запитів, взаємодії з базою даних MySQL, а також реалізацію механізмів авторизації, валідації й управління логікою бізнес-процесів. Основною метою серверної частини є забезпечення надійної взаємодії між клієнтським інтерфейсом (frontend) та сховищем даних, де реалізується передача, обробка, фільтрація й агрегація інформації, яку користувач вводить через форму або яку система обчислює автоматично. Архітектура побудована за принципами модульності, що дозволяє легко масштабувати код, розділяти логіку на окремі файли й каталоги, спрощуючи підтримку проєкту.

Головним елементом серверної частини є файл запуску, зазвичай server.js, у якому ініціалізується додаток Express, підключаються середовища (через dotenv), задається порт для прослуховування HTTP-запитів, додається middleware для обробки JSON-запитів, CORS, логування, а також підключаються основні маршрути. Весь код організовано так, щоб розділити загальні конфігурації, маршрутизацію, логіку обробки запитів і взаємодію з базою даних. Наприклад, підключення до бази MySQL відбувається в окремому файлі db.js, де через mysql2.createPool() створюється пул з’єднань, а параметри (host, user, password, database) беруться з .envфайлу. Це дозволяє змінювати параметри середовища без модифікації основного коду й гарантує безпеку при роботі з конфіденційною інформацією. Далі цей пул імпортується до інших модулів,

зокрема до моделей та контролерів, де використовуються асинхронні запити до бази.

Маршрутизація організована в окремому каталозі routes, де кожен функціональний модуль має власний маршрутний файл. Наприклад, у authRoutes.js описано всі маршрути, що стосуються автентифікації користувача - /api/auth/register, /api/auth/login, які надсилають запити до відповідних функцій у authController.js. Аналогічно, у dataRoutes.js описані маршрути для введення даних (/api/data/ – POST), а у dashboardRoutes.js- маршрути для отримання статистики (/api/dashboard/:userId – GET). Кожен маршрут пов'язаний із відповідним контролером, що зберігається в каталозі controllers. Контролери відповідають за логіку обробки HTTP-запитів: вони перевіряють вхідні дані, викликають методи з моделей, здійснюють SQL-запити або логіку агрегації, формують відповіді у форматі JSON. Наприклад, контролер dataController.js обробляє збереження показників у таблицю data_entries, а dashboardController.js - агрегацію та формування графічних даних.

Моделі зосереджено в каталозі models, де кожна з них описує функції для взаємодії з конкретною таблицею бази даних. Наприклад, у userModel.js реалізовано методи реєстрації користувача, перевірки електронної пошти, хешування пароля та авторизації. У dataEntry.js описано методи збереження, вибірки, пошуку записів по даті, а також перевірки наявності дубліката. Основна ідея полягає в тому, щоб усі SQL-запити були централізовані у моделях, а контролери лише викликали потрібні методи, зберігаючи чистоту та логічність коду. Такий підхід дозволяє уникати дублювання SQL-логіки у різних частинах програми, забезпечує зручну інтеграцію змін у запитах і дозволяє легко тестувати окремі функції. Зовнішні ключі, які пов'язують таблиці між собою (user_id у data_entries, user_settings), обробляються в межах цих моделей, що дозволяє гарантувати цілісність транзакцій.

Окремо варто відзначити реалізацію авторизації користувача, що є критично важливою частиною архітектури серверної частини. Ми використали бібліотеку jsonwebtoken для створення та перевірки JWT-токенів. При успішній

реєстрації або вході в систему користувач отримує токен, який зберігається у браузері на стороні клієнта (у localStorage), а при кожному подальшому запиті передається у заголовку Authorization. У серверній частині middleware-функція перевіряє наявність і валідність токена, декодує його і дозволяє доступ до захищених маршрутів, таких як /api/data/, /api/dashboard/, /api/settings/. Якщо токен недійсний або відсутній, система повертає статус 401 або 403, блокуючи несанкціонований доступ. Це дозволяє ізолювати аналітику й персональні дані кожного користувача та гарантувати їхню конфіденційність.

Ще одним важливим елементом архітектури серверної частини є обробка помилок. Для цього реалізований глобальний обробник помилок, який фіксує всі винятки, що виникають у контролерах або при запитах до бази, і повертає уніфіковану JSON-відповідь з кодом статусу, повідомленням про помилку й деталями. Це дозволяє легко діагностувати проблему на стороні клієнта і зменшує ризик втрати даних. Наприклад, при спробі зберегти запис із порушенням зовнішнього ключа (наприклад, неіснуючий user_id), система поверне код 500 з описом помилки SQL. Крім того, для налагодження архітектури ми використовували middleware morgan, який логував усі запити, їхні типи, маршрути й час обробки, що полегшувало тестування та моніторинг.

Загалом, архітектура серверної частини нашого застосунку є багаторівневою, модульною й повністю адаптованою до REST-підходу. Вона передбачає поділ на логічні компоненти (ініціалізація, маршрути, контролери, моделі), забезпечує зручне підключення до бази через пул з’єднань, реалізує надійну авторизацію, захист даних і обробку запитів з будь-якого рівня складності. Така архітектура дозволяє ефективно обслуговувати як прості запити на збереження, так і складні аналітичні запити з агрегацією, фільтрацією та групуванням. Вона забезпечує масштабованість і гнучкість системи, відкриваючи можливості для інтеграції зовнішніх сервісів, перенесення на хмарні середовища, створення API-документації (наприклад, через Swagger) та додавання нових функцій без ризику порушення існуючої логіки. Усі використані технології - Node.js, Express, mysql2, dotenv, bcrypt, jsonwebtoken, morgan - були

обрані з урахуванням стабільності, активної підтримки спільноти та відповідності сучасним стандартам розробки. Таким чином, архітектура серверної частини відіграє визначальну роль у забезпеченні функціональної повноти, стабільності та безпеки всього застосунку.

Клієнтська частина розробленого Web-застосунку для моніторингу та аналізу продуктивності життєдіяльності реалізована на основі бібліотеки React, яка забезпечує побудову швидкого, реактивного та адаптивного інтерфейсу користувача. Обрана технологія дозволяє реалізувати підхід SPA (Single Page Application), де всі переходи між сторінками здійснюються без повного перезавантаження сторінки, а лише за рахунок внутрішньої маршрутизації. Це забезпечує високу швидкодію, плавність анімацій та приємний користувацький досвід. Проєкт стартував із використанням Create React App, що дозволило одразу отримати налаштоване середовище з підтримкою hot-reload, розділенням на build та development середовища, а також готовими до використання конфігураціями Webpack і Babel. У файлі package.json було встановлено залежності, необхідні для роботи з формами, запитами, маршрутизацією та побудовою графіків, серед яких: react-router-dom, axios, @mui/material, @emotion/react, @emotion/styled, react-chartjs-2, chart.js.

Структура каталогу клієнтської частини організована за принципом розділення на функціональні компоненти. У каталозі src розміщено основні файли застосунку: index.js - точка входу, яка монтує кореневий компонент у DOM; App.js - головний контейнер, де реалізована логіка маршрутизації через BrowserRouter, Routes і Route. Кожна сторінка представлена окремим компонентом: Login.js, Register.js, Dashboard.js, DataEntry.js, Settings.js, які зберігаються у відповідних підкаталогах або спільному каталозі pages. Ці компоненти реалізують власну логіку відображення, завантаження даних з сервера, обробки форм і навігації. Наприклад, Login.js містить форму авторизації, яка зчитує введені email і пароль через useState, відправляє їх на сервер за допомогою axios.post('/api/auth/login'), отримує токен, зберігає його в

localStorage і перенаправляє користувача на дашборд. Аналогічно, Register.js реалізує форму реєстрації з відповідною валідацією.

Основна логіка взаємодії з даними відбувається на сторінці Dashboard.js, яка є центральним елементом інтерфейсу після входу в систему. Тут реалізовано запит на сервер за маршрутом /api/dashboard/:userId, який виконується в useEffect після завантаження компонента. Отримані з бекенду агреговані дані обробляються та передаються у вигляді масивів до компонентів графіків, побудованих на основі react-chartjs-2. Для кожного показника (сон, кроки, калорії, настрій) створено окремі графіки, які реагують на налаштування користувача: якщо у налаштуваннях деактивовано певний параметр, відповідний графік не відображається. Візуальна частина інтерфейсу побудована на основі Material UI (MUI), що дозволяє швидко формувати адаптивні та стилістично узгоджені елементи - кнопки, поля введення, сітки, панелі, іконки. Наприклад, форма введення даних у DataEntry.js побудована на основі компонентів TextField, Button, FormControl, що забезпечує сучасний вигляд і відповідність принципам UX-дизайну.

Окремо варто відзначити реалізацію системи введення даних у DataEntry.js, де користувач заповнює поля щодо своєї життєдіяльності за обраний день. Після натискання кнопки "Зберегти", усі дані збираються в об’єкт і надсилаються через POST-запит на /api/data/. У випадку успіху на екрані з’являється підтвердження, після чого користувач може перейти назад на дашборд. Дані, які вводяться, повністю відповідають структурі таблиці data_entries, що дозволяє зберігати їх без додаткових трансформацій. Для дати використовується компонент DatePicker, який дозволяє вибрати будь-який день, що важливо для можливості внесення даних заднім числом. Обробка стану форми реалізована через useState, що дозволяє динамічно відображати помилки або підтвердження, залежно від відповіді сервера.

Компонент Settings.js забезпечує персоналізацію інтерфейсу: користувач може обрати, які саме показники він хоче бачити на головному екрані. Для цього використано чекбокси (Checkbox із MUI), які відображають поточні значення

полів з таблиці user_settings. Дані зчитуються за допомогою GET-запиту до /api/settings/:userId, а при зміні зберігаються через POST-запит, що дозволяє адаптувати дашборд під конкретні потреби користувача. Усі ці зміни зберігаються на сервері, тому навіть при повторному вході в систему або переході на інший пристрій налаштування залишаються активними. Таким чином, React забезпечує повний цикл: від завантаження, заповнення форм, взаємодії з REST API - до відображення аналітики й персоналізації.

Для підтримки безпеки клієнтської частини ми реалізували збереження токена автентифікації в localStorage. При кожному запиті до захищених маршрутів (/api/data/, /api/dashboard/, /api/settings/) токен додається до заголовку Authorization, а у випадку його відсутності або закінчення терміну дії користувач автоматично перенаправляється на сторінку входу. Це дозволяє виключити несанкціонований доступ до персональних даних. Крім того, у App.js реалізовано захист маршрутів: за допомогою умовного рендерингу користувач не може потрапити на дашборд або форму введення даних без попередньої авторизації. Це реалізовано через перевірку наявності токена в localStorage і автоматичний редирект.

Взаємодія між frontend і backend у розробленому Web-застосунку для моніторингу та аналізу продуктивності життєдіяльності побудована відповідно до REST-архітектури, де клієнтська частина, реалізована на React, надсилає HTTP-запити до серверної частини на Node.js + Express, отримує відповіді у форматі JSON і на їх основі формує інтерфейс користувача. Така модель дозволяє чітко розділити обов’язки між обома частинами системи: frontend відповідає за збір введених даних, їх представлення, обробку подій інтерфейсу й візуалізацію аналітики, тоді як backend обробляє бізнес-логіку, забезпечує доступ до бази даних MySQL, реалізує валідацію, авторизацію та збереження інформації. Кожен запит відправляється через бібліотеку axios, а сервер відповідає структурованими JSON-об’єктами, що забезпечує стандартизовану та зрозумілу комунікацію між обома рівнями.

Одним із найважливіших сценаріїв взаємодії є реєстрація та логін користувача. На фронтенді у компонентах Register.js та Login.js реалізовано форми, які зчитують значення полів email і password через useState, після чого при натисканні кнопки надсилають POST-запит на /api/auth/register або /api/auth/login відповідно. Запит містить тіло з об’єктом { email, password }, а сервер у свою чергу приймає ці дані, обробляє їх через authController.js, перевіряє правильність даних (наявність користувача, відповідність пароля), хешує пароль через bcrypt (при реєстрації) або звіряє хеш (при логіні), після чого у відповідь надсилає або повідомлення про успіх, або токен авторизації у форматі JWT. Отриманий токен зберігається у localStorage, після чого фронтенд автоматично перенаправляє користувача на дашборд. Усі подальші запити до захищених маршрутів - наприклад, /api/data/, /api/dashboard/, /api/settings/ - включають цей токен у заголовок Authorization: Bearer <token>, що дозволяє серверу ідентифікувати користувача й надати доступ до персональних даних. На рисунку 2.4 представлений фрагмент коду, що відповідає за авторизацію та реєстрацію користувача з JWT.

У процесі щоденного використання застосунку ключовим видом взаємодії є надсилання нових записів про життєдіяльність. Користувач переходить на сторінку введення даних (DataEntry.js), де заповнює форму з показниками - тривалість сну, кількість кроків, калорій, настрій, дата - після чого натискає «Зберегти». Усі значення збираються в об’єкт і передаються через POST-запит на маршрут /api/data/. Бекенд приймає запит, перевіряє токен, дістає user_id, валідує дані (перевірка на числові значення, обмеження на допустимі діапазони), формує SQL-запит і зберігає запис у таблицю data_entries MySQL-бази. У разі успіху повертається повідомлення про збереження, а React-компонент показує відповідне сповіщення. Важливо, що ця взаємодія відбувається без перезавантаження сторінки, і завдяки використанню useState та useEffect, форма одразу реагує на зміну стану - очищається, виводить повідомлення, змінює колір кнопки тощо. Це дозволяє зробити досвід користувача швидким та інтуїтивно зрозумілим.

```
const userModel = require('../models/user');
const bcrypt = require('bcrypt');
const jwt = require('jsonwebtoken');

const JWT_SECRET = process.env.JWT_SECRET || 'defaultsecret'; // Рекомендовано задати через .env

module.exports = {
  // Реєстрація нового користувача
  register: (req, res) => {
    const { email, password } = req.body;
    if (!email || !password) {
      return res.status(400).json({ error: 'Email та пароль обов'язкові.' });
    }
    // Перевіряємо, чи існує користувач з таким email
    userModel.findUserByEmail(email, (err, user) => {
      if (err) return res.status(500).json({ error: err.message });
      if (user) return res.status(400).json({ error: 'Користувач з таким email вже існує.' });

      // Створюємо нового користувача
      userModel.createUser(email, password, (err, result) => {
        if (err) return res.status(500).json({ error: err.message });
        res.status(201).json({ message: 'Користувача успішно зареєстровано.' });
      });
    });
  },

  // Вхід користувача
  login: (req, res) => {
    const { email, password } = req.body;
    if (!email || !password) {
      return res.status(400).json({ error: 'Email та пароль обов'язкові.' });
    }
    userModel.findUserByEmail(email, (err, user) => {
      if (err) return res.status(500).json({ error: err.message });
      if (!user) return res.status(401).json({ error: 'Невірні дані для входу.' });

      bcrypt.compare(password, user.password, (err, isMatch) => {
        if (err) return res.status(500).json({ error: err.message });
        if (!isMatch) return res.status(401).json({ error: 'Невірні дані для входу.' });

        const token = jwt.sign({ id: user.id, email: user.email }, JWT_SECRET, { expiresIn: '1h' });
        res.json({ token });
      });
    });
  }
};
```

Рис. 2.4. Фрагмент коду, що відповідає за авторизацію та реєстрацію користувача з JWT

Окремо варто розглянути взаємодію при побудові аналітики. На сторінці Dashboard.js при завантаженні компонента виконується axios.get на /api/dashboard/:userId, де в заголовок додається токен для авторизації. Серверна частина (через dashboardController.js) здійснює SQL-запити з агрегацією даних - наприклад, AVG(sleep_hours) або SUM(steps) - і повертає у відповідь масиви

об’єктів, у яких кожна дата відповідає певному значенню. Наприклад, { date: "2024-04-10", sleep_hours: 7.5 }. React-компонент перетворює ці масиви у формат, зручний для Chart.js, і будує графіки. Якщо користувач у налаштуваннях (Settings.js) вимкнув певні показники (через Checkbox), то відповідні запити не надсилаються або відповіді не відображаються, що забезпечує динамічну та персоналізовану взаємодію. Зміна налаштувань відбувається через POST-запит на /api/settings/:userId, який містить структуру активних показників - наприклад, { show_sleep: true, show_mood: false }. Сервер обробляє ці параметри й оновлює таблицю user_settings, щоб у майбутньому автоматично застосовувати інтерфейс під вибір користувача.

З технічної точки зору, взаємодія між фронтендом і бекендом базується на принципі запит-відповідь, де всі дії чітко структуровані за методами HTTP - GET для отримання даних (наприклад, дашборд, налаштування), POST для створення нових записів (дані, користувач), PUT або PATCH для оновлення (налаштування), і DELETE (опційно) - для видалення записів. Ці запити формуються у React через axios, з використанням заголовків і тіло-запитів, які точно відповідають очікуванням бекенду. На стороні сервера Express.js маршрути організовані за каталогами routes, controllers, models, де кожен запит обробляється конкретним контролером, який у свою чергу взаємодіє з базою через підключений драйвер mysql2. Фрагмент коду, що відповідає за підключення до бази даних представлений на рисунку 2.5.

Отже, взаємодія між frontend і backend у рамках розробленого застосунку є прикладом ефективної, надійної та розширюваної архітектури, де кожен компонент системи виконує свою функцію, а комунікація між рівнями забезпечується через стандартизовані REST-запити з авторизацією, обробкою даних і поверненням результатів у форматі, готовому до відображення. Такий підхід не лише полегшує підтримку й розвиток системи, а й формує позитивний досвід користування за рахунок швидкої реакції інтерфейсу, захищеності даних і зручного контролю над усіма аспектами моніторингу життєдіяльності. Код серверної та клієнтської частини Web-додатку наданий в Додатках А та Б.

```
require('dotenv').config();
const mysql = require('mysql2');

// Створюємо з'єднання з базою даних
const connection = mysql.createConnection({
  host: process.env.DB_HOST || 'localhost',
  user: process.env.DB_USER || 'your_mysql_username',
  password: process.env.DB_PASSWORD || 'your_mysql_password',
  database: process.env.DB_NAME || 'monitoring_db'
});

// Підключення до MySQL
connection.connect((err) => {
  if (err) {
    console.error('Помилка підключення до MySQL:', err);
  } else {
    console.log('Підключення до MySQL встановлено.');
  }
});

module.exports = connection;
```

Рис. 2.5. Фрагмент коду, що відповідає за підключення до бази даних

В основі цієї взаємодії - бібліотека React, Express.js, MySQL, jwt, axios, і правильна структура маршрутизації, що дозволяє реалізувати повноцінну, масштабовану систему з чітким поділом відповідальностей і відкритістю до подальших доповнень.

Висновки до розділу 2

Проєктування та розробка Web-застосунку для моніторингу та аналізу продуктивності життєдіяльності базується на чіткій структурі з урахуванням взаємозв’язку між даними, технологічною архітектурою та сценаріями користувацької взаємодії. Сформована логічна модель даних, що включає таблиці users, data_entries та user_settings, забезпечує структуроване зберігання та гнучке оброблення інформації з можливістю розширення. Реалізація сховища із застосуванням MySQL, підключеного через mysql2 та конфігурованого за

допомогою .env, гарантує стабільний доступ до даних і захист конфіденційної інформації. Застосування пулу з’єднань і підготовлених запитів посилює ефективність та безпеку збереження введених показників.

Клієнтська частина, побудована на React, підтримує SPA-архітектуру, забезпечуючи безперервну навігацію, оперативне відображення аналітики й зручне керування станом інтерфейсу. Інтеграція бібліотек axios, react-router-dom, Material UI та react-chartjs-2 дала змогу реалізувати сучасний, адаптивний і візуально зрозумілий інтерфейс користувача. Серверна частина, реалізована на Express.js, чітко поділена на маршрути, контролери та моделі, що сприяє підтримуваності коду й чіткості взаємодії з базою даних. Система авторизації через JWT та обробка запитів у форматі JSON забезпечують захищений обмін даними між frontend і backend.

Злагоджена взаємодія клієнтської й серверної частини, впровадження REST API для збирання, агрегації та повернення даних, а також врахування персональних налаштувань кожного користувача створюють фундамент для ефективної цифрової системи самоспостереження. Реалізоване рішення не лише відповідає вимогам сучасного користувацького досвіду, а й демонструє технічну готовність до масштабування, доповнення новими функціональними модулями та подальшого розвитку з використанням хмарних інфраструктур або сторонніх сервісів.

РОЗДІЛ 3
ФУНКЦІОНУВАННЯ ТА ЕЛЕМЕНТИ ІНТЕРФЕЙСУ WEB-ЗАСТОСУНКУ

3.1 Огляд реалізованого функціоналу

Основні можливості користувача в реалізованому Web-застосунку для моніторингу та аналізу продуктивності життєдіяльності зосереджені навколо персоналізованої взаємодії з інтерфейсом, обробки щоденних показників життєдіяльності, формування індивідуального дашборду, а також гнучких налаштувань системи під власні потреби. Початковим етапом взаємодії користувача з платформою є проходження процесу автентифікації, що реалізовано через форми реєстрації та входу у відповідних компонентах React (Register.js, Login.js). Загальний вигляд форми авторизації представлений на рисунку 3.1.

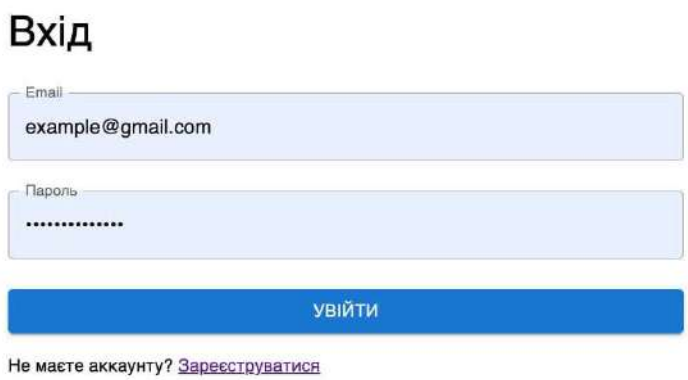

Рис. 3.1. Форма авторизації розробленого Web-застосунку

Користувач має змогу створити обліковий запис, ввівши адресу електронної пошти та пароль, які передаються на сервер через POST-запит на маршрут /api/auth/register. При реєстрації пароль хешується з використанням бібліотеки bcrypt, а при подальшому вході автентифікація відбувається через перевірку даних на сервері, після чого користувач отримує токен у форматі JWT,

що зберігається в localStorage та використовується для доступу до всіх захищених маршрутів. Завдяки цьому, система дозволяє забезпечити індивідуальне середовище для кожного користувача з ізольованими даними.

Після входу в систему користувач потрапляє на дашборд, що реалізовано в компоненті Dashboard.js, де він може переглядати узагальнену статистику власної життєдіяльності у формі ключових показників і графіків. Дашборд відображає такі агреговані дані, як середня тривалість сну за обраний період, сумарна кількість кроків, загальна кількість спожитих калорій і графік змін настрою. Для цього при завантаженні сторінки надсилається GET-запит на маршрут /api/dashboard/:userId, після чого отримані з backend-частини дані (у форматі JSON) передаються до компонентів графіків, реалізованих за допомогою бібліотеки react-chartjs-2. Користувач бачить інформацію у вигляді лінійних діаграм, гістограм або блоків з числовими індикаторами. При цьому оброблені дані фільтруються відповідно до його налаштувань, що зберігаються у таблиці user_settings. Таким чином, одна з основних можливостей користувача полягає в перегляді персоніфікованої аналітики, що формується на основі його історичних записів, внесених у попередні дні. Загальний вигляд створеного користувацького дашборду представлений на рисунку 3.2.

Важливим функціональним елементом системи є можливість введення даних про життєдіяльність, що реалізовано у компоненті DataEntry.js. Тут користувач має змогу вручну зафіксувати основні показники свого дня: кількість годин сну, кількість зроблених кроків, спожиті калорії, настрій та дату, за яку ці дані вводяться. Інтерфейс форми побудовано за допомогою бібліотеки Material UI, що дозволяє легко вводити числові значення, вибирати дату за допомогою візуального календаря та надсилати всі дані на сервер через POST-запит на маршрут /api/data/. Система підтримує перевірку правильності введення значень: наприклад, тривалість сну не може перевищувати 24 години, а настрій повинен бути в межах адекватної шкали. У разі успішного збереження сервер повертає підтвердження, і на інтерфейсі відображається повідомлення про успішне

введення. Ця можливість дозволяє користувачу фіксувати життєві показники регулярно, що створює основу для побудови об'єктивної статистики на дашборді.

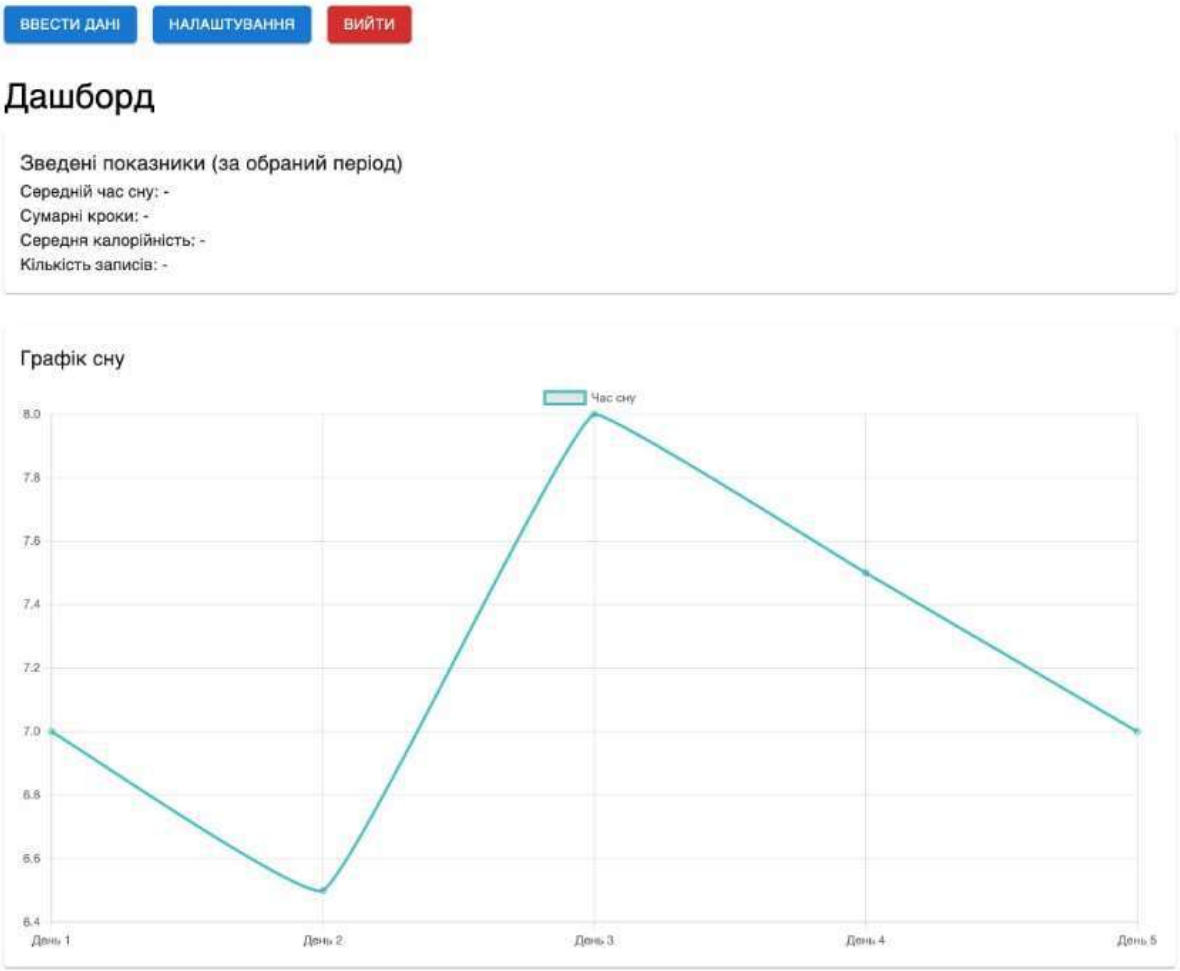

Рис. 3.2. Користувацький дашборд розробленого Web-застосунку

Окрім збору та перегляду інформації, користувач має можливість персоналізувати інтерфейс системи відповідно до власних потреб. Для цього реалізовано компонент Settings.js, у якому можна обрати, які показники відображати на дашборді: сон, настрій, кроки, калорії або лише частину з них. Зміна цих налаштувань надсилається через POST-запит на маршрут /api/settings/:userId, після чого сервер зберігає нові параметри у таблиці user_settings. При наступному вході в систему або оновленні сторінки дашборд автоматично підлаштовується до збережених налаштувань користувача, відображаючи лише ті показники, які він вважає найбільш релевантними. Це

забезпечує високий рівень гнучкості системи та дозволяє адаптувати її під різні потреби: наприклад, користувач, що фокусується на контролі сну, може приховати калорії чи настрій.

Ще однією важливою можливістю є захищений вихід із системи. На дашборді передбачено кнопку «Вийти», натискання якої очищує локальне сховище браузера від збереженого токена автентифікації й перенаправляє користувача на сторінку логіну. Це дозволяє завершити сесію без ризику несанкціонованого доступу, особливо якщо застосунок використовується на спільному комп'ютері чи мобільному пристрої. Також реалізовано перевірку авторизації: якщо при завантаженні сторінки токен відсутній або недійсний, користувач автоматично перенаправляється на сторінку входу, що гарантує цілісність та конфіденційність його персональних даних.

Кожен новий запис передається з клієнтської частини через форму на сторінці DataEntry.js, де користувач вручну вказує значення для кожного показника. Зібрані дані передаються у форматі JSON через axios.post на маршрут /api/data/. На сервері запит перехоплюється відповідним маршрутом у dataRoutes.js, після чого передається до контролера dataController.js, де відбувається перевірка наявності й коректності переданих полів, валідація типів, обробка токена й формування запиту на вставку в таблицю. У разі, якщо для вказаної дати вже існує запис користувача, система може або оновити його (при реалізації додаткової логіки), або відхилити спробу повторного запису. Це виключає дублювання даних і дозволяє забезпечити чистоту інформації в базі. На рівні структури бази data_entries має первинний ключ id, зовнішній ключ user_id, а також поля date, sleep_hours, steps, calories, mood, які відповідають формі введення. Такий підхід дає змогу легко здійснювати фільтрацію, агрегацію й сортування записів відповідно до запитів аналітики.

З технічної точки зору збереження реалізується через використання підготовлених запитів (prepared statements), які дозволяють безпечно підставити значення у SQL-запит без ризику SQL-ін'єкцій. Крім того, всі дані перед надсиланням перевіряються на клієнтському рівні: значення не можуть бути

порожніми, нечислові значення - вводитися в числові поля, настрій - перевищувати умовні межі. Це дозволяє ще до моменту надсилання на сервер відсіяти потенційно некоректні введення. На рисунку 3.3 представлена форма для введення поточних даних користувачем.

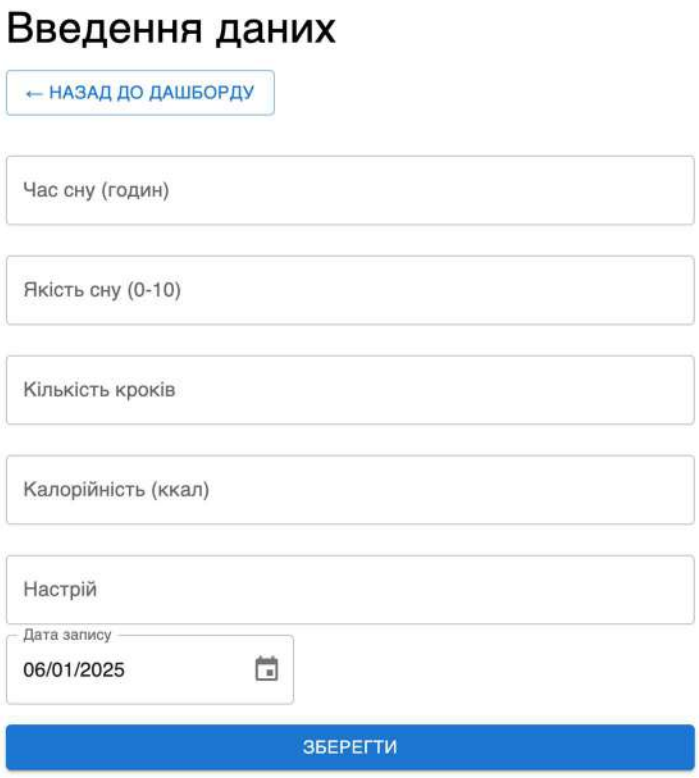

Рис. 3.3. Форма для введення поточних даних користувачем

Для адміністративної обробки даних у межах кожного облікового запису користувач може здійснювати доступ до накопиченої статистики, яка формує аналітичний дашборд. Взаємодія з цими даними реалізується за допомогою контролера dashboardController.js, який виконує запити до таблиці data_entries з використанням агрегатних функцій SQL - AVG, SUM, COUNT, GROUP BY, ORDER BY. У такий спосіб формуються масиви об’єктів із щоденними або середньотижневими показниками, які потім через JSON передаються на frontend у вигляді, зручному для побудови графіків. У клієнтській частині ці дані опрацьовуються у компоненті Dashboard.js, де користувач може бачити зміни своєї активності, рівня енергії чи настрою за певний період. Хоча на цьому етапі

користувач не має можливості редагувати або видаляти записи вручну через інтерфейс, реалізація такої функції можлива завдяки збереженій структурі - кожен запис має унікальний id, тож логіка пошуку, оновлення (PUT) або видалення (DELETE) може бути впроваджена без кардинальної зміни архітектури.

Адміністрування даних включає також аспект персоналізації інтерфейсу. Для цього в базі реалізовано окрему таблицю user_settings, у якій зберігаються налаштування відображення: які показники користувач хоче бачити на дашборді. При кожному вході в систему або оновленні сторінки на frontend відправляється запит на маршрут /api/settings/:userId, який повертає об'єкт налаштувань: наприклад, { show_sleep: true, show_steps: false, show_calories: true, show_mood: true }. Ці значення впливають на логіку рендерингу графіків у Dashboard.js, що дозволяє зробити інтерфейс адаптивним до побажань користувача. Зміна цих налаштувань здійснюється через компонент Settings.js, де за допомогою чекбоксів користувач може активувати або деактивувати певні показники, після чого axios.post надсилає нову конфігурацію на сервер, і вона записується в базу.

Уся система адміністрування даних функціонує в межах авторизованого простору. Перед виконанням будь-якої операції, пов'язаної із записами, сервер перевіряє наявність і валідність токена JWT, який зберігається у браузері користувача. У middleware на backend, реалізованому через express-jwt або кастомну перевірку, відбувається декодування токена, отримання user_id, а також перевірка його відповідності користувачу, для якого здійснюється запит. Якщо токен відсутній або недійсний - система повертає статус 401 і припиняє обробку. Таким чином, адміністрування даних є не лише інтерфейсною дією, але й безпечною транзакцією з авторизаційною перевіркою. У подальшому система може бути доповнена модулем історії змін, журналюванням операцій або переглядами з боку адміністратора, якщо передбачена багаторівнева система прав доступу.

Генерація аналітики та формування дашборду в розробленому web-застосунку виконується на основі структурованого підходу до обробки

накопичених користувацьких даних з метою їх подальшого візуального представлення у зручному форматі. Дашборд є головним інтерфейсом, який користувач бачить після авторизації, і він виконує функцію центрального вузла прийняття рішень та самоспостереження. Його основна задача - показувати узагальнені значення показників життєдіяльності користувача, виявляти тенденції, динаміку змін і формувати уявлення про загальний стан активності, харчування, сну та настрою. На відміну від інших сторінок, дашборд не передбачає ручного введення даних - він побудований виняткова на механізмах аналітичної обробки інформації, яка вже є в системі.

Алгоритм формування дашборду розпочинається з отримання даних про користувача з бази через відповідний API-запит, який надсилається одразу після завантаження компонента Dashboard.js. Зі сторони frontend-а, запит виконується через бібліотеку axios, яка надсилає HTTP GET-запит на маршрут /api/dashboard/:userId, доповнений авторизаційним токеном у заголовку. Після того, як сервер отримує запит, у контролері dashboardController.js відбувається виклик кількох SQL-запитів до таблиці data_entries, які обробляються з урахуванням ідентифікатора користувача. Зібрані дані агрегуються на рівні бази: для кожного з показників (сон, кроки, калорії, настрій) визначаються середні або сумарні значення, а також часові ряди для побудови графіків. Після обробки отримані значення повертаються на клієнтську частину у форматі JSON, де вони передаються до візуальних компонентів для побудови графічних і числових індикаторів.

Користувач бачить дашборд як набір логічно розташованих блоків, кожен з яких відповідає за окремий аспект аналізу даних. Наприклад, один блок може виводити поточну середню тривалість сну за останні 7 днів, інший - сумарну кількість кроків за поточний тиждень, ще один - графік змін настрою за обраний діапазон часу. Візуалізація реалізована через бібліотеку react-chartjs-2, що дозволяє відображати дані у вигляді лінійних графіків, стовпчикових діаграм та індикаторів з круговим відображенням (наприклад, прогрес бар). Графіки не є статичними: вони формуються динамічно на основі тих даних, які повертає

сервер, і оновлюються кожного разу при повторному зверненні до сторінки або зміні параметрів. Важливо, що на клієнтському рівні дані не обробляються математично - усі розрахунки відбуваються виключно на сервері, що дозволяє уникнути надлишкового навантаження на браузер і підтримувати високу продуктивність інтерфейсу.

Крім стандартних графіків, на дашборді передбачено вивід індикаторів, які сигналізують про досягнення або недотримання заданих норм. Наприклад, якщо кількість кроків за день була меншою за 5000, відповідний індикатор може підсвічуватись жовтим або червоним кольором, тоді як нормальні значення підсвічуються зеленим. Така система кольорових маркерів дає змогу швидко оцінити ситуацію без потреби вчитуватись у числові значення. Додатково, якщо у базі не зафіксовано жодного запису за певну дату або період, відповідний блок дашборду відображає повідомлення «Дані відсутні», що спонукає користувача внести інформацію. Це важливо з точки зору підтримки регулярного користування системою та формування звички моніторингу.

Формування дашборду також враховує індивідуальні налаштування, які зберігаються у таблиці user_settings. При вході в систему дашборд відображає лише ті показники, які активовані користувачем. Така поведінка забезпечує інтерфейсну гнучкість і дозволяє кожному користувачу адаптувати аналітичну панель під власні пріоритети.

Інтерактивність дашборду забезпечена також можливістю фільтрації періоду: користувач може вибрати перегляд даних за день, тиждень або місяць. Це реалізовано через інтерактивні елементи інтерфейсу (наприклад, кнопки або випадаючі списки), які змінюють параметри запиту до серверного API. Коли користувач обирає певний діапазон, frontend формує запит із додатковими параметрами startDate і endDate, які сервер обробляє і формує SQL-запити з фільтрацією по датах. Така реалізація дає змогу не лише бачити поточний стан, а й здійснювати ретроспективний аналіз - наприклад, порівняти активність за тиждень до і після певної події або виявити зміни в настрої на тлі зміни фізичної активності.

Дашборд також виконує функцію стартової точки для подальшої навігації. Користувач має змогу перейти на сторінку DataEntry для внесення нових даних або на сторінку Settings для коригування відображуваних показників. Усі переходи відбуваються без перезавантаження сторінки завдяки використанню маршрутизації react-router-dom, що зберігає швидкість і зручність взаємодії. У разі відсутності авторизаційного токена або його завершення, дашборд автоматично перенаправляє користувача на сторінку логіну, що реалізовано через перевірку наявності токена у localStorage при завантаженні компонента.

Отже, дашборд у реалізованому застосунку виконує функцію не лише відображення даних, а й аналітичної інтерпретації інформації, що зберігається в системі. Він поєднує агреговані результати SQL-запитів, динамічну побудову графіків, індивідуальні налаштування й механізми фільтрації для створення адаптивного середовища, яке відповідає потребам користувача. Завдяки використанню перевірених технологій - Express, React, Chart.js, axios - та правильно реалізованій логіці взаємодії між клієнтом і сервером, дашборд перетворюється на повноцінний інструмент самоспостереження, який сприяє підвищенню усвідомленості користувача щодо власного стилю життя та його динаміки.

3.2 Сценарії взаємодії користувача з розробленим Web-інтерфейсом

Структура клієнтського інтерфейсу в розробленому web-застосунку реалізована на базі компонентної архітектури React, яка забезпечує гнучкість побудови, повторне використання елементів, логічне розділення логіки інтерфейсу та високу продуктивність. Головною концепцією при створенні інтерфейсу було побудувати зрозумілу, інтуїтивну й адаптивну систему, яка не потребує додаткових пояснень для користувача, дозволяє швидко орієнтуватися у функціоналі, вводити дані, переглядати аналітику й керувати налаштуваннями. Вся структура клієнтської частини побудована в межах директорії src, яка містить основний контейнер App.js, точку входу index.js, а також підкаталоги для

сторінок (pages), компонентів (components), стилів (styles) та службових функцій (utils або services). Така організація дозволяє уникнути надмірної складності й забезпечує логічне розміщення кожного функціонального блоку інтерфейсу.

Після запуску застосунку у вікні браузера завантажується головний компонент App, який відповідає за маршрутизацію між сторінками за допомогою бібліотеки react-router-dom. У ньому описано всі маршрути, зокрема: /login, /register, /dashboard, /data-entry, /settings, і для кожного з них вказано компонент, який буде відображатися. Це дозволяє реалізувати логіку переходів без перезавантаження сторінки (SPA-підхід), що покращує швидкодію та користувацький досвід. Структурно інтерфейс поділений на кілька основних екранів: сторінка входу (Login.js), сторінка реєстрації (Register.js), головна сторінка-аналітика (Dashboard.js), форма введення показників (DataEntry.js) та сторінка налаштувань (Settings.js). Кожна з них реалізована як окремий компонент з унікальними функціональними та візуальними елементами, які відповідають за відповідну дію в межах системи.

В основі інтерфейсного оформлення використовується бібліотека Material UI (@mui/material), що дозволяє будувати сучасні, адаптивні, візуально привабливі компоненти з високою швидкістю розробки. Наприклад, на всіх сторінках використовуються такі MUI-компоненти, як TextField, Button, Container, Grid, Checkbox, Typography, які легко кастомізуються під потреби дизайну. Окрім цього, в окремих модулях використовуються стилі, написані за допомогою бібліотеки Emotion (@emotion/react), що дозволяє створювати стилізовані компоненти та медіа-запити безпосередньо в коді компонентів. Такий підхід забезпечує узгоджений вигляд усіх елементів інтерфейсу й робить адаптацію під різні розміри екранів логічною та швидкою. Усі сторінки повністю підтримують мобільну адаптивність, зокрема за допомогою Flexbox або Grid-структур, що дозволяє користуватися застосунком з різних пристроїв.

Інтерфейс починається зі сторінки авторизації. Компонент Login.js має структуру з полями для введення email і пароля, кнопкою підтвердження та коротким повідомленням про результат дії. Усі поля прив’язані до локального

стану за допомогою useState, що дозволяє в реальному часі перевіряти введення, виводити підказки, повідомляти про помилки або успіх. Структура компонента дозволяє миттєво після успішного входу зберегти токен у localStorage і перенаправити користувача на сторінку дашборду. Аналогічна структура реалізована для Register.js, однак тут додатково передбачено підтвердження пароля, а також обробку сценарію, коли користувач уже існує. Кожен із цих компонентів має логічну структуру - заголовок, форму, повідомлення й посилання для переходу - що відповідає загальноприйнятим стандартам UX-дизайну.

Головна інформаційна частина інтерфейсу розміщена в Dashboard.js. Тут структура включає шапку сторінки, панель навігації (або іконки), блоки з індикаторами, а також окремі секції з графіками, побудованими на основі бібліотеки react-chartjs-2. Компонент використовує useEffect для завантаження даних після відкриття сторінки, а також useState для зберігання й обробки отриманих з API масивів. Кожен блок - наприклад, сон або калорії - відображається у вигляді графіка, а якщо дані відсутні, компонент виводить повідомлення типу “Немає записів за обраний період”. Структура дашборду передбачає також фільтри - наприклад, перемикачі для вибору періоду (день, тиждень, місяць), які змінюють параметри запиту до серверної частини. Таким чином, інтерфейс не лише відображає статистику, а й дає змогу змінювати спосіб її представлення.

На сторінці введення даних (DataEntry.js) інтерфейс організовано як форма з полями для введення показників: sleep_hours, steps, calories, mood, а також вибору дати. Усі поля представлені за допомогою MUI-компонентів, які мають вбудовану валідацію, що запобігає введенню некоректних значень. Після натискання кнопки «Зберегти» дані надсилаються через axios.post, а результат дії відображається у вигляді короткого повідомлення під формою. Візуально компонент витримано в єдиному стилі з іншими сторінками, що забезпечує візуальну узгодженість. Стан форми скидається після успішного збереження, що дозволяє знову ввести нові дані без перезавантаження сторінки. Навігація між

сторінками підтримується через використання посилань типу <Link>або useNavigate, що дозволяє переміщатися між формою введення, дашбордом і налаштуваннями без порушення поточного стану застосунку.

Сторінка налаштувань (Settings.js) побудована у вигляді списку опцій, які представлені чекбоксами. Структурно інтерфейс ділиться на заголовок, перелік доступних показників і кнопку «Зберегти». При натисканні на чекбокси відбувається зміна локального стану, а при підтвердженні - надсилається POST-запит на сервер, де оновлюється таблиця user_settings. Отримані налаштування використовуються в Dashboard.js, щоб визначити, які блоки відображати, а які приховати. Компонент також інформує користувача про результат збереження - як успішний, так і помилковий. Така структура дозволяє легко змінювати вигляд дашборду, не втручаючись у структуру даних або бізнес-логіку, що свідчить про правильне розділення відповідальностей між компонентами інтерфейсу.

Сценарії користувацької взаємодії в розробленому Web-застосунку визначають послідовність дій, які здійснює користувач під час роботи із системою для досягнення певної цілі: від початкового доступу до застосунку до аналізу власної продуктивності. У межах реалізованого функціоналу було побудовано кілька ключових сценаріїв, які охоплюють як одноразові дії (наприклад, реєстрація або зміна налаштувань), так і циклічні щоденні процеси (введення даних, перегляд аналітики). Особливістю реалізованої взаємодії є те, що вона відбувається в межах SPA-архітектури, без повного перезавантаження сторінки, що забезпечує швидку реакцію інтерфейсу на дії користувача, зручну навігацію й інтеграцію всіх модулів у єдиний сценарний простір.

Первинний сценарій користування починається зі знайомства з інтерфейсом входу, реалізованого у компоненті Login.js. Користувач вводить email і пароль, після чого надсилає форму, що тригерить POST-запит до /api/auth/login. У разі помилки - наприклад, невірний пароль або відсутній користувач - в інтерфейсі з’являється відповідне повідомлення, і користувач має змогу повторити спробу. Якщо авторизація проходить успішно, користувач отримує токен, який зберігається в localStorage, і виконується автоматичне

перенаправлення на дашборд. Токен додається в заголовки всіх подальших запитів, що є критичним для реалізації інших сценаріїв. У випадку, якщо користувач ще не зареєстрований, він може перейти за посиланням на сторінку Register.js, де вводить ті самі поля. Усі помилки обробляються на фронтенді й відображаються динамічно - наприклад, повідомлення про те, що користувач з таким email уже існує.

Наступний важливий сценарій - регулярне введення нових даних про стан користувача. Після входу на дашборд, користувач має можливість перейти до форми DataEntry.js, де фіксує щоденні показники: сон, кроки, калорії, настрій і дату. Цей сценарій реалізовано як послідовність введення значень у поля форми, після чого натискання кнопки «Зберегти» запускає логіку збереження. Інтерфейс одразу реагує на неправильні типи введення - наприклад, якщо в полі для годин сну введено текст, користувач отримає підказку про помилку ще до відправки. Після успішного запиту, що надсилається на /api/data/, форма скидається, а користувачу надсилається повідомлення про збереження. Користувач може одразу ж ввести інші дані, не оновлюючи сторінку, що дозволяє зберігати послідовність взаємодії. Якщо дані за певну дату вже були введені, система запобігає дублюванню, видаючи відповідне повідомлення.

Одним із найважливіших сценаріїв є взаємодія з дашбордом (Dashboard.js). Після авторизації цей компонент завантажується автоматично і виконує GET-запит до /api/dashboard/:userId, який повертає попередньо агреговані дані. Сценарій передбачає пасивну взаємодію користувача з візуальними елементами - графіками, індикаторами, підсумковими блоками. Водночас користувач має можливість змінити часовий діапазон, наприклад, обрати тиждень замість місяця, натиснувши на відповідний контрол у верхній частині сторінки. Це запускає повторний запит з іншими параметрами до бекенду. На основі відповіді оновлюється інтерфейс, змінюється графік або відображаються повідомлення про відсутність даних. У разі недостатньої активності користувача (наприклад, якщо не введено жодного запису за останні 7 днів), дашборд надає підказки з пропозицією перейти на сторінку введення даних.

Сценарії користувацької взаємодії також включають зміну персональних налаштувань у Settings.js. Користувач може активувати або вимкнути відображення певних блоків у дашборді за допомогою чекбоксів. Ці дії відбуваються локально в інтерфейсі, і після натискання кнопки «Зберегти» зміни надсилаються через POST-запит до /api/settings/:userId. У разі успішного збереження, у верхній частині сторінки з'являється підтвердження, а самі зміни набирають чинності при наступному вході або оновленні дашборду. Такий сценарій дозволяє користувачеві адаптувати застосунок під власні потреби: наприклад, якщо він слідкує лише за сном і настроєм, він може відключити показ кроків або калорій. Усі зміни зберігаються у базі, що забезпечує сталість конфігурації навіть після повторного входу в систему.

Окремий сценарій передбачає завершення сесії - тобто вихід із системи. На будь-якій сторінці (наприклад, дашборді або налаштуваннях) користувач має доступ до кнопки «Вийти», натискання якої очищує localStorageвід збереженого токена і перенаправляє користувача на сторінку логіну. Це реалізовано через виклик функції logout, яка виконує логіку очищення та навігації через useNavigate(). Такий сценарій критично важливий для забезпечення безпеки в разі використання загальнодоступних пристроїв, а також для перезапуску сесії при зміні користувача. Якщо користувач намагається отримати доступ до захищеної сторінки без токена, маршрутизатор перенаправляє його на логін автоматично, що виключає можливість неавторизованого доступу.

Сценарії також враховують дії у разі помилок, втрати з'єднання або непередбачуваних ситуацій. Наприклад, якщо сервер не відповідає, користувач отримає повідомлення про помилку під час виконання запиту, яке з'явиться як окремий компонент. У випадку втрати мережі або відсутності відповіді від сервера, функції зберігання даних або запиту аналітики повернуть відповідні повідомлення - «Помилка з'єднання», «Спробуйте пізніше». Усі повідомлення вбудовано в інтерфейс компонентів через useState, тому вони оновлюються динамічно без перезавантаження.

UX-рішення в реалізованому Web-застосунку були спрямовані на забезпечення максимальної простоти користування, інтуїтивного розуміння структури інтерфейсу та ефективного досягнення користувачем цілей без зайвих дій або когнітивного навантаження. Основою концепції стало застосування мінімалістичного, але функціонального підходу, де кожен елемент має чітке призначення, не перевантажує простір і водночас дозволяє користувачу швидко орієнтуватися в межах системи. Візуальна ієрархія побудована таким чином, що головні дії (реєстрація, вхід, введення даних, перегляд аналітики, налаштування) мають домінуючі акценти, які забезпечують швидке виявлення інтерактивних зон навіть для користувача без досвіду у взаємодії з цифровими інтерфейсами. Головні екрани (Login, Dashboard, DataEntry, Settings) мають чітке позиціонування контенту, логічні блоки та єдиний візуальний стиль, який підтримується завдяки бібліотеці компонентів Material UI.

Одним із ключових UX-рішень стало обмеження кількості одночасно доступних дій на кожному екрані. Наприклад, при вході в систему користувач бачить лише два активні поля та кнопку підтвердження - без відволікаючих елементів або зайвого тексту. Такий підхід дає змогу зосередити увагу на основному завданні - автентифікації. Після входу весь інтерфейс зосереджується навколо головної функції - перегляду власної аналітики. На дашборді дані відображаються у вигляді великих, читабельних блоків, де підписи, іконки та колірна система дають змогу швидко інтерпретувати інформацію. Впровадження колірних маркерів - зелений для нормальних значень, червоний для критичних, жовтий для нейтральних - дозволяє користувачеві за лічені секунди оцінити свій стан без потреби вчитуватись у числові дані.

Окрему увагу приділено формуванню логіки навігації між сторінками. Навігація реалізована через react-router-dom, завдяки чому вся взаємодія з додатком відбувається без перезавантаження сторінок, що знижує час очікування і підвищує відчуття безперервності. Перемикання між дашбордом, введенням даних і налаштуваннями відбувається через доступні посилання або кнопки, які мають стабільне розміщення. Наприклад, у правому верхньому куті завжди

доступна кнопка виходу, а переходи на інші сторінки реалізовано у вигляді зрозумілих кнопок з підписами. Крім цього, застосовано принцип збереження контексту: після успішної дії (наприклад, збереження запису) користувач бачить повідомлення, але залишається на поточній сторінці, що дозволяє не втрачати послідовність взаємодії. Така логіка усуває необхідність повторного пошуку або навігації, покращуючи загальне враження від системи.

UX-підхід до організації форм передбачає зменшення кількості кліків, необхідних для виконання базових дій. Наприклад, у формі DataEntry всі поля згруповані вертикально, що дозволяє зручно переходити між ними за допомогою клавіатури. Компонент вибору дати (DatePicker) має зрозумілий вигляд, адаптований під мобільні екрани. Кнопка збереження завжди знаходиться в межах екрану - без потреби прокручувати сторінку. У разі помилки, користувач одразу бачить відповідне повідомлення під конкретним полем або у верхній частині форми. Колір помилки - червоний, і він супроводжується коротким текстом типу “Будь ласка, введіть кількість кроків у числовому форматі”. Такі дрібні, але продумані деталі забезпечують стабільність, передбачуваність і довіру до системи.

Зручність навігації посилюється також завдяки тому, що всі сторінки мають спільну стилістичну концепцію - однакова палітра кольорів, типографіка, розміри полів і відступи. Завдяки цьому користувач швидко навчається інтерфейсу і вже через кілька взаємодій легко передбачає, де знаходиться та чи інша функція. Крім цього, всі інтерактивні елементи мають візуальний фідбек: кнопки змінюють колір при наведенні, а натискання супроводжується легким анімованим ефектом, що підтверджує дію. Така поведінка робить систему більш «живою» і зрозумілою, особливо для користувачів, які звикли до сучасних застосунків.

Важливе UX-рішення полягає в тому, що система не перевантажує користувача інформацією. Наприклад, на сторінці Dashboard одночасно показується лише актуальна статистика за обраний період. При цьому користувач не бачить великої кількості непотрібної історичної інформації, якщо не вибере її самостійно через зміну параметрів. Це відповідає принципу «показуй лише те,

що потрібно в даний момент». Якщо ж даних немає - відображається просте повідомлення на кшталт «Поки що немає записів за цей період», що дозволяє уникати помилок інтерфейсу й підтримувати позитивний досвід.

UX-рішення були адаптовані і під мобільні пристрої. Компоненти автоматично перебудовуються під ширину екрану, елементи форми масштабуються, навігаційні кнопки розміщуються так, щоб ними було зручно користуватися пальцями. Інтерфейс був протестований на пристроях різного розміру - планшетах і смартфонах - і оптимізований так, щоб ключовий функціонал був доступним незалежно від розміру дисплея. Це робить систему зручною для тих користувачів, які ведуть моніторинг своєї життєдіяльності «на ходу».

Таким чином, застосовані UX-рішення у проєкті не лише забезпечили зручну навігацію й логіку інтерфейсу, але й створили комфортне середовище, в якому користувач орієнтується інтуїтивно, не потребуючи інструкцій або навчання. Кожен крок взаємодії - від входу до аналізу результатів - був спроєктований з урахуванням принципів доступності, логічності та наочності, що суттєво підвищує загальну ефективність використання системи та сприяє формуванню сталого користувацького досвіду. Реалізація на базі React, Material UI та підтримка динамічного оновлення інтерфейсу дозволила реалізувати всі ці рішення у максимально гнучкому та масштабованому вигляді.

3.3 Аналіз ефективності та можливих обмежень системи

Оцінка працездатності та стабільності реалізованого Web-застосунку проводилась на основі цілеспрямованого аналізу взаємодії між клієнтською та серверною частинами, ефективності збереження й обробки даних, коректності відображення аналітичних компонентів, а також стійкості до типових помилок і збоїв у роботі. Система демонструє високу стабільність під час щоденного використання, коли здійснюється регулярне введення нових показників життєдіяльності, зчитування статистики з бази даних, перемикання між

сторінками та налаштування вигляду інтерфейсу. Усі запити, що надходять із клієнтської частини (React), обробляються на сервері (Node.js + Express) асинхронно з відповідною перевіркою валідності, що дозволяє підтримувати плавність і безпомилкову логіку відгуку на дії користувача. Навіть у разі тимчасової затримки у відповіді, інтерфейс передбачає механізми обробки станів - наприклад, показ індикатора завантаження або текстове повідомлення про помилку, що забезпечує прозорість процесу.

Важливим чинником стабільності є безпомилкове функціонування маршрутизованих запитів. У межах реалізованого API, кожен маршрут (/api/auth, /api/data, /api/dashboard, /api/settings) обробляється окремими контролерами, логіка яких була рознесена в окремі модулі для зменшення складності обробки. Завдяки використанню middleware для перевірки авторизаційного токена та перевірки полів запиту, система здатна самостійно блокувати некоректні або потенційно небезпечні дії. Наприклад, у випадку, коли користувач надсилає POST-запит без одного з обов'язкових параметрів (наприклад, без date або sleep_hours), сервер не лише не обробляє запит, а й повертає структуровану помилку у форматі JSON. Це дозволяє клієнтському інтерфейсу миттєво зреагувати на помилку, не порушуючи загальної стабільності роботи.

З технічної точки зору система демонструє працездатність навіть у випадках багаторазових викликів одних і тих самих запитів за короткий проміжок часу. Наприклад, при повторному надсиланні форми введення даних (коли користувач натискає кнопку декілька разів), backend виявляє дублювання дати для поточного користувача й відхиляє спробу дублювати запис, залишаючи базу даних у стабільному стані. Цей механізм реалізовано на рівні контролера dataController.js, де перед виконанням вставки в таблицю data_entries відбувається перевірка наявності запису з такою самою датою й user_id. Така логіка не лише унеможливлює надмірне заповнення бази, а й свідчить про продуману модель захисту від неуважності або помилок користувача.

Під час тестування працездатності також враховувалась здатність системи реагувати на проблеми з доступом до сервера або бази даних. Усі запити через axios мають вбудовану обробку .catch(), яка дозволяє відловлювати будь-які критичні помилки, пов’язані з відсутністю відповіді або внутрішніми помилками сервера. Наприклад, якщо база даних тимчасово недоступна (через перезавантаження або втрату з’єднання), система повертає статус 500, і інтерфейс виводить повідомлення «Неможливо зберегти дані. Спробуйте пізніше». Це дозволяє уникнути фрустрації з боку користувача, оскільки він одразу отримує зворотний зв’язок щодо проблеми, не залишаючись у стані невизначеності.

Працездатність і стабільність застосунку також підтверджується його здатністю підтримувати зв’язок з базою MySQL навіть у випадку серії запитів з різними параметрами фільтрації. Наприклад, при зміні періоду перегляду на дашборді (Dashboard.js) система виконує новий GET-запит із оновленими параметрами до /api/dashboard/:userId, і кожного разу результат повертається без втрат у продуктивності. Це стало можливим завдяки використанню пулу з’єднань у драйвері mysql2, що дозволяє багаторазово використовувати відкриті підключення без необхідності кожного разу створювати нові. Така архітектура оптимізує роботу з базою і знижує ризик зависань або втрат з’єднання під навантаженням.

Усі компоненти інтерфейсу демонструють стабільну роботу незалежно від пристрою - у процесі тестування додаток був запущений на десктопах, ноутбуках і смартфонах з різними розмірами екранів. Усі основні функції - введення даних, навігація між сторінками, перегляд аналітики - залишаються доступними, елементи не зсуваються, графіки коректно масштабується, а адаптивні стилі забезпечують збереження зручності навіть при мінімальній ширині екрана. Така стабільність стала можливою завдяки використанню Material UI, де всі компоненти розраховані на адаптивну поведінку, та продуманому CSS через Emotion, де були задані межі, відступи й логіка відображення для різних breakpoint-ів.

Під час перевірки працездатності також оцінювалася логіка збереження користувацьких налаштувань. Система стабільно зчитує з таблиці user_settings інформацію про обрані показники (сон, кроки, калорії, настрій) і формує дашборд лише з тих блоків, які активовані користувачем. У разі зміни налаштувань через Settings.js, після оновлення дашборду зміни зберігаються, і не відбувається помилок відтворення. Це свідчить про правильну синхронізацію між frontend-станом і даними на сервері, що є ознакою надійної взаємодії між компонентами системи. Навіть якщо користувач вносить зміни кілька разів поспіль або залишає сторінку до завершення збереження, додаток не зависає і повертає зворотній зв'язок про статус операції.

Попри ефективну реалізацію основних компонентів web-застосунку для моніторингу та аналізу продуктивності життєдіяльності, система має низку потенційних обмежень, які пов'язані як із технічними аспектами обробки даних, так і з архітектурними рішеннями, закладеними на етапі розробки. Одне з головних обмежень полягає у відсутності механізму редагування або видалення раніше введених записів. У поточній реалізації, після надсилання даних через DataEntry.js, запис фіксується у базі data_entries, і змінити його через інтерфейс неможливо. Це створює труднощі у випадку, якщо користувач помилково ввів неправильне значення або обрав хибну дату. Виправлення таких ситуацій можливе лише через втручання в базу або додавання додаткового функціоналу, якого наразі не передбачено. Відсутність CRUD-повного управління записами є обмеженням з точки зору гнучкості користувача у веденні власної статистики.

Ще одним суттєвим обмеженням є те, що система не підтримує багатокористувацькі сценарії в одному акаунті. Кожен обліковий запис призначений лише для однієї особи, а тому використання застосунку сім'єю або групою осіб неможливе без створення окремих акаунтів. Такий підхід унеможливлює порівняння статистики, спільний аналіз або ведення динаміки групового рівня. Це особливо помітно в контексті використання застосунку в освітніх, тренувальних або корпоративних середовищах, де була б доречною

функція адміністрування кількох користувачів із централізованим контролем. У такому разі потрібне суттєве розширення функціоналу через введення ролей, прив’язки акаунтів до групи й реалізація фільтрації даних за категоріями.

Інтерфейсна логіка, хоча й побудована за принципом SPA, має обмеження у глибокому налаштуванні. Наприклад, користувач не має можливості створити власні категорії або ввести додаткові параметри самостійно. Список фіксованих показників - сон, кроки, калорії, настрій - є закритим, і додати новий тип даних (наприклад, рівень води або години фокусування) можна лише шляхом втручання в код і структуру бази даних. Таким чином, розширення системи можливе лише силами розробника, що обмежує користувача у кастомізації під власні потреби. Це також впливає на універсальність платформи, яка наразі орієнтована на базовий набір показників, без підтримки модульного налаштування.

З точки зору аналітики, функціонал побудований довкола агрегованих значень, які генеруються на сервері, однак не включає просунутих механізмів обробки - наприклад, аналізу відхилень, прогнозування, виявлення аномалій або порівняльних трендів між періодами. Уся логіка обмежена середніми значеннями, сумарними підрахунками і візуалізацією на рівні базових графіків. Для більш досвідчених користувачів це може бути недостатньо інформативно, оскільки не враховується динаміка змін, сезонність чи вплив факторів. Крім того, у реалізації відсутня можливість експорту даних для подальшої обробки в інших програмах - користувач не може завантажити свою статистику у форматі CSV або PDF, що обмежує використання системи як інструменту для звітності або стороннього аналізу.

Іще одним обмеженням є відсутність повідомлень або нагадувань про необхідність внесення даних. Поточна система повністю пасивна: лише користувач ініціює дії, а сама платформа не містить жодного елемента взаємодії з користувачем за межами інтерфейсу. Наприклад, якщо користувач забув зафіксувати дані за день - жодне повідомлення не нагадає про це. Відсутність push-нотифікацій, email-інформування чи календарного планування знижує регулярність використання й перетворює систему з інтерактивної в просто

інструмент збереження. Для переходу на новий рівень взаємодії необхідно реалізувати системи сповіщень, трекер регулярності, історію пропущених днів та динамічні поради.

Функціонально також обмежено виведення історичних даних: користувач бачить лише агреговані графіки, але не має доступу до повного списку введених записів. Наприклад, він не може переглянути таблицю зі всіма своїми даними за місяць або рік у табличному вигляді, де кожен день представлений окремим рядком. Це створює складнощі у випадках, коли потрібно вручну перевірити хронологію або знайти конкретну дату. Відсутність такого переліку також унеможливлює швидке виявлення помилок у введенні або просто зручного перегляду минулих результатів. Реалізація історії записів із можливістю фільтрації, сортування й пошуку значно підвищила б прозорість і зручність системи.

Серверна частина застосунку побудована без обліку масштабного навантаження або підтримки одночасного великого потоку користувачів. Поточна структура із використанням Node.js + Express та MySQL не оптимізована для масштабування у хмарі або горизонтального розширення інфраструктури. У разі різкого зростання кількості запитів або запровадження багатокористувацьких сценаріїв можуть виникнути обмеження у продуктивності. Крім того, не реалізовано систему кешування відповідей для аналітичних запитів, що при великій кількості користувачів призведе до підвищеного навантаження на базу даних. Ці технічні аспекти наразі не створюють проблем, але є потенційними обмеженнями при масштабуванні.

Загальний інтерфейс, хоча й адаптований під мобільні пристрої, не підтримує жестову навігацію, темну тему або можливість персоналізувати інтерфейс візуально. Всі користувачі бачать однаковий вигляд застосунку без можливості змінити кольорову палітру, розміщення блоків чи розмір шрифтів. Це може створювати труднощі для людей із вадами зору або індивідуальними потребами у візуальному комфорті. Відсутність елементів доступності (accessibility features), таких як озвучення, масштабування шрифтів,

контрастність, зменшує інклюзивність застосунку, що вважається важливим аспектом у сучасному UX-дизайні.

Перспективи розвитку та масштабування розробленого web-застосунку для моніторингу та аналізу продуктивності життєдіяльності охоплюють широкий спектр напрямів, які передбачають як розширення функціональних можливостей системи, так і її адаптацію до масштабного використання з більшим навантаженням, інтеграцією зовнішніх сервісів і персоналізацією користувацького досвіду. Одним із першочергових кроків у цьому напрямі є впровадження повного CRUD-функціоналу для введених користувачем записів. На поточному етапі додаток підтримує лише створення нових записів, тоді як реалізація функцій редагування та видалення дозволить користувачу більш гнучко керувати власною інформацією, виправляти помилки, оновлювати старі записи та очищати дані у випадку потреби. Для цього необхідно доповнити серверну логіку новими маршрутами PUT та DELETE, а клієнтську частину - формою редагування та списком усіх наявних записів із відповідними кнопками керування.

Другим напрямом розвитку є поглиблення аналітичного інструментарію. Поточна реалізація обмежується базовою візуалізацією статистики - такими як середні значення чи суми за період, що є достатнім для загального уявлення про тенденції. Однак для покращення інформативності доцільним є впровадження розширеної обробки: аналіз відхилень, побудова трендових ліній, прогнозування показників на основі попередніх даних, а також виявлення аномалій (наприклад, різке зниження активності чи погіршення настрою). Це дозволить користувачу отримати глибші інсайти з власної інформації, а також краще розуміти взаємозв’язки між поведінкою та станом здоров’я. Така аналітика потребуватиме інтеграції з бібліотеками обробки статистичних даних, наприклад, Chart.js з аналітичними плагінами або сторонніх сервісів на базі Python-скриптів.

Окремим перспективним напрямом є реалізація багатокористувацьких сценаріїв. Це означає введення можливості створення групових акаунтів, у межах яких кілька користувачів можуть ділитися даними, порівнювати прогрес або

керувати аналітикою в межах сім’ї, навчального колективу чи тренувальної групи. Такий функціонал потребує створення ієрархії ролей, наприклад: адміністратор, учасник, гість. Для цього необхідно доповнити модель usersдодатковими полями, реалізувати систему прав доступу та інтерфейс групового перегляду статистики. Це значно розширить аудиторію застосунку та дозволить використовувати його в освітніх, медичних або спортивних установах, де персоналізоване моніторування важливо поєднувати з загальною динамікою групи.

У межах масштабування також доцільно впровадити систему обробки повідомлень та нагадувань. Зокрема, користувачі мають отримувати сповіщення про необхідність заповнення даних, якщо певний день залишився без запису. Це може бути реалізовано через email-сповіщення або push-нотифікації, якщо додаток буде адаптовано до мобільної версії. Така система дозволить підтримувати регулярність використання, формувати звичку до самоспостереження та підвищити залученість. Окрім того, можна реалізувати рекомендаційний модуль, який на основі даних попередніх періодів пропонуватиме користувачеві поради - наприклад, збільшити фізичну активність або нормалізувати режим сну. Це потребуватиме впровадження простих алгоритмів машинного навчання та гнучкого шаблонізатора повідомлень.

Ще однією вагомою перспективою розвитку є розширення бази підтримуваних показників. Наразі система зосереджена на чотирьох основних параметрах: сон, кроки, калорії, настрій. Проте користувачі можуть мати потребу у фіксації додаткових змінних - наприклад, рівень гідратації, кількість випитої кави, години концентрації, рівень стресу або якість харчування. Для цього необхідно реалізувати механізм динамічного створення полів на вимогу користувача. Така гнучкість дозволить кожному адаптувати систему під власні звички, цілі або медичні рекомендації, що розширить функціональну привабливість застосунку. При цьому слід реалізувати модуль шаблонів, який дозволятиме зберігати й повторно використовувати певні комбінації полів, щоб уникнути повторного налаштування при кожному вході.

З технічної точки зору перспективним кроком є адаптація архітектури до масштабного навантаження. Поточна реалізація базується на локальному розгортанні Node.js + Express із підключенням до MySQL, що є придатним для обслуговування обмеженої кількості користувачів. Для подальшого масштабування варто впровадити розміщення в хмарній інфраструктурі, зокрема AWS або Firebase, із використанням балансування навантаження, кешування відповідей (наприклад, через Redis), асинхронної обробки фонового завантаження (через queue-системи) та резервного збереження даних. У контексті розгортання слід також передбачити CI/CD-процеси, що дозволить регулярно оновлювати застосунок, тестувати оновлення та гарантувати стабільність без збоїв у продакшені.

Персоналізація інтерфейсу також є важливою частиною перспективи розвитку. Поточна реалізація є універсальною, однак усі користувачі бачать однаковий вигляд застосунку. Подальше вдосконалення UX-досвіду може включати реалізацію темної теми, зміну кольорової палітри, підтримку масштабування шрифтів, вибір розташування блоків на дашборді, а також адаптацію інтерфейсу до особливих потреб (наприклад, високий контраст для людей із вадами зору). Інтерфейс, який користувач може налаштувати відповідно до власних уподобань, значно покращує взаємодію, формує емоційний зв’язок із системою та підвищує частоту використання.

Окремою перспективною ціллю є інтеграція із зовнішніми системами й пристроями. Наприклад, підключення до фітнес-трекерів (Fitbit, Apple Health, Google Fit), що автоматично зчитують кількість кроків, сон і пульс, дозволить мінімізувати ручне введення даних і забезпечити точнішу статистику. Для цього необхідно реалізувати API-з’єднання з відповідними сервісами, обробку авторизації OAuth та механізм синхронізації даних із зовнішнім джерелом. Така інтеграція зробить систему більш автоматизованою, знизить бар’єр входу для користувача й забезпечить постійне оновлення інформації без додаткових зусиль.

Зрештою, масштабування може включати розширення платформи до мобільних застосунків. Поточна web-реалізація вже має адаптивний дизайн, але

створення нативної iOS або Android-версії з підтримкою офлайн-режиму, сповіщеннями й глибшою інтеграцією з пристроями користувача відкриє доступ до ширшої аудиторії. Також перспективним є створення десктопної програми на базі Electron, що дозволить запускати систему як окремий додаток з повною підтримкою функцій. Це особливо актуально для користувачів, які ведуть щоденну аналітику у робочому середовищі та прагнуть мати постійний доступ до своїх показників у зручному форматі.

Отже, перспективи розвитку та масштабування застосунку охоплюють як функціональне розширення (редагування даних, нові показники, інтелектуальні підказки), так і технічну еволюцію (хмарні сервіси, інтеграції, мобільні версії), що в сукупності формують потенціал для трансформації з базового інструмента самоспостереження у повноцінну платформу здорового способу життя з багатофункціональною підтримкою, високим рівнем персоналізації та масштабованістю на рівні команди, родини чи організації. Реалізація цих можливостей залежить від стратегічного бачення подальшого розвитку, технічних ресурсів і зворотного зв’язку від користувачів, які вже формують ядро аудиторії продукту.

Висновки до розділу 3

Функціонування та елементи інтерфейсу розробленого Web-застосунку демонструють цілісну та ефективну реалізацію концепції персоналізованого цифрового інструмента для щоденного моніторингу життєдіяльності користувача. Основні можливості системи забезпечують логічно впорядковану взаємодію, починаючи з автентифікації, внесення даних, перегляду аналітики, до налаштування відображуваних показників. Клієнтський інтерфейс, побудований на компонентній архітектурі React, структурований з урахуванням логіки сценарного використання, що забезпечує швидку навігацію, доступність і передбачувану поведінку. Окрему роль відіграє зручна візуалізація даних, яка не

лише відображає статистику, але й сприяє формуванню регулярної взаємодії з системою.

Оцінка працездатності платформи засвідчує її стабільну реакцію на стандартні дії користувача, ефективне оброблення запитів, надійну авторизацію та цілісність даних. Попри це, наявні потенційні обмеження - зокрема відсутність редагування записів, фіксований набір показників, обмежені аналітичні засоби - окреслюють вектори подальшого вдосконалення. Перспективи розвитку системи включають впровадження інтелектуальної аналітики, інтеграцію з зовнішніми сервісами, реалізацію багатокористувацького функціоналу, а також розширення можливостей інтерфейсу для підвищення індивідуалізації досвіду. В результаті сформовано гнучку основу для масштабування застосунку до рівня повноцінної платформи самоспостереження, орієнтованої на широке коло користувачів.

ВИСНОВКИ

Розробка Web-застосунку для моніторингу та аналізу продуктивності життєдіяльності є актуальним прикладом впровадження сучасних інформаційних технологій у повсякденну практику ведення особистого самоспостереження. Підвищений інтерес до цифрових платформ, які дозволяють фіксувати індивідуальні показники здоров'я, активності, харчування чи психоемоційного стану, зумовлюється не лише зростанням популярності здорового способу життя, а й загальною цифровізацією особистісних процесів. Реалізований проєкт має на меті надати користувачеві простий, доступний та надійний інструмент для контролю власних життєвих параметрів, який базується на актуальних підходах до frontend- та backend-розробки, з використанням сучасних фреймворків і баз даних.

Ключовим елементом створеної системи є аналітична панель - дашборд, який генерується на основі введених користувачем даних. Серверна частина здійснює попередню обробку записів: обчислення середніх значень, агреговані підрахунки, формування часових рядів. Отримані результати надсилаються на клієнтську частину, де відображаються у вигляді графіків або індикаторів. При цьому, особлива увага приділена користувацькому досвіду (UX). Структура інтерфейсу передбачає простоту навігації, наочність основних блоків, миттєвий зворотний зв'язок після дій користувача (збереження даних, повідомлення про помилки, підтвердження дій). Це дозволяє системі залишатись доступною навіть для недосвідчених користувачів і сприяє регулярному використанню.

Тестування працездатності продемонструвало стабільну роботу системи під час стандартних сценаріїв: щоденного введення даних, багаторазового заповнення форм, переходів між сторінками, перевірки автентифікації. Реалізовано механізми захисту від дублювання записів, обробки виняткових ситуацій, відсутності з'єднання або помилок запитів. Інтерфейс забезпечує повну зворотну сумісність з мобільними пристроями - усі основні функції залишаються доступними в мобільному браузері завдяки адаптивній верстці. Завдяки

структурі на базі компонентів React, систему можна легко масштабувати й розширювати без перебудови архітектури.

Водночас, у процесі аналізу функціоналу було виявлено низку обмежень, які визначають напрями подальшого розвитку. Зокрема, важливим напрямом є реалізація повноцінного CRUD-функціоналу, що дозволить користувачеві редагувати або видаляти помилкові записи. Окрему цінність становить розвиток модуля аналітики: інтеграція алгоритмів машинного навчання для автоматичного виявлення відхилень, генерації прогнозів і персоналізованих рекомендацій. Розширення функціоналу можливе також через підключення зовнішніх трекерів.

Отже, у результаті реалізовано стабільний, функціонально завершений і структурно гнучкий Web-застосунок, що виконує основну задачу - фіксацію, обробку та візуалізацію щоденних показників життєдіяльності користувача. Архітектура проєкту дозволяє ефективно управляти даними, персоналізувати інтерфейс, аналізувати статистику та приймати рішення на основі індивідуальних патернів. Отриманий результат довів доцільність застосування сучасних інструментів розробки (Node.js, React, MySQL, JWT, Chart.js), які забезпечують високу якість коду, масштабованість і розширюваність. Незважаючи на обмеження, система створена з урахуванням майбутнього розширення, і тому має значний потенціал розвитку як у напрямку персональної ефективності, так і в сфері цифрового добробуту загалом.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Барановська Т. П. Інформаційні системи у сфері охорони здоров’я. Київ: Центр учбової літератури, 2020. 312 с.

2. Бахмат Н. В. Web-технології у проєктуванні освітніх середовищ. Черкаси : Видавництво ЧНУ, 2019. – 185 с.

3. Бєляєв О. О. Основи програмної інженерії. – Львів : ЛНУ імені Івана Франка, 2021. 248 с.

4. Биков В. Ю., Шишкіна М. П. Хмаро орієнтоване освітнє середовище. – Київ : Педагогічна думка, 2014. 348 с.

5. Ващенко І. П. Комп’ютерні системи підтримки прийняття рішень. Харків : НТУ «ХПІ», 2017. 226 с.

6. Величко В. М. Архітектура інформаційних систем. Київ : КНЕУ, 2018. 214 с.

7. Гаврилюк А. І. Аналіз та візуалізація даних. Тернопіль : ТНТУ, 2022. 176 с.

8. Галіцин В. В. Основи кібербезпеки. Київ : Вид. дім «Освіта України», 2020. 192 с.

9. Гончарук О. А. Розробка програмного забезпечення засобами JavaScript. Одеса : ОНПУ, 2021. 168 с.

10. Грищенко В. І. Методи обробки інформації. Київ : КНЕУ, 2022. 320 с.

11. Данилюк М. І. Основи проєктування баз даних. – Львів : Видавництво ЛНУ, 2019. – 234 с.

12. Дем’янюк В. А. Веб-програмування. Луцьк : СНУ імені Лесі Українки, 2021. 201 с.

13. Джерелейко Т. І. Технології візуалізації результатів досліджень. Чернівці : ЧНУ, 2018. 180 с.

14. Дмитренко А. С. Програмування клієнт-серверних додатків. Дніпро : ДНУ, 2020. 229 с.

15. Євтушенко Л. П. Мови програмування: огляд та аналіз. Київ : НАУ, 2021. 140 с.

16. Залевський В. Ю. Web-сервери та бази даних. Харків : ХНУРЕ, 2020. 275 с.

17. Іванюк І. Я. Програмні платформи: інструменти і технології. Рівне : НУВГП, 2021. 197 с.

18. Калашник О. А. Хмарні технології у веб-розробці. Київ : НАУ, 2022. 210 с.

19. Кириченко Н. В. Інформаційні системи та технології. Суми : СумДУ, 2020. 256 с.

20. Климко С. М. Основи інтерактивного дизайну. Київ : КНУКіМ, 2019. 144 с.

21. Ковальчук Т. В. Розробка веб-застосунків. Житомир : ЖДУ, 2022. 198 с.

22. Козак Ю. Г. Інформаційні технології в менеджменті. Київ : КНЕУ, 2018. 238 с.

23. Кононенко С. І. Системи керування базами даних. Хмельницький : ХНУ, 2021. 172 с.

24. Кравець Л. Б. Тестування програмного забезпечення. Вінниця : ВНТУ, 2019. 163 с.

25. Кузьменко О. О. Адаптивні інформаційні системи. Чернігів : ЧНТУ, 2020. 200 с.

26. Левченко І. С. Системне адміністрування. Київ : КНУТД, 2021. 194 с.

27. Макаренко О. П. Основи JavaScript. Полтава : ПНПУ, 2019. 183 с.

28. Марченко М. Ю. Клієнт-серверна архітектура. Київ : КНЕУ, 2021. 224 с.

29. Матвієнко Н. С. Електронні освітні ресурси. Харків : УІПА, 2018. 157 с.

30. Мельник А. І. Створення інтерфейсів користувача. Тернопіль : ТНПУ, 2020. 209 с.

31. Міщенко В. С. Node.js: архітектура та застосування. Київ : КПІ, 2022. 188 с.

32. Назаренко Л. І. REST API: теорія і практика. – Львів : ЛНУ, 2021. 176 с.

33. Олійник В. Ю. Мережеві технології. Дніпро : ДНУ, 2020. 211 с.

34. Орлик Р. С. Автоматизовані системи збору даних. Київ : НАУ, 2019. 236 с.

35. Павленко М. І. Веб-аналітика. Суми : СумДУ, 2022. 149 с.

36. Панченко А. В. Сучасні інструменти frontend-розробки. Луцьк : СНУ, 2020. 198 с.

37. Петренко Ю. О. UX/UI-дизайн: основи. Київ : КНУКіМ, 2021. 132 с.

38. Піскун І. М. Розробка адаптивних інтерфейсів. Вінниця : ВНТУ, 2022. 174 с.

39. Поліщук О. В. Захист інформації в web-системах. Черкаси : ЧДТУ, 2020. 190 с.

40. Рибак І. В. Серверна логіка в Node.js. Харків, 2021. 206 с.

41. Романенко А. В. React.js: від основ до проєкту. Київ : КПІ, 2022. 180 с.

42. Савчук А. І. Програмна інженерія. Львів : ЛНУ, 2020. 243 с.

43. Семенюк Д. Б. Інформаційна безпека. Київ : КНУ, 2021. 220 с.

44. Соловей Ю. В. Бази даних та SQL. Житомир : ЖДУ, 2019. 168 с.

45. Старченко Л. М. Web-дизайн. – Київ : НАОМА, 2020. – 155 с.

46. Тимошенко К. С. Автентифікація в веб-системах. Луцьк : СНУ, 2021. 164 с.

47. Ткаченко І. О. Веб-архітектура. Полтава : ПНПУ, 2022. 178 с.

48. Федоренко Н. П. Кешування і продуктивність. Харків : ХНУРЕ, 2021. 160 с.

49. Хоменко Л. Г. Аналіз користувацьких сценаріїв. Київ : КНУТД, 2020. 195 с.

50. Ярошенко О. Д. Розробка SPA-додатків. Львів : ЛНУ, 2022. 187 с.

Додаток Е

ЗГОДА здобувача(чки) вищої освіти

Державного університету економіки і технологій про
перевірку кваліфікаційної роботи на прояви
академічного плагіату
та розміщення в Репозитарії Університету

Я, Даць Софія Володимирівна (ПІП),
підтримую політику Державного університету економіки і технологій з академічної доброчесності і відкритого доступу.

Засвідчую, що кваліфікаційна бакалаврська (магістерська) робота

на тему: „Розробка Web-застосунку для моніторингу та аналізу продуктивності діяльності"

(назва роботи повністю) виконана самостійно та не містить академічного плагіату. Я не надавав(ла) і не одержував(ла) недозволену допомогу під час підготовки цієї роботи. Робота містить результати власних досліджень. Використання ідей, результатів і текстів інших авторів мають посилання на відповідне джерело.

Із чинним Положенням про запобігання та виявлення академічного плагіату в роботах здобувачів вищої освіти Державного університету економіки і технологій ознайомлений(а). Чітко усвідомлюю, що в разі виявлення у кваліфікаційній роботі порушення норм академічної доброчесності робота не допускається до захисту або оцінюється незадовільно.

Також я поінформований(на), що відповідно до «Положення про Репозитарій (електронну базу даних) Державного університету економіки і технологій» зазначена робота буде розміщена в Електронному архіві Університету (Репозитарії ДУЕТ). З умовами такого розміщення ознайомлений(на).

Дата	підпис	ініціали, прізвище (власноруч)
10.06.2025р.		Даць С.В.