НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ
імені ІГОРЯ СІКОРСЬКОГО»

Факультет інформатики та обчислювальної техніки

Кафедра інформаційних систем та технологій

Індивідуальний дослідницький проєкт

на здобуття ступеня бакалавра
за освітньо-професійною програмою «Інтегровані інформаційні системи»
спеціальності 126 «Інформаційні системи та технології»
на тему: «Конструктор шаблонів для розробки вебзастосунків»

Виконав:
студент IV курсу, групи ІА-82
Рогов Олександр Олександрович _____

Керівник:
Викладач каф. ІСТ,
Моргаль Олег Михайлович _____

Засвідчую, що у цьому дослідницькому проєкті немає запозичень з праць інших авторів без відповідних посилань.
Студент _____

Київ – 2022 року

Національний технічний університет України
«Київський політехнічний інститут імені Ігоря Сікорського»
Факультет інформатики та обчислювальної техніки
Кафедра інформаційних систем та технологій

Рівень вищої освіти – перший (бакалаврський)

Спеціальність – 126 «Інформаційні системи та технології»

Освітньо-професійна програма «Інтегровані інформаційні системи»

ЗАВДАННЯ

на індивідуальний дослідницький проєкт студенту

Рогову Олександру Олександровичу

1. Тема проєкту «Конструктор шаблонів для розробки вебзастосунків», керівник проєкту викладач Моргаль Олег Михайлович

2. Термін подання студентом проєкту: 15 червня 2022 року

3. Вихідні дані до проєкту:

Мова програмування Typescript, середовище розробки Visual Studio Code, СУБД MongoDB, архітектурний підхід Redux, фреймворк для серверної частини Express, бібліотека для візуальної частини React, розгортання на сервісі Netlify з використанням Netlify Functions.

4. Зміст пояснювальної записки:

Перелік термінів та скорочень, вступ, постановка завдання на індивідуальний дослідницький проєкт, дослідження предметної області, вибір технологій та засобів розробки, інструкція користувача, висновки, перелік використаної літератури.

5. Перелік графічного матеріалу

Інфологічна модель предметної області, діаграма компонентів, діаграма залежностей, UML-діаграма варіантів використання

6. Дата видачі завдання 1 грудня 2021 року

Календарний план

№ з/п	Назва етапів виконання проєкту	Термін виконання етапів проєкту	Примітка
1	Вивчення теоретичних матеріалів та аналіз предметної області	18.04.2022	Виконано
2	Огляд існуючих рішень	25.04.2022	Виконано
3	Вибір технологій та засобів для розробки	25.04.2022	Виконано
4	Проєктування СУБД	02.05.2022	Виконано
5	Реалізація проєкту	16.05.2022	Виконано
6	Оформлення пояснювальної записки	06.06.2022	Виконано

Студент Олександр РОГОВ

Керівник Олег МОРГАЛЬ

АНОТАЦІЯ

Рогов О. О. Конструктор шаблонів для розробки вебзастосунків. КПІ ім. Ігоря Сікорського, Київ, 2022.

Пояснювальна записка містить 81 с. тексту, 42 рисунки, 1 додаток та 20 літературних джерел. Графічна частина включає 4 кресленики формату А3.

Ключові слова: шаблони, веб-додаток, бази даних, залежності, конфігурування, конструктор шаблонів.

Об’єктом розробки є конструктор шаблонів для розробки вебзастосунків.

Мета розробки – створення застосунку для управління інформацією про шаблон, додавання та конфігурування його залежностей, збереження створеного шаблону віддалено та з можливістю завантаження локально.

При проєктуванні та розробці індивудального дослідницького проєкту було створено застосунок на базі MERN стеку з використанням мови програмування Typescript та середовища розробки Visual Studio Code. Проведено аналіз предметної області та визначено ключові сутності, щодо яких відбувалась подальша розробка. Застосунок містить базовий функціонал для оперування над шаблонами, сучасну спроєктовану архітектуру і простий та зрозумілий користувацький інтерфейс.

Отримані результати можуть бути використані для вивчення досліджуваної проблематики та отримання більшого розуміння про проблему користувача, яку може вирішити конструктор шаблонів для розробки вебзастосунків.

SUMMARY

Rohov O.O. Template designer for the web application development. Igor Sikorsky KPI, Kyiv, 2022.

The explanatory note contains 81 pages of text, 42 figures, 1 appendices and 20 literary sources. The graphic part includes 4 drawings of A3 format.

Keywords: templates, web application, databases, dependencies, configuration, template designer.

The object of development is a template designer for web application development.

The purpose of the development is to create an application for managing information about the template, adding and configuring its dependencies, saving the created template remotely and with the ability to download locally.

During the design and development of the thesis project, an application based on the MERN stack was created using the Typescript programming language and the Visual Studio Code development environment. The analysis of the subject area is carried out and the key essences concerning which further development took place are defined. The application contains basic functionality for operating on templates, modern designed architecture and a simple and clear user interface.

The results can be used to study the research problem and gain a better understanding of the user problem that can be solved by the designer of templates for web application development.

Номер рядка	Формат	Позначення	Найменування	Кільк. аркуші	Номер екзем.	Примітка
1			Документація загальна			
2						
3			Знову розроблена			
4						
5	А4	ІА82.210БАК.003 ПЗ	Пояснювальна записка	83		
6						
7	А3	ІА82.210БАК.003 Д1	Конкструктор шаблонів для	1		
8			розробки вебзастосунків.			
9			Інфологічна модель			
10						
11	А3	ІА82.210БАК.003 Д2	Конструктор шаблонів для	1		
12			розробки вебзастосунків.			
13			Діаграма компонентів			
14						
15	А3	ІА82.210БАК.003 Д3	Конструктор шаблонів для	1		
16			розробки вебзастосунків.			
17			Діаграма залежностей			
18						
19	А3	ІА82.210БАК.003 Д4	Конструктор шаблонів для	1		
20			розробки вебзастосунків.			
21			Діаграма варіантів			
22			використання			
23						
24						
25						
26						
27						
28						

Зм.	Аркуш	№ докум.	Підпис	Дата	ІА82.210БАК.003 ТП		
Розроб.		Рогов О.О.			Конструктор шаблонів для розробки вебзастосунків. Відомість проєкту	Літ.	Аркуш / Аркушів
Керівн.		Моргаль О.М.				Т	1
						КПІ ім. Ігоря Сікорського ФІОТ Група ІА-82	
Затв.							

Пояснювальна записка
до індивідуального дослідницького проєкту
на тему: «Конструктор шаблонів для розробки вебзастосунків»

Київ – 2022 року

ЗМІСТ

ПЕРЕЛІК ТЕРМІНІВ ТА СКОРОЧЕНЬ

API - Application Programming Interface

DB - database

NPM - Node Package Manager

DOM - Document Object Model

IDE – Integrated Development Environment

JSON - JavaScript Object Notation

NoSQL - not only Structured Query Language

БД - база даних

CDN - Content Delivery Network

HTTP - HyperText Transfer Protocol

REST - Representational State Transfer

DNS - Domain Name System

SSL - Secure Sockets Layer

ODM - Object Data Modeling

DRY - Don't Repeat Yourself

CRUD - Create-Read-Update-Delete

AJAX - Asynchronous JavaScript and XML

XML - Extensible Markup Language

URL - Uniform Resource Locator

HTML - HyperText Markup Language

ВСТУП

Під час процесу розробки програмного забезпечення, перед технічним спеціалістом, найчастіше, постає задача написання коду. Але, не весь час її вирішення, є створення бізнес-логіки, що є ключовим виміром ефективності для клієнта. Технічний спеціаліст все більше знаходить себе у ситуації, коли репетитивні дії можна автоматизувати, а час процесу налагодження середи розробки та шанс на людську помилку мінімізувати, тим самим підвищити ефективність створення програмного забезпечення.

Метою індивідального дослідницького проєкту є створення веб-додатку, який в свою чергу надасть можливість конструювати шаблони коду із мінімально-необхідним налаштуванням для прискорення початкового етапу розробки. Функціонал додатку можна поділити на такі особливості:

- отримання, додавання та редагування інформації про створений шаблон;
- надання користувачу інформації про можливі структурні рішення для шаблону;
- додавання та редагування структури створеного шаблону;
- надання користувачу інформації про пакетні залежності сторонніх бібліотек;
- додавання та редагування залежностей шаблону від сторонніх бібліотек;
- конфігурування сторонніх бібліотек;
- можливість збереження створеного шаблону.

Для виконання завдання проєкту було поставлено на виконання наступні задачі:

- ознайомлення з видами робіт, що виконуються під час побудови шаблонів для розробки веб-застосунків;
- аналіз предметної області та визначення основних сутностей процесу взаємодії з застосунком;

- ознайомлення з існуючими аналогами, їх перевагами та недоліками;
- вибір та дослідження технологій для проєктування та розробки додатку;
- формування вимог до користувацького інтерфейсу та функціоналу додатку;
- проєктування інфраструктури;
- реалізація додатку.

Особливістю розробленого веб-застосунку є можливість збереження шаблонів віддалено та на локальному накопичувачі, конфігурування залежностей, автоматична перевірка на валідність вхідних даних та наявність зручного візуального інтерфейсу. Дослідницький проєкт складається з наступних розділів: вступ, постановка завдання на проєкт, дослідження предметної області, вибір технологій та засобів розробки, проєктування та реалізація застосунку, інструкція користувача, висновки, список використаних джерел із 20 найменувань, 1 додаток. Графічна частина включає 4 кресленики формату А3. Загальний обсяг 81 сторінка.

1 ПОСТАНОВКА ЗАВДАННЯ НА ДОСЛІДНИЦЬКИЙ ПРОЄКТ

У даному дослідницькому проєкті необхідно розробити рішення для користувача, що дозволяє спростити первинний етап розробки веб-застосунків, надавши можливість створювати шаблони для подальшої розробки, для цього було вирішено використовувати високорівневу мову програмування JavaScript.

У розробленому додатку повинна бути присутня реалізація наступних функцій:

- отримання та редагування параметрів про створюваний шаблон
- надання пропозицій користувачу про можливі структурні рішення для шаблону;
- створення та змінення файлової структури створеного шаблону;
- надання інформації користувачу про залежні зв’язки із сторонніми бібліотеками;
- додавання сторонніх бібліотек до створеного шаблону та можливість їх видалення;
- налагодження конфігурації сторонніх бібліотек в шаблоні;
- збереження обраних параметрів та подальший експорт шаблону користувачеві.

Повинно бути реалізовано візуальне повідомлення про несумісність пакетних залежностей за версіями, або некоректних даних користувачем. Для управління функціями конструктора та параметрів створюваного шаблону необхідно розробити меню, та візуалізувати елементи управління, щоб забезпечити зручне та інтуїтивне використання сервісу користувачем.

Додаток буде розроблений як веб-середовище із переліком вищезгаданих структурних елементів із їх відповідними функціональними обов’язками. Створений сервіс буде можливим використати за посиланням у браузері, тому повинен бути адаптованим під старіші версії браузерів.

2 ДОСЛІДЖЕННЯ ПРЕДМЕТНОЇ ОБЛАСТІ

Досліджуваним об’єктом в даній дослідницькій роботі є початковий етап розробки веб-застосунків, яким є створення структури проєкта та його конфігурації. Створюваний сервіс розглядається як спеціалізований саме для веб-застосунків, але не є прив’язаним до єдиного підходу чи деякої технології.

Перевагою такого гнучкого рішення є надання широкої можливості конфігурування та структуризації проєкту перед початком його розробки, що дозволяє зменшити час необхідний для додавання необхідних залежностей, формуванні папкової структури та мінімізувати шанс людської помилки.

В проєктованій системі був зроблений акцент на додаванні та видаленні пакетних модулей шаблону, отримання інформації про їх поточний стан, актуальність, зміні загальної конфігурації та редагуванні конкретної для кожної із доданих пакетних залежностей, а також збереження шаблону для його подальшого використання.

Сервіс за своєю роботою можна поділити на кілька етапів:

- отримання від користувача вхідних даних для обробки, обрання структури шаблону та загальних особливостей;

- проведення перевірки користувацьких даних та конфігураційних налаштувань на валідність;

- визначення пакетних залежностей, які можуть бути додані та пропозиції про можливі архітектурні рішення;

- надання інформації про стан пакетних залежностей, доступних версій, міжпакетних конфліктів та варіантів їх вирішення, в разі необхідності;

- вихідний контроль;

- збереження шаблону та надання його користувачу.

В роботі системи бере участь вибірка із основних сутностей, це користувач, який надає вхідні дані для отримання шаблону, інтерфейс прикладного управління для їх обробки (API), та сайт, який грає

безпосередню роль у візуальному зображенні меню користувацького управління.

В реальних умовах, на місці API может бути будь-який відкритий сервіс, який в залежності від вхідних даних, надаватиме інформацію про пакетні модулі та передаватиме вихідний шаблон у форматі, можливому для завантаження через браузер.

Уся робота виконана зовнішнім інтерфейсом, в свою чергу може бути поділена на кілька етапів:

- надання пропозицій про пакетні модулі за пошуковим запитом;
- надання інформації про конкретний пакетний модуль;
- аутентифікація існуючих юкористувачів та реєстрація нових;
- надання списку шаблонів авторизованого користувача;
- додавання нового шаблону та збереження його у базі даних;
- додавання та видалення пакетних залежностей шаблону;
- зміна інформації про пакетні залежності та їх конфігурація;
- надання користувачу можливості завантажити збережені шаблони.

Відповідно, API включає в себе інтерфейсні виклики, які несуть відповідальність за виконання певного типу роботи, і меню користувацького управління, яке відповідає згаданому функціоналу із додатковим візуальним сповіщенням із релевантною інформацією про шаблон.

При створенні нового шаблону для розробки веб-застосунків потрібно зберігати загальну інформацію про потенційний проєкт, а саме його назву, опис, ключові слова, автора, рівень публічної доступності тощо. У разі додавання спільно несумісних пакетних залежностей, або при введенні некоректних даних у конфігурацію шаблону чи індивідуальних його частин, необхідно проводити загальний вхідний контроль та аналіз ризиків дій користувача і сповіщення його про наслідки.

2.1 Огляд існуючих рішень

2.1.1 Інструмент Create React App для створення React додатку

Скрипт Create React App забезпечує користувача рядом функцій, серед яких:

- автоматичне створення проєкту за попередньо визначеним шаблоном з упередженою папковою структурою “однією командою в терміналі”;

- першочергове неявне встановлення пакетних модулей без можливості отримати інформацію про їх назви чи версії;

- підтримка версійності залежностей розробниками, через це зберігається принцип єдиного джерела правди, користувач не може змінювати інформацію про залежності створеного шаблону вручну.

Детальний опис всього циклу роботи скрипту, його особливостей, підтриманих можливостей, шаблону створеного проєкту та багато іншого викладено в офіційній документації.

- просте використання - при виклику скрипту потрібно вказати тільки папку для завантаження та встановлення всіх даних;

- сповіщення користувача про некоректно введені вхідні дані, шлях до папки, або неможливість підключення до інтернету тощо;

- тільки одна пакетна залежність - скрипти всередині інструменту, на яких він базується;

- команди запуску, побудови проєкту та тестування, вже прописані у загальному конфігураційному файлі;

- складність налаштування підтримки рендеру на стороні серверу через відсутність першочергової можливості редагування створеного шаблону під потребності користувача.

Основним недоліком скрипту є відсутність візуального інтерфейсу (рисунок 2.1) та неможливість контролю встановлюваних бібліотек та редагуванні їх конфігураційних файлів при створенні проєкту, файлової структури, оскільки він позиціонується як інструмент для пришвидшення

веб-розробки із не глибокою кривою навчання та абстракції від конфігурацій та налаштування.

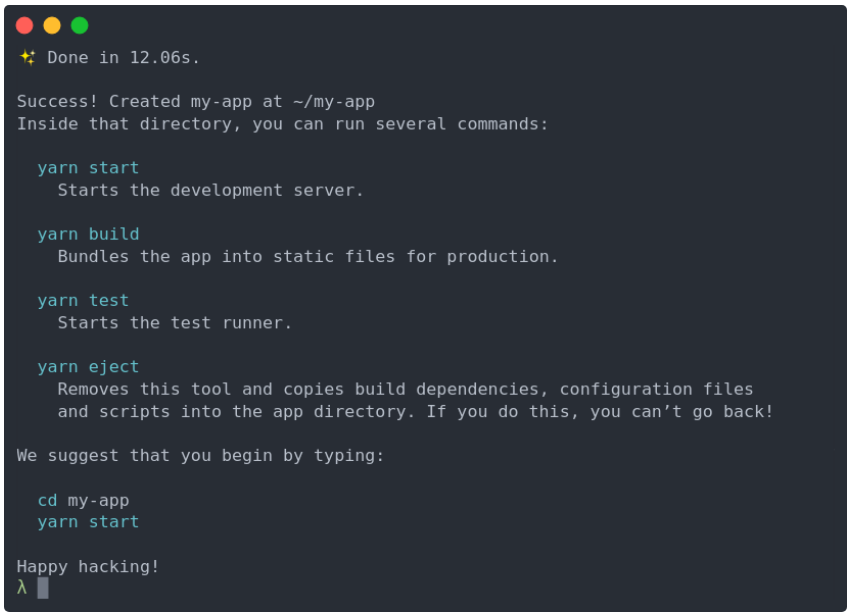

Рисунок 2.1 – Вивід терміналу після використання інструменту CRA

2.1.2 Веб-додаток Spring Initializr

Зразок користувацького інтерфейсу зображено на рисунку 2.2.

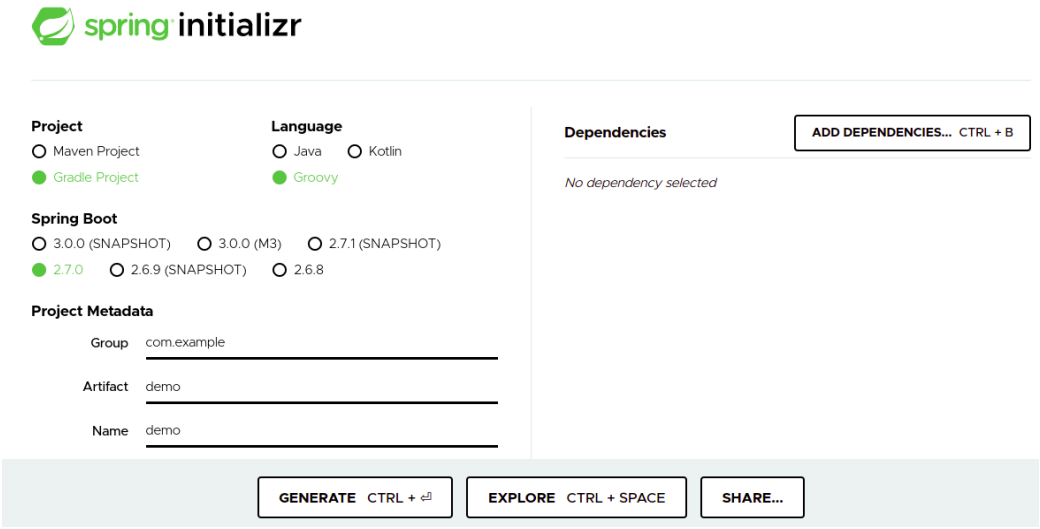

Рисунок 2.2 – Інтерфейс веб-додатку Spring Initializr

Веб-додаток Spring Initializr [1] – це безкоштовний універсальний інструмент для пришвидшення створення проєкту на мові програмування Java із зазначеними бажаними вхідними даними. Базовий шаблон може підійти розробнику, який має цілі абстрагуватися від початкового етапу розробки, який передбачає за собою процес додавання мета-даних шаблону для майбутнього проєкту та бібліотек чи модулей для розробки із наданого списку.

Додаток «Spring Initializr» дозволяє: встановити помічник, який буде використовуватися як підґрунтя для подальшої побудови проєкту, обрати мову програмування із переліку Java базованих під-мов, зробити вибір версії створеного шаблону.

Додаток дозволяє обрати пакетні залежності із існуючого переліку сортованого за категоріями, але без можливості обрання версії пакетів та редагування їх конфігурацій.

Створений шаблон можна поширити у вигляді посилання для передачі можливості його переглядання, змінення, та рядом інших потенційно можливих дій зі сторони користувача. Сайт, на якому розгорнутий додаток, має лаконічний інтерфейс, із візуальними компонентами зрозумілими за своїм функціональним призначенням.

Також його комфортно переглядати та використовувати вночі, адже є підтримка темного оформлення кольорів для зниження навантаження на сприйняття картинки людським зором.

Основні переваги:

- майже миттєва готовність до запуску проєкту відразу після встановлення на комп'ютер всіх залежностей, вказаних в загальному конфігураційному файлі, немає необхідності проводити додаткові налаштування модулів та компонентів;

- загальний огляд потенційної папко-файлової структури проєкту;

- додавання та видалення залежностей шаблону від пакетних модулей;

- мінімалістичний та інтуїтивно-зрозумілий користувацький візуальний інтерфейс;

- можливість поширення створеного шаблону за посиланням;

- збереження шаблону шляхом завантаження до локального дискового сховищу.

Недоліком додатку є максимальна прив’язка до мови програмування Java та неможливість обрання версій пакетних залежностей і отримання додаткової корисної інформації про актуальність, підтримку бібліотек чи модулей спільнотою, кіл-ть завантажень за деякий період для оцінки поширеності тощо.

2.1.3 Ручне налаштування

Як альтернативний підхід до створення шаблону проєкту, так званого “boilerplate”, можна розглядати мануальне (див. ручне) налаштування шаблону проєкту з нуля без використання додаткових допоміжних інструментів чи додатків.

Для цього потрібно бути ознайомленим із рядом інструкцій необхідних для отримання шаблону і подальшого використання його для стартового етапу розробки, бути досвідченим у створенні архітектури, підключенні модулів та їх оцінки виходячи з аналізу актуальності використання та поширеності в світовій спільноті.

Даним альтернативним методом є гнучка система з можливістю редагування конфігураційних файлів на загальному рівні, та на рівні окремих модулів.

Версійність програмних пакетів обирається самим користувачем і не обмежується вимушеною вибіркою.

Попередньо збережений шаблон можна перевикористати та модифікувати під конкретний випадок використання. За допомогою ручного налаштування гарантується повний контроль над метаданими, надається

можливість додавати ті пакетні залежності, не перелікові ні в єдиному каталозі модулів, розроблені самостійно.

Немає обмеження по вибору мови програмування чи направленості проєкту на зазначену платформу, веб чи програмний додаток.

Основними недоліками мануального налаштування є підвищений час на дослідницьку діяльність по вивченню інформаційної складової пакетних модулів в інтернеті, відсутність зручного графічного інтерфейсу та велика ймовірність людської помилки.

2.2 Інфологічна модель предметної області в нотації Чена

Інфологічна модель має за собою намір відобразити реальний світ у деякій концепції, зрозумілій людині, яка є цілком незалежною від параметрів середовища даних, звідки вони надходять.

Підходи до побудови таких моделей можна представити у вигляді однієї із форм в переліку обмеженої множини: графові моделі, семантичні мережі, модель "сутність-зв'язок" і т.д.

Модель "сутність-зв'язок" виявилася популярнішою серед інших. Ця модель ще має назву ER-модель (від англ. Entity-Relationship, тобто сутність-зв'язок). Моделювання предметної області базується на використанні графічних діаграм, що включають в себе задане кінцевий набір компонентів різного роду. У зв'язку з наочністю уявлення концептуальних схем системи, ER-моделі одержали широке поширення в системах CASE, що підтримують автоматизоване проєктування розробляємої системи.

У них сутності найчастіше зображуються прямокутниками, асоціації (зв'язки) - позначеними ромбами, атрибути - позначеними овалами, а зв'язки між ними - об’єднуючими ребрами без напряму, над якими може проставлятися ступінь зв'язку (1 або буква, що заміняє слово "багато") і необхідне пояснення.

Інфологічна модель предметної області даних зображена в документі ІА82.210БАК.003 Д1.

На моделі зображені основні сутності системи, а саме:

- інформація про користувача, або автора шаблону. Йому присутній єдиний атрибут ім’я.

- інформація про шаблон з обов'язковими для нього мета-атрибутами (назва, опис, автор);

- пакетні залежності, які присутні після додавання їх до шаблону. Атрибутами пакетного модуля є назва, опис, версія, та актуальність пакету, його додаткова інформація;

- папкова структура створеного шаблону. Атрибутами сутності є назва папок та рівень вкладеності;

- вихідний файл шаблону після зберігання. Атрибутами є інформація про автора шаблону, інформація про сам шаблон, пакетні залежності, структуру, розмір файлу;

- сервер для обробки запиту користувача. Атрибутами є, вхідні параметри запиту та вихідний файл;

- валідація введених даних. Атрибутами є назва поля та правило перевірки.

Висновки до розділу 2

У даному розділі було розглянуто та досліджено деякі існуючі рішення у формі скриптового інструменту, веб-застосунку, та ручного налаштування, направлених сумісно на вирішення проблеми ефективного пришвидшення стартового етапу процесу розробки програмного забезпечення.

Основні недоліки готових рішень:

- відсутність візуального інтерфейсу та неможливість контролю встановлюваних бібліотек та редагуванні їх конфігураційних файлів при створенні проєкту та файлової структури шаблону;

- прив’язка до конкретної мови програмування та неможливість обрання версій пакетних залежностей і отримання додаткової корисної інформації про актуальність тощо;
- підвищений час на аналіз потенційних бібліотек та інших залежностей шаблону;
- складність налаштування конфігураційних файлів та велика ймовірність людської помилки.

Відповідно до цих недоліків можна сформувати перелік вимог до функціоналу сервісу, який розробляється в межах теми дослідницького проєкту.

Основною вимогою є впровадження веб-застосунку з легким у сприйнятті інтерфейсом для користувачів а також підтримка та забезпечення роботи сервісу на старіших версіях браузерів. Важливим аспектом є простота у використанні та можливість збереження створеного шаблону на локальному дисковому сховищі. Також у розділі було проведено аналіз предметної області та визначено основні сутності проєктованої системи, а також атрибути цих сутностей. На основі визначених даних побудовано інфологічну модель предметної області з відображенням сутностей та зв’язків у нотації Чена.

Дана модель відповідає вимогам, висунутим до системи, що проєктується, тому доцільним буде використання цієї моделі для подальшого проєктування перелічених сутностей, створення архітектурної моделі системи, що проєктується та програмної реалізації.

3 ВИБІР ТЕХНОЛОГІЙ ТА ЗАСОБІВ ДЛЯ РОЗРОБКИ

3.1 Мова програмування Typescript

Typescript - це надмножина над мовою програмування JavaScript, що має за собою мету розширити існуючий функціонал та можливості вже існуючої мови. За своїм походженням її можна вважати об'єктно-орієнтованою мовою програмування загального призначення для статичної перевірки типів на етапі написання коду. На відміну від її посередника, потребує компіляції. Розробники програмного забезпечення знайшли застосування мови Typescript для написання візуальної клієнтської частини та серверної, використовуючи її розширені можливості, одна з яких це статична типізація.

Цей інструмент при написанні коду може бути скомпільований на різних користувацьких платформах. Найчастіше, встановлюється як глобальний пакетний модуль на робочу машину, налаштовується конфігураційний файл який визначає правила компіляції відповідно до створеного проєкту, та використовується у файлах із розширенням типу “.ts” і “.tsx”.

Мова програмування Typescript [8] є платформо-агностичною, тому може бути використана в розробці веб додатків, мобільних застосунків тощо. Популярність зросла після визначення спільнотою JavaScript мовою програмування нового покоління та все ширшого знаходження її застосування.

Для роботи з мовою Typescript необхідно встановити IDE (інтегроване середовище розробки) з підтримкою підсвічування синтаксичних конструкцій. Розглядаючи як один із альтернативних варіантів розробки, вихідний код Typescript можна писати за допомогою вбудованого в більшість оперативних систем текстового редактора за замовчуванням, наприклад Блокноту, та компілювати за допомогою командного рядка терміналу

оперативної системи із встановленим однойменним пакетним модулем-компілятором.

Мова э платформо-агностичною та середовище-агностичною, надаючи розробнику вдосконалений шлях розробки, зручний та придатний для написання код із можливостями створення типів та перевірки на етапі написання, прибираючи їх із компільованого коду, яким являється код синтаксису JavaScript.

Оскільки Typescript є обгорткою над мовою програмування JavaScript, розробник має можливість доступу до глобальної численної бібліотеки модулів NPM, лінтерів, інших допоміжних засобів розробки таких як Babel для транспіляції коду в більш старий синтаксис коду, ESLint для встановлення правил використання синтаксичних конструкцій для підтримки узгодженості коду та кращих практик розробки, Vite як збірний інструмент усіх модулів та підключення додаткових плагінів та ін.

3.2 MongoDB

База даних MongoDB є представником NoSQL баз даних, зберігаючи всі записи у вигляді документів в форматі об’єктів конвенції JSON. Об'єкт в мові є окремою сутністю з властивостями та типом. Клас об'єкта представляє один із типів даних JavaScript. Він використовується для зберігання різноманітних колекцій ключів і більш складних сутностей Так як мова програмування JavaScript використовує у своїх механізмах представлення функцій та інших сутностей “поза сценою” об’єкти, їх можна вважати зручним способом використання та трансформації інформації у зв’язці JavaScript та NoSQL представником баз даних MongoDB.

Для зовнішнього доступу до бази даних API власне MongoDB надає користувачу набір методів, що можуть бути викликаними. Існує також можливість інтеграції з будь-яким додатком, або середовищем виконання

вихідного коду, яким наприклад є NodeJS, отримавши доступ до бази даних через API. Робота через API походить від імені попередньо авторизованого користувача.

Для базового використання, покриваючи необхідність зберігання даних, їх передачу, валідацію, додавання та видалення, достатньо безкоштовного аккаунту. Для безкоштовного тарифу діють деякі обмеження. Наприклад, дозволений одночасне підключення до бази даних тільки п’ятистам користувачів, а перераховані раніше операції можуть бути здійснені лише в кількості однієї сотні в секунду.

MongoDB являється системою взаємодії та управління нереляційною базою даних на основі документів-об’єктів. Також, взаємозамінно її називають об’єктно-орієнтованою системою. Цей інструмент був розроблений після зростання популярності MySQL для подальшої заміни структури реляційної бази даних, зарекомендувавши себе як спрощенний спосіб роботи з даними у зовнішньому сховищі. У порівнянні з MySQL, яка є системою на основі таблиць, або реляційною базою даних, NoSQL у свою чергу не оперує зв’язками між документами.

Ще одна з відмінностей яка може мати вагу при виборі типу сховища є той факт, що перелічений вище тип бази даних швидше, ніж MySQL, завдяки його здатності обробляти великі обсяги неструктурованих даних. Також, вагомою рисою становить можливість використання MongoDB із будь-якими мовами програмування, так як популярність цієї СУБД розповсюдила створення пакетних модулей для зручної інтеграції в проєкт та посилання запитів, виступаючи в ролі провайдера.

Працювати з базою можна через особистий виділений кластер в веб середовищі на офіційному сайті бази даних [9], яке в свою чергу створює рівень абстракції для розробника, оперуючи з даними через авторизованого провайдера з обліковими даними активного аккаунту.

MongoDB розроблена як розподілена база даних. Кластери можуть бути створені або склоновані в реальному часі. А великі колекції, або колекції

з високим навантаженням чи значним ресурсоспоживанням можна розподілити між кількома кластерами для підтримки продуктивності та горизонтального масштабування. Тобто, щоб впоратися з новими вимогами, до існуючої інфраструктури можуть бути додані додаткові вузли або машини.

3.2.1 Mongoose ODM

Mongoose ODM (від англ. Object Data Modeling - моделювання об’єктних даних) це бібліотека для нереляційної бази даних з NoSQL MongoDB та середовища NodeJS. Ця бібліотека допомагає керувати зв’язками між існуючими даними. При експлуатації даних у нереляційної бази даних можна виділити декілька важливих етапів, які забезпечуються цим допоміжним інструментом, це перевірка створених схем що використовуються, перенос записів між колекціями об’єктів-документів (таблиці в SQL) та представлення цих об’єктів у MongoDB.

При розробці із використанням стеку без включення бібліотеки Mongoose потребується написання значно більшої кількості строк коду, аніж у порівнянні з нею. MongoDB - це база даних із документами без строго заданих схем на відміну від того, який підхід використовується в SQL базах даних. Це означає, що при необхідності, в ній можна зберігати документи JSON із будь-якою структурою цих документів. Це одна з переваг використання NoSQL, оскільки такий вибір технології прискорює розробку додатків і зменшує складність розгортання застосунку.

Та в будь-якому випадку, є корисною практикою схеми даних створити, а вхідні дані до них проганяти через процес їх валідації. Для спрощення циклу розробки, бібліотека Mongoose дозволяє попередньо дефініювати схему об’єктів відповідної колекції. Хоча база даних Mongo не має схем, а SQL визначає схему за допомогою визначення реляційних зв’язків та сутностей у таблиці.

Схема у Mongoose - це деяка форма, або структура даних документа, яка задається через прикладний рівень програмного додатку.

3.3 Функціональне програмування

Функціональне програмування [4] - це одна з найбільш використовуваних парадигм програмування у сучасній розробці. Природа цієї парадигм диктує, що вихідний код повинен будуватися шляхом застосування та складання функцій. Парадигма функціонального програмування [5] підпадає під рід декларативного програмування, в якій функції визначаються як вирази, які відображають суть виконуваної функції, а не послідовність визовів імперативних операторів, що оновлюють стан виконання програми, на відміну від імперативного програмування.

Функціональне програмування, як парадигма, розглядає всі створені функції як передбачувані математичні функції, або чисті функції. Коли чисті функції викликаються із деякими заданими аргументами то результат, повернений цими функціями не буде змінюватися, до тих пір, поки аргументи не зміняться. На відміну від нечистих процедурних викликів, поширених в імперативному програмуванні, вони не можуть мати вплив скоєний змінюваними станами, приймати вхідні дані від користувача, чи інші побічні ефекти.

Побічними ефектами є будь-які зміни стану програмного середовища, які відбуваються за межами викликаної функції. При обранні функціонального програмування як базу подальшої розробки, є сенс дотримуватися кращих практик. Прихильники суто цієї парадигми стверджують те, що обмежуючи побічні ефекти в програмі, можна по-перше звести кількість помилок до мінімума.

По-друге, програмні помилки, що виникли в процесі розробки, стає легше детерміновувати та тестувати, а також вони більше підходять для

перевірки без написання автоматичних тестів, так званої формальної перевірки.

Найголовніша характеристика функціонального програмування яку можна відокремити, це фокусировка парадигми на результатах виконання синтаксичних конструкцій вихідного коду, а не на процесі завдання детального їх опису.

Популярний підхід функціональній розробці є функції вищого порядку, які дозволяють застосовувати їх частково. Ця техніка застосовує функцію до своїх аргументів по черзі, оскільки кожне виконання повертає нову функцію, яка приймає наступний аргумент. Каррування або каррінг (англ. currying) в програмній розробці - метод обчислення функції від багатьох аргументів, перетворенням її в послідовність функцій одного аргумента.

В об’єктно-орієнтованому програмуванні з’являється проблема у підтримці об’єктів із поступовим ростом рівня успадкування. Це також порушує принцип інкапсуляції і створює не розділений на модулі код. З іншого боку, у функціональному програмуванні для кожної функції створюється новий об’єкт, тому для виконання програм потрібно значно більше оперативної пам’яті.

3.4 Бібліотека React

React - безкоштовна бібліотека з відкритим вихідним кодом для побудови користувацьких веб-інтерфейсів. Написана на мові програмування JavaScript, велика підтримка надходить від корпоративного гіганта та водночас представника великого бізнесу, Facebook. Також, може бути використана сумісно з Typescript, шляхом додавання додаткових пакетів із типами.

Сьогодні популярність React перевершила популярність більшості інших фреймворків, які використовувалися раніше для розробки компонентів користувацького інтерфейсу через низку релевантних причин.

На відміну від інших фреймворків, можна відмітити спрощений шлях створення динамічних та інтерактивних візуальних систем. Бібліотека React полегшує створення веб-додатків, оскільки для досягнення результативного минимума потрібно значно менше кількості часу виділеного під написання коду, також пропонується більше функціональних можливостей аніж JavaScript, коли складність розробки зростає дуже швидко.

При веб-розробці, на відміну від типу взаємодії, який присутній при використанні мови програмування JavaScript, коли стандартно методи викликаються над DOM. Для прискорення розробленого застосунку, бібліотека React вирішує додати свій Virtual DOM над яким і проводиться подальше оперування. Завдяки цьому, зростають показники продуктивності. Додаткова віртуалізація DOM відрізняється збільшеною швидкістю через особливий алгоритм порівняння поточних станів компонентів із попередніми, тобто збереженими їх нащадками, та оновлює лише ті елементи в справжньому DOM, які були змінені, замість звичної поведінки коли знову оновлюються всі компоненти дерева без встановленої послідовності, як цей механізм функціонує в роботі звичайних веб-додатків.

Ментальна модель розробки із використанням React є розбиття інтерфейсу на компоненти. Компоненти є будівельними блоками будь-якого додатку React та впроваджує ідеологію перевикористання створених компонентів, що значно скорочує час розробки програми та робить її уніфікованою. Найчастіше, розробники приходять до практики розділення компонентів на презентативні та компоненти, що містять бізнес-логіку. Цей підхід робить додаток зручним у тестуванні та розширює можливості перевикористання створених компонентів.

Бібліотека React слідує та знаходить активне застосування ідеї односпрямованого потоку даних. Це означає, що при розробці сервісу,

розробник створює батьківський компонент-обгортку над вкладеними в нього дочірніми компонентами. Оскільки дані слідують одному напрямку із зв’язком типу предок-нащадок, то стає легше визначати місця утворення помилок.

Бібліотека React є проста у вивченні та має не глибоку криву навчання, що ідеально підійде для новачків в сфері веб-розробки. React легко навчитися, оскільки він переважно поєднує базові концепції HTML і JavaScript з деякими корисними доповненнями.

Платформа, під яку розробляється застосунок, не обмежується тільки веб-середовищем, оскільки похідний інструмент від React є фреймворк React Native, що має схожу популярність, підтримку спільнотою та активно використовується для розробки під мобільні пристрої. Таким чином, насправді, React можна використовувати для додатків під різні платформи пристроїв.

Існують розширюючі інструменти-додатки, які інтегруються в браузер, як наприклад Chrome, та дозволяють відслідковувати помилки, які виникли під час розробки, в реальному часі.

3.5 Фреймворк Express

Express - це мінімалістичний і гнучкий фреймворк, створюючи серверні додатки для запуску у середовищі виконання програмного коду Node.js. Цей інструмент надає надійний набір функцій, необхідний для розробки REST API незалежно від платформи кінцевого користувача, веб чи мобільної.

Завдяки безлічі допоміжних методів протоколу HTTP і проміжного програмного забезпечення у розпорядженні розробника, створення надійного API є швидким і, відносно інших фреймворків, нескладним шляхом.

Express створює рівень абстракції виконання базових методів CRUD (від англ. аббр. Create - створити, Read - прочитати, Update - оновити, Delete

- видалити), не відмовляючись від вбудованих методів NodeJS. Також, багато інших фреймворків засновані на Express.

Можна перерахувати деякі особливості цього фреймворку для розробки серверної частини застосунку компетентним, а вибір його у стеку обгрунтованим.

- просте налаштування та конфігурація. Мінімальні вимоги запуску серверу на Express потребує лише середовище виконання NodeJS;
- дозволяє визначати маршрути доступу до API серверу на основі базових стандартизованих методів із переліку протоколу HTTP, та на основі URL-адрес;
- для екосистеми Express існують різні модулі проміжного програмного забезпечення, які можна використовувати для виконання додаткової обробки між запитом і відповіддю, оперуючи цими параметрами;
- легко інтегрувати з різними механізмами шаблонів, для надання візуальної частини програмного інтерфейсу;
- дозволяє визначити проміжне програмне забезпечення для обробки помилок, в тому числі і їх логування чи збереження тощо;
- не становить складності необхідність взаємодії з статичними файлами та ресурсами створеного додатку;
- найчастіше використовується для розробки серверу типу REST API;
- легко підключатися до таких баз даних, як MongoDB, Redis, MySQL.

Для використання фреймворку Express потрібен інструмент для встановлення пакетних модулів NPM. Після цього етапу, розробник повинен переконатися у доступності встановленого NodeJS для успішного запуску на вашому комп’ютері та подальшого виконання.

Проміжне програмне забезпечення - це функції, які мають доступ до об’єктів запиту та відповіді. Ці методи можуть виконувати будь-який довільний код, запускаються послідовно один за одним відповідно до

порядку розміщення проміжного програмного забезпечення. Проміжне програмне забезпечення є частиною програми, яка додає певну функціональність до створеного, в даному випадку серверного додатку. Проміжне програмне забезпечення не повинно змінювати хід роботи програми, тому додаючи, чи видаленні його із порядку виконання, не потребує особливих змін та налаштувань.

3.6 Хостинг Netlify

Netlify - це компанія з хмарних обчислень, яка знаходиться в Сан-Франциско,. Вона пропонує хостинг і безсерверні послуги для веб-додатків і статичних веб-сайтів. Позиціонується як швидка та стійка мережа для веб-застосунків. Безсерверна архітектура являється способом для компаній створювати і запускати програми без необхідності керування внутрішньою інфраструктурою. Це надає спосіб вилучити з робочого навантаження обов'язки щодо розгортання та підтримки архітектури, включаючи ревізію обладнання, масштабування та обслуговування. Масштабування може бути автоматичним, тобто платити треба лише за те, що використовується безпосередньо в системі. В хостингу Netlify, в свою чергу, доступ до кожного розгорнутого додатку надається в усіх глобальних та периферійних локаціях. Власні веб-сервери та CDN більше не потрібні. Netlify працює напряму через репозиторій розробника на GitHub і запускає процес збирання додатку із користувацькою конфігурацією необхідною для цього. Серві Netlify створює власний репозиторій, який пересилає на Github і власні мікросервіси. Потім він виконує, а надалі і поширює вміст по CDN, щоб надати відвідувачам попередньо зібрані статичні веб-сайти.

Найкраща риса Netlify є те, що хостинг обирає найкращий CDN, який вважає більш сприятним, враховуючи географічне розташування кінцевого користувача, і поширює контент.

Статично-згенерована веб-сторінка у поєднанні з цим підходом завантажується швидше, ніж у традиційних хостингових мережах. Також, у Netlify присутній механізм кешування статичних файлів, який можна налаштувати власноруч, та за замовчуванням, замість того, щоб завантажувати цілий сайт під час кожного відвідування, відвідувач отримує попередньо завантажену версію напряму з найближчого сервера. Це різко скорочує час до першого завантаження сайту користувачем.

Netlify має вбудоване керування DNS і сертифікатами SSL. Сертифікати надаються автоматично та безкоштовно, без обмеженням за строком дії, означає меншу складність у запуску веб-додатку.

Netlify дає змогу швидше налаштувати програмне забезпечення та частіше поставляти кращі продукти за допомогою сучасних робочих процесів Netlify для розробників. Його можна використовувати для створення веб-додатку незалежно від використаних при розробці фреймворків чи бібліотек.

Ще одна із компетентних особливостей цього сервісу, це додаткова націленість на комфорт розробки додатків технічним спеціалістам.

Щоразу, коли редактор хоче внести оновлення на сайт, він дає змогу створювати необмежену кількість гілок свого веб-сайту. Кожну окрема зміну вмісту вихідного коду у сховищі Github можна переглядати в різних середовищах. Тобто, за замовчуванням, Netlify відслідковує зміни у всьому репозиторії, головній гілці та створених додаткових. Надаючи редакторам контенту можливість переглядати зміни у реальному часі, не порушуючи відповідний процес розробникам.

3.6.1 Netlify Functions

Netlify Functions - це файли, які можуть бути написані, використовуючи один із декількох підтримуваних сервісом мов програмування, це JavaScript, TypeScript або Go, а потім створені та заповнені за відповідною конвенцією

Netlify та дотримуючись синтаксису обраної мови програмування, розміщуються у проєкті під шляхом, коригованим самим користувачем.

Створені функції виконуються синхронно або асинхронно з максимально можливим часом виконання десять секунд. Також можливе підвищення часу виконання та перетворення їх у функції, які мають тривалість до п’ятнадцяти хвилин та працюють в фоновому режимі.

Функції Netlify запускаються у Netlify середовищі, тому наслідують його особливості, переваги та недоліки. Можна виділити перевагу їх використання у практиці, це розгортання серверних методів, який потребує виділених ресурсів, та візуального інтерфейсу разом. Такий підхід прибирає необхідність у додатковому обслуговуванні хостингу серверної частини та відокремленого розгортання від лицевої частини додатку. Це дозволяє зберігати код, моніторити змінювати, та мати доступ до посилання для попереднього перегляду веб-застосунку.

3.7 Протокол HTTP та REST API

REST (від англ. Representational State Transfer - “передача репрезентативного стану”) - це архітектурний стиль розробки програмного забезпечення, який був створений для керівництва проєктування та розробки всесвітньої мережі. Із самого початку, цей термін був введений Роєм Філдингом. REST визначає набір обмежень та вимог до того, як повинна вести себе архітектура розподіленої гіпермедійної системи Інтернет. Процес переходу за вказаним URL в пошуковому полі браузеру є прикладом комунікації із сервером. Розглядаючи більш детально цей приклад, серверу відправляється запит на отримання контенту веб-сайту, який ідентифікується URL-адресою.

Потім цей сервер формує і видає відповідь. Важливим є формат цих запитів і відповідей. Цей формат відповідає протоколу HTTP (від англ. Hyper Text Transfer Protocol - “протокол передачі гіпертексту”).

При вказанні URL-адреси в браузері, він відправляє запит GET - отримати, на відповідний до цієї адреси сервер. Потім сервер відповідає на деякий запит відповіддю формату HTTP, який містить дані у форматі HTML. Потім браузер отримує цей HTML-код і відображає його відповідно до налаштувань користувача.

Щоб отримати необхідне уявлення про один з інших методів, присутніх REST API. Розглянемо простий приклад, коли користувач на сайті заповнює форму, які зазвичай присутня у загальному списку елементів сайту, у випадку необхідності збору деякої інформації. У такому випадку, коли користувач натискає кнопку Відправити, формує HTTP-запит POST із введеними даними на відправляється на сервер, де відбувається подальша обробка, збереження тощо.

REST є сервісом без заданого стандарту, тобто розробник сам обирає шлях використання та застосування кращих практик і підходів для цього. Він є гнучким у визначенні, і якщо розроблений серверний сервіс не потребує обробки даних від користувача, то може бути розглянутий сценарій, коли немає POST запиту, а лише GET, чи навпаки. Стандарт запитів, які приймає REST API і відповідей, задається індивідуально до конкретного випадку. Тобто, стандарт формується контекстом, що ще раз підкреслює гнучкість архітектури. Для створення відповідної документації зі стандарту API, найчастіше використовується релевантний до цього інструмент Swagger.

HTTP визначає наступну структуру запиту:

- строка запита (або рядок запиту) - визначає тип відправляємого повідомлення;
- заголовки запиту (поля заголовку) - характеризують параметри передачі та іншу додаткову інформацію повідомлення;
- “тіло” повідомлення. Цей пункт зазвичай є необов’язковим;

Протокол HTTP визначає наступну структуру відповідного повідомлення (відповідь):

- строка стану (або рядок стану). Включає код стану із переліку кодів відповідей HTTP та повідомлення про причини;
- поля заголовка відповіді та додаткове тело повідомлення.

3.8 Git

Git (від англ. Global Information Tracker - “глобальний трекер інформації”) - одна із існуючих систем контролю версій, являється безкоштовним програмним забезпеченням із відкритим вихідним кодом для відстеження змін у будь-якому наборі файлів. Це програмне забезпечення зазвичай використовується для синхронізації координування роботи розробників, які займаються спільним написанням вихідного коду будь-якого програмного застосунку.

Ідея Git полягає у тому, щоб підвищити швидкість розробки, зберегти цілісність даних і надати підтримку розподілених, нелінійних робочих процесів (це може бути багато гілок відокремлених від оригінальної, на яких одночасно паралельно ведеться розробка нових особливостей конкретної системи або виправлення недоліків тощо). На відміну від більшості інших схожих систем, у Git кожен каталог коду на всіх індивідуальних комп’ютерах розробників є повноцінним сховищем зі збереженою глобальною історією та можливостями відстеження попередніх версій. Це можна зробити як онлайн, так і без належного доступу до Інтернету чи серверу, зберігаючи на своїй робочій машині, а при підключенні до мережі, додавши нову до існуючої цілісної історії версій.

Накопичений досвід Лінуса Торвальдса з Linux, коли вуін мав змогу приймати безпосередню участь у розробці та підтримці великого розподіленого проєкту та його поглибленні знання про швидкість файлової

системи операційної системи, отриманими від того ж проєкту, і нагальною потребою створити робочу систему в короткі терміни. Ці ситуації вплинули на впровадження подальших до наступних варіантів реалізації:

- потужна підтримка нелінійного розвитку. Найважливіша особливість Git це підтримка розгалуження на індивідуальні гілки та їх злиття між собою. Також, для зручності розробникам, включає спеціальні інструменти для візуального відображення нелінійної історії розвитку та навігації. Гілки в середовищі Git майже нічого не важать, адже за своєю природою є лише посиланням на конкретну точку в історії розробки;
- розподілений розвиток. Git надає кожному розробнику локальну копію повної історії розробки, а зміни переносяться і однієї такої копії репозиторію в іншу;
- ефективна робота з великими проєктами. Розробник системи додав описання інструменту Git як швидкому та масштабованому, а проведені заміри швидкодії доказали високу продуктивність системи. В свою чергу, отримання збереженої історії версій з локального репозиторію може досягати значної швидкості у порівнянні з віддаленого;
- криптографічна аутентифікація історії. Історія Git зберігається таким чином, що ідентифікатор конкретної версії (коміт у термінах Git) залежить від повної історії розробки, яка вела до цього коміту. Після публікації неможливо змінити старі версії без того, щоб це було непомічено;

3.9 Axios

Axios - це HTTP-клієнт, відповідно як для роботи у браузері так і для NodeJS. Ця бібліотека спрощує спілкування з REST API при розробці серверу чи користувацького інтерфейсу. Надає додаткові можливості при написанні викликів стандартних методів, присутніх протоколу HTTP.

Хоча цього також можна досягти за допомогою інших методів, fetch або AJAX, які по замовчуванню вбудовані в більшість браузерів, але Axios надає більше функціональних можливостей, які дуже корисні для програм, в особливості при розробці користувацького інтерфейсу.

Порівнюючи зі стандартною специфікацією запиту-відповіді REST API, базованому на протоколі HTTP. Сигнатура об’єкту відповіді бібліотеки AXIOS виглядає наступним чином:

- отримані дані - це корисна частина відповіді, що повертається сервером. За замовчуванням, бібліотека очікує отримання формату JSON від серверу і трансформує його у об’єкт-примітив мови програмування JavaScript;
- статус - це код інформування відповіді за стандартом HTTP, який повертається сервером;
- текст статусу - посилається на повідомлення про статус HTTP, яке повертає сервер, мета-інформація;
- заголовки. Поміщені сервером усі заголовки, які стосуються конкретної відповіді;
- конфігурація. Містить оригінальну конфігурацію запиту, створену розробником під час використання бібліотеки;
- запит - містить посилання на оригінальний об'єкт XMLHttpRequest.

Висновки до розділу 3

В розділі було проведено вибір технологій та сучасних засобів розробки для програмної реалізації веб-додатку. Усі обрані технології та засоби є актуальними на момент написання дослідницького проєкту, постійно підтримуються та оновлюються розробниками відповідних сервісів.

Для програмної реалізації додатку було обрано середовище Visual Studio Code єдиної доступної безкоштовної версії. В середовищі присутня можливість створення додатків будь-якою мовою програмування, в конкретному випадку на мові Typescript із використанням фреймворку

Express з архітектурою REST API, бібліотекою React та хостинговим рішенням Netlify.

Крім того, для Express є багато додаткових пакетних модулів для серверної комунікації з NoSQL базою даних MongoDB, також має вбудовані методи серверної розробки, успадкований із NodeJS. Безсерверне рішення хостингу, який надає Netlify дозволяє зекономити час та ресурси для розробки та розгортання веб-додатків. У Visual Studio Code присутня вбудована можливість переглядання архітектури проєкту, зручна навігація та додатки, які допоможуть при розробці відповідного веб-сервісу.

Суттєвим недоліком обраних рішень є те, що повинні бути практичні знання для швидкого розгортання та розробки програмного забезпечення. Обрані технології не потребують великих ресурсів швидкості зчитування та запису даних, але обсяг оперативної пам’яті повинен бути достатнім, що може суттєво уповільнити процес розробки. Розроблений веб-застосунок запускається кінцевим користувачем у браузері, тому запуск може бути здійснений на обчислювальних машинах із малими потужностями. Особливою перевагою є те, що усі обрані технології, поширюються безкоштовно.

4 ПРОЄКТУВАННЯ ТА РЕАЛІЗАЦІЯ ЗАСТОСУНКУ

Після виконання аналізу предметної області було визначено основні сутності та види зв’язків між ними. Нижче представлено таблиці бази даних, що представляють ці сутності та зв’язки у структурованому вигляді. Опис сутності User показано в таблиці 4.1.

Таблиця 4.1 – Опис сутності User

Поле	Тип	Опис
_id	ObjectId	Унікальний ідентифікатор, що присвоюється автоматично при створенні нового документу в Mongo базі даних
name	String	Атрибут, який містить ім’я користувача. Може повторюватися із вже існуючими записами
surname	String	Атрибут, який містить прізвище користувача. Може повторюватися із вже існуючими записами
email	String	Атрибут, який містить електронну пошту користувача. Не може повторюватися із вже існуючими записами. Повинен бути унікальним
password	String	Атрибут, який містить пароль для аутентифікації користувача. Може повторюватися із вже існуючими записами
templates	Array<ObjectId>	Атрибут, який містить масив унікальних ідентифікаторів, які відносяться, як головні ключі шаблонів. Не може повторюватися із вже існуючими записами

Масив унікальних ідентифікаторів [Templates], які знаходяться в попередньо описаній таблиці, можуть використовуватися для заповнення даними відповідних шаблонів. Опис сутності Template представлено в таблиці 4.2.

Таблиця 4.2 – Опис сутності Template

Поле	Тип	Опис
_id	ObjectId	Унікальний ідентифікатор, що є первинним ключем таблиці
name	String	Атрибут, який містить назву шаблону. Може повторюватися із вже існуючими записами.
author	String	Атрибут, який містить ім’я або нікнейм автора. Може повторюватися із вже існуючими записами.
version	String	Атрибут, який містить версію шаблону. Може повторюватися із вже існуючими записами.
packages	Array<ObjectId>	Атрибут, який містить масив унікальних ідентифікаторів, які відносяться, як головні ключи пакетів. Може повторюватися із вже існуючими записами

Масив унікальних ідентифікаторів [Packages], які знаходяться в попередньо описаній таблиці, можуть використовуватися для заповнення даними відповідних пакетів із атрибутами, вказаних нижче. Опис сутності Package представлено в таблиці 4.3.

Таблиця 4.3 – Опис сутності Package

Поле	Тип	Опис
_id	ObjectId	Унікальний ідентифікатор, що є первинним ключем таблиці
name	String	Атрибут, який містить назву пакету. Може повторюватися із вже існуючими записами.
version	String	Атрибут, який містить версію пакету. Може повторюватися із вже існуючими записами.
description	String	Атрибут, який містить опис пакету. Може повторюватися із вже існуючими записами.
date	Date	Атрибут, який містить дату останньої публікації. Може повторюватися із вже існуючими записами.
link	String	Атрибут, який містить посилання до ресурсу із більш детальнішою інформацією про відповідний пакетний модуль. Може повторюватися із вже існуючими записами.
starsCount	Number	Атрибут, який містить кількість зірочок, які відображають вподобання спільноти до конкретного пакету. Може повторюватися із вже існуючими записами.

dependencies	Map<Object>	Атрибут, який містить пакетні залежності та їх версії, які відносяться до конкретного модуля. Може повторюватися із вже існуючими записами.
quality	qualitySchema	Атрибут, який містить схему із показниками якості пакету. Може повторюватися із вже існуючими записами.
popularity	popularitySchema	Атрибут, який містить схему із показниками популярності пакету. Може повторюватися із вже існуючими записами.
maintenance	maintenanceSchema	Атрибут, який містить схему із показниками підтримки пакету. Може повторюватися із вже існуючими записами.
score	scoreSchema	Атрибут, який містить схему із середніми показниками пакету, описаних вище. Може повторюватися із вже існуючими записами.

Деякі атрибути, вказані в таблиці 4.3, а саме ті, що містять постфікс “Schema”, мають посилання на сигнатуру однієї з допоміжних схем. Опис допоміжної схеми qualitySchema представлено в таблиці 4.4.

Таблиця 4.4 – Опис схеми QualitySchema

Поле	Тип	Опис
carefulness	Number	Атрибут, при розрахунку якого, враховується присутність наступних елементів в пакеті з різною вагою для кожного з показників: ліцензія (0,33), README (0,38), журнал змін

		(0,08), використання статичних аналізаторів коду на помилки (0,13), а також у пакеті є файл з наявним ігноруванням внутрішніх пакетів чи присутніх файлів. Може повторюватися із вже існуючими записами.
tests	Number	Атрибут, при розрахунку якого оцінюються три параметри: наявність тестових файлів (0,6), охоплення (0,15) і статуси Github, серед яких можливий невдалий коміт, або навпаки успішний (0,25). Може повторюватися із вже існуючими записами.
health	Number	Атрибут, при розрахунку якого здоров'є пакету корелюється із кількістю застарілих залежностей і наявність вразливостей для пакета. Якщо всі залежності оновлені і немає відомих вразливостей, пакет отримує повну оцінку. Може повторюватися із вже існуючими записами.
branding	Number	Атрибут, при розрахунку якого враховуються ваги двох параметрів: наявність домашньої сторінки проєкту (0,4) і кількість значків-бейджів, яку проєкт має на README (0,6), це можуть бути бейджи із посиланням на зовнішний ресурс, наприклад, про успішне розгортання на Netlify. Може повторюватися із вже існуючими записами.

Опис допоміжної схеми popularitySchema представлено в таблиці 4.5.

Таблиця 4.5 – Опис схеми PopularitySchema

Поле	Тип	Опис
communityInterest	Number	Атрибут, який містить зацікавленість спільноти. Може повторюватися із вже існуючими записами.
downloadsCount	Number	Атрибут, який містить кількість завантажень. Може повторюватися із вже існуючими записами.
downloadsAcceleration	Number	Атрибут, який містить динаміку зростання кількості завантажень. Може повторюватися із вже існуючими записами.

Опис допоміжної схеми MaintenanceSchema представлено в таблиці 4.6.

Таблиця 4.6 – Опис схеми MaintenanceSchema

Поле	Тип	Опис
releasesFrequency	Number	Атрибут, який містить частоту випуску пакету. Може повторюватися із вже існуючими записами.
commitsFrequency	Number	Атрибут, який містить частоту комітів до репозиторію. Може повторюватися із вже існуючими записами.
openIssues	Number	Атрибут, який містить кількість відкритих зауважень чи питань з

		працездатності пакету. Може повторюватися із вже існуючими записами.

Опис схеми scoreSchema представлено в таблиці 4.7.

Таблиця 4.7 – Опис схеми scoreSchema

Поле	Тип	Опис
final	Number	Атрибут, який містить середнє значення показників, вказаних у схемі нижче. Може повторюватися із вже існуючими записами.
detail	scoreDetailSchema	Атрибут, який містить схему із середніми значеннями показників пакету, вказаних у допоміжних схемах вище. Може повторюватися із вже існуючими записами.

Деякі атрибути, вказані в таблиці 4.7, а саме [detail] має посилання на сигнатуру однієї з допоміжних схем. Опис допоміжної схеми scoreDetailSchema представлено в таблиці 4.8.

Таблиця 4.8 – Опис схеми ScoreDetailSchema

Поле	Тип	Опис
quality	Number	Атрибут, який містить середній показник значень, перерахованих та описаних в таблиці 4.4. Може повторюватися із вже існуючими записами.

popularity	Number	Атрибут, який містить середній показник значень, перерахованих та описаних в таблиці 4.5. Може повторюватися із вже існуючими записами.
maintenance	Number	Атрибут, який містить середній показник значень, перерахованих та описаних в таблиці 4.6. Може повторюватися із вже існуючими записами.

4.1 Діаграма варіантів використання мовою UML

Діаграми варіантів використання потрібні для опису взаємозв'язків і залежностей між групами варіантів використання та діючими особами, що беруть участь у процесі. Також важливо мати на увазі, що схеми варіантів використання не призначені для відображення проєкту в цілому, та не можуть описувати внутрішні системи проєктованої системи.

Діаграми варіантів використання призначені для поглиблення взаємодії з майбутніми користувачами розробляємої системи. Іншими словами, діаграми варіантів використання потрібні для висловлення думки про те, що система повинна робити, не вказуючи методи, що застосовуються. Діюча особа (актор) не є безпосереднім елементом системи, а навпаки, є зовнішнім джерелом, тобто взаємодіє із системою через деякий варіант використання. В ролі діючої особи можуть бути представлені реальні люди (наприклад, кінцеві користувачі системи), а також безліч інших прикладів, описані комп'ютерними системами або зовнішніми подіями.

Важливо розуміти те, що діючі особи уособлюють не конкретно фізичних людей або системи, а обмежуються лише їх ролями. Це означає, що коли людина взаємодіє із системою різними способами (для цього

передбачається використання різних ролей), вона відображається декількома діючими особами.

Варіант використання описує, з точки зору актора, групу дій у системі, яка призводить до конкретного результату наприкінці дії.

Найчастіше, варіанти використання описані типовими взаємодіями, які можуть трапитися у сумісній взаємодії між користувачами системи та безпосередньо самою системою. Вони відображають зовнішній інтерфейс системи та визначають форму того, що саме проєктована система повинна робити у відповідь на дію.

Кожен варіант використання має ініціатора, тобто відноситься як мінімум до одного діючого актора, та призводить до відповідного кінцевого результату. Варіанти використання також можуть взаємодіяти з іншими варіантами використання. В рамках дослідницького проєкту була розроблена діаграма варіантів використання, що наведена в документі ІА82.210БАК.003 Д4. Основними ролями на діаграмі виступають користувач та розроблена система.

Після процесу реєстрації користувач може авторизуватись в системі та переглядати створені попередньо шаблони веб-застосунків. Також користувач має можливість створити новий шаблон, вказати програмні залежності шаблону, попередньо надавши мета-інформацію про створений шаблон. Користувач також має можливість змінювати інформацію про папкову архітектуру шаблону, включаючи:

- зміну створених папок;
- зміну вкладеності папок;
- обрання готової структури папок.

Користувач має можливість додавання модульних залежностей до проєкту, змінювати їх версії, видаляти їх, підтверджувати додавання пакетів при несумісності їх версій. Користувач також може редагувати інформацію про кінцевий створений шаблон, зберігати їх віддалено та завантажувати локально.

4.2 Моделі

Моделі за своєю суттю являються описом деякої структури даних, що використовується, та логікою взаємодії цих даних. Всі моделі створюються як окремі класи мовою програмування TypeScript. У моделях не описується логіка поведінки додатку, а лише зберігаються стани сутностей у вигляді властивостей.

Моделі не залежать від представлення, ані від бізнес-логіки, що дозволяє розробляти їх незалежно, а також створювати необмежену кількість представлень для конкретної моделі. В моделі також відбувається валідація вихідних даних, що отримуються від користувача запитом із користувацького інтерфейсу застосунку.

На рисунку 4.1 зображено код класу User, що являє собою модель користувача.

functions > db > TS User.ts > ...

```ts
import mongoose from 'mongoose'
import { IUserEntity } from './types'

const userSchema = new mongoose.Schema({
  name: {
    type: String,
    required: [true, "Ім'я - обов'язкове поле для заповнення"],
    minlength: [2, "Ім'я має містити більше 2-х символів"],
    maxlength: [32, "Ім'я має містити не більше 32-х символів"],
  },
  surname: {
    type: String,
    required: [true, "Прізвище - обов'язкове поле для заповнення"],
    minlength: [2, 'Прізвище має містити більше 2-х символів'],
    maxlength: [64, 'Прізвище має містити не більше 64-х символів'],
  },
  email: {
    type: String,
    required: [true, "Електронна пошта - обов'язкове поле для заповнення"],
    validate: {
      validator: (value: string) => /^[a-z1-9]+@[a-z1-9]+\.[a-z1-9]+$/.test(value),
      message: 'Електронна пошта невірного формату',
    },
  },
  password: {
    type: String,
    required: [true, "Пароль - обов'язкове поле для заповнення"],
    minlength: [2, 'Пароль має містити більше 2-х символів'],
  },
  templates: {
    type: [mongoose.Schema.Types.ObjectId],
    ref: 'Template',
  },
})

/**
 * Модель сутності користувача
 */

export default mongoose.model<IUserEntity>('User', userSchema)
```

Рисунок 4.1 – Код класу User

Властивостями сутності конкретної моделі представлено основні атрибути сутності «Користувач», а саме: унікальний в межах бази даних MongoDB ідентифікатор документу ObjectId, ім’я Name, прізвище Surname, електронна пошта Email, пароль Password, а також Templates. Властивість Templates може містити масив даних, які є зовнішніми посиланням на відповідний ідентифікатору шаблон.

Кожне з полей моделі MongoDB може бути заданим обмеженою вибіркою деяких властивостей. Як наприклад, для конкретної моделі було використано тип поля [Type], флаг [Required] із текстом повідомлення який буде повернено у випадку пропуску поля, мінімальна кількість символів необхідна для поля [MinimumLength], максимальна кількість символів для вводу [MaximumLength].

Поле Email має безпрецедентну валідацію, а отже використовується специфічна функція validator, яка виконує булеве значення в залежності від валідності вхідних даних до схеми перевірки у формі регулярного виразу, який відображає базову перевірку електронної пошти, та message, що містить повідомлення у випадку непроходження даних на валідність.

Так як додаток розробляється з застосуванням строго типизованої мові програмування Typescript, яку Mongoose підтримує з початку встановлення без додаткових налаштувань.

Для зручності подальшої розробки, можна передати інтерфейс сутності користувача IUserEntity, який за своєю природою відображає існуючі дані моделі (рисунок 4.2). Всі інші моделі побудовані за схожим принципом і відрізняються лише назвою атрибутів, наявністю умов перевірки валідації даних.

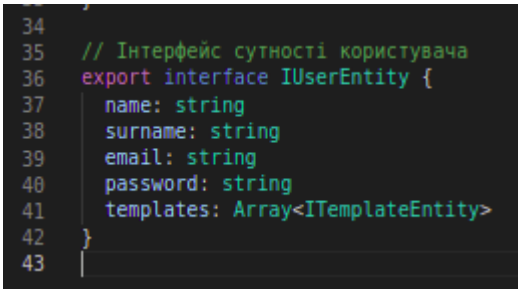

Рисунок 4.2 – Інтерфейс сутності моделі користувача

4.3 Попереднє налаштування представлення

Представлення відповідає за користувацький інтерфейс. Воно являється HTML-шаблоном, який повинен відображати елементи керування HTML із специфічною їм поведінкою, найчастіше, оперуючи із даними, отриманими з серверу.

Представлення використовується для надання користувачу можливої інтерактивності з системою, функціональності, відображення даних за допомогою екземпляру класу моделі.

Відповідні представлення зберігаються у папці проєкту під назвою Pages, які відповідають вигляду сторінки, коли користувач переходитиме за певною URL-адресою. Ці компоненти сторінок не передаються користувачу у браузері самі по собі. Отже, потрібно провести деяке попереднє налаштування навігаційної логіки.

Для налагодження навігаційних роутів, використаємо бібліотеку React-Router, що дозволить додавати шляхам компоненти-макети, що дозволять дотримуватися принципу програмування DRY (з англ. “Don’t repeat yourself” - “Не повторюйся”, принцип розробки програмного забезпечення, спрямований на зменшення повторюваності шаблонів програмування, використовуючи заміну їх абстракціями або використання нормалізації даних, щоб уникнути надмірності повторюваного коду). В ній також можна створювати так звані “динамічні роути”, при переході на які, компонент, який відповідає за представлення, отримує на вхід частину URL як параметри.

Для додавання обгорток до React застосунку будемо використовувати підхід створення контейнерів-обгорток, подальшому імпорті та застосуванні їх у головному файлі, що містить компонент із глобальною аплікацією (рисунок 4.3).

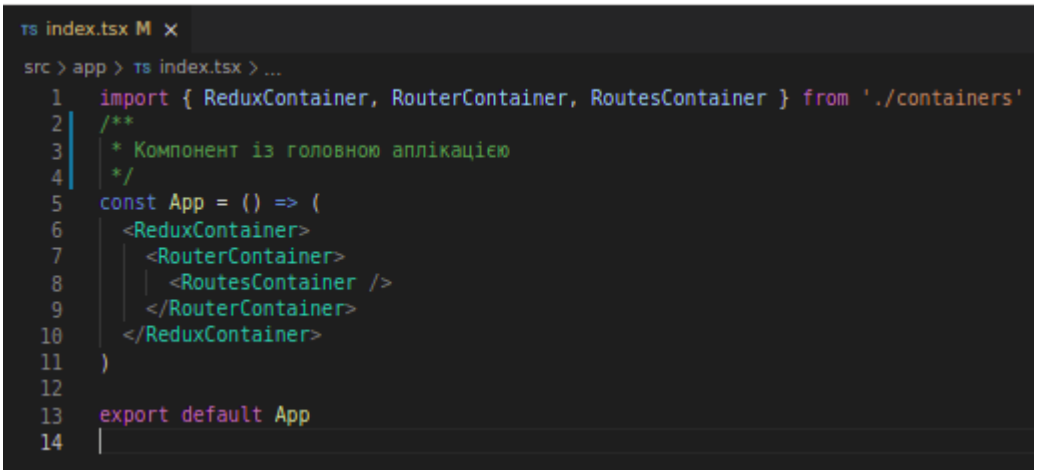

Рисунок 4.3 – Код класу App

При налаштуванні початкового етапу роботи з бібліотекою React-Router потрібно додати обгортку у вигляді компоненту BrowserRouter (рисунок 4.4).

Рисунок 4.4 – Код класу RouterContainer

Після цього, ми можемо використовувати весь функціонал бібліотеки React-Router, який надається за замовчуванням. В додатку також присутня

бібліотека React-Redux для створення та менеджменту глобального сховища на рівні всього додатку, а для комфорту користувача, додана бібліотека Redux-Persist, що надає можливість зберігання стану сховища Redux після оновлення сторінки користувачем.

А при завантаження сторінки, коли дані, які надходять із локального сховища браузеру, ще не були отримані та оброблені додатком, Redux-Persist має атрибут [loading], який приймає на вхід будь-який компонент, що буде відображатися під час загрузки даних.

Також, для інтеграції проміжного програмного забезпечення, яке буде перехоплювати дії, передані до обробників Redux, найчастіше, з асинхронним ходом виконання, використовується допоміжний метод запуску runSaga. Упорядкована частина коду створеного контейнера, яка за це відповідає, зображена на рисунку 4.5.

index.redux.tsx M

src > app > containers > index.redux.tsx > ...

```tsx
import initializeStore from '@/store'
import { RootState } from '@/store/index.reducer'
import { CircularProgress } from '@material-ui/core'
import { Provider } from 'react-redux'
import { Store } from 'redux'
import { PersistGate } from 'redux-persist/integration/react'

type Props = {
  children?: React.ReactNode
  store?: Store<RootState>
}
/**
 * Ініціалізуємо глобальний стейт додатку та допоміжні функції
 */
const { store: initialStore, runSaga, persistor } = initializeStore()
/**
 * Запускаємо обробку дій, переданих до Redux. проміжним програмним забезпеченням
 */
runSaga()
/**
 * Обгортка яка використовує Redux для глобального стейт-менеджменту,
 * та Redux-Persist для зберігання стану сховища після оновлення сторінки
 */
const ReduxContainer = ({ children, store = initialStore }: Props) => (
  <Provider store={store}>
    <PersistGate loading={<CircularProgress />} persistor={persistor}>
      {children}
    </PersistGate>
  </Provider>
)

export { ReduxContainer }
```

Рисунок 4.5 – Код класу ReduxContainer

Для того щоб було дотримано принципу програмування DRY, перевикористано загальну логіку представлень серед сторінок, а також для обробки того випадку, коли неавторизований користувач намагається перейти за шляхом із даними, можливими для перегляду лише після авторизації.

На рисунку 4.6 показано фрагмент коду представлення контейнеру, який описує можливі шляхи додатку та зіставляє їх із компонентами, необхідними для відображення конкретної частини застосунку, а також узагальнюючі макети-обгортки для деяких із них.

В контейнері використовується вкладена структура елементів, що дозволяє явно вказати належність типу “предок-дитина”. Компонент високого порядку Route, із React-Router, передає неявно параметри, присутні бібліотеці React-Router кожному елементу, використаному для відображення шляху.

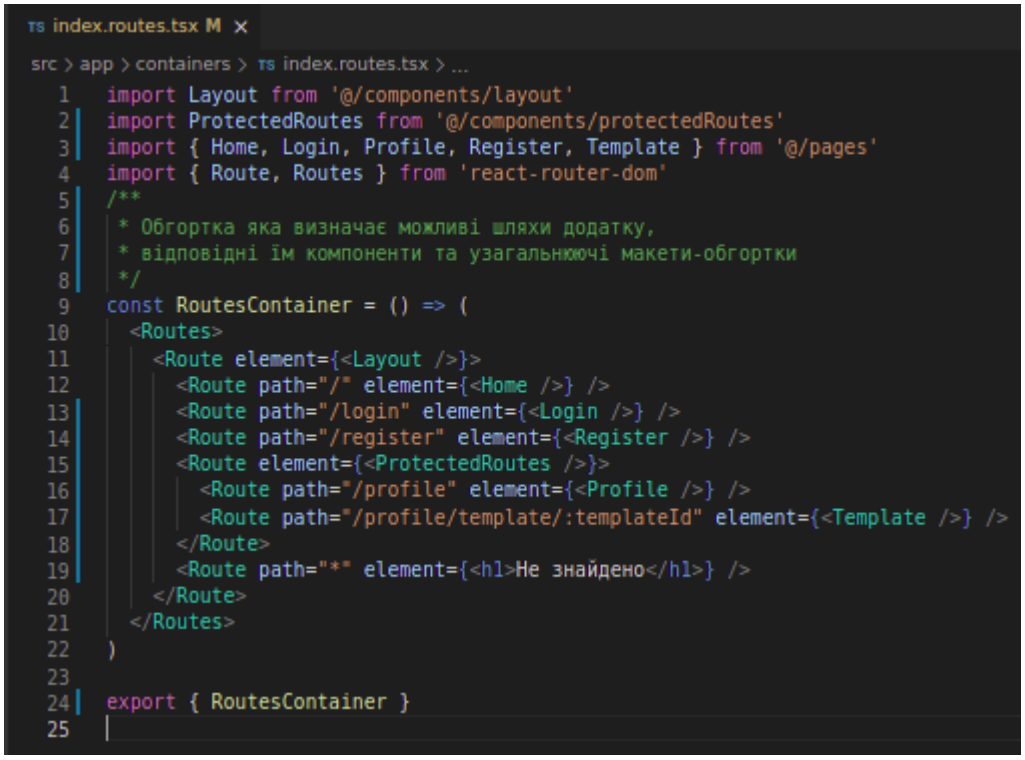

Рисунок 4.6 – Код класу RoutesContainer

Деякі шляхи, а саме шлях із особистим профілем користувача, або шлях для отримання інформації про конкретний шаблон, використовується макет-обгортка ProtectedRoutes, яка за суттю своєї реалізації робить тривіальну задачу, перевіряє поточний стан авторизації користувача.

У випадку невиконання умови, при якій користувач авторизован, вона робить перегляд візуального наповнення шляху неможливим, завдяки перенаправленню до шляху з формою авторизації та реєстрації.

Для вибіркового обрання порядку відображення вкладених роутів, можна обрати один із двох підходів.

Один із таких підходів це викликати функцію із бібліотеки React-Router всередині компоненту-макету та використати її вихідне значення в операторі повернення return.

Використаємо інший підхід, який відрізняється більш декларативним методом визову функції, а саме компонент Outlet із тієї самої бібліотеки, який потім буде замінено на вкладений компонент шляху. Фрагмент описаного коду представлено на рисунку 4.7.

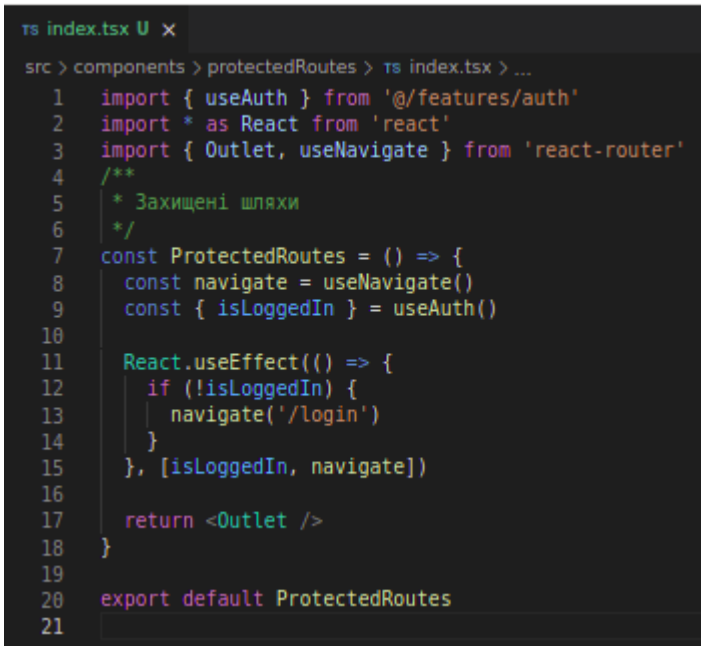

Рисунок 4.7 – Код класу ProtectedRoutes

Таким чином кодова база додатку має менше повторів логічних патернів, а тому легше читається, має єдину точку правди, яку можна легко змінити, не торкаючись релевантних частин коду, а також код набуває можливості перевикористання.

Для всіх інших переходів користувача, у випадку, коли URL не співпадає ні з одним із існуючих роутів, прописаних в контейнері RoutesContainer на рисунку 4.6, з'являється повідомлення про відсутність візуального результату за запитом.

Прикладом одного із представлень, а саме, головної сторінки, на якій можна побачити створення базового вікна з текстом привітання та варіантами вибору сторінки для переходу (рисунок 4.8).

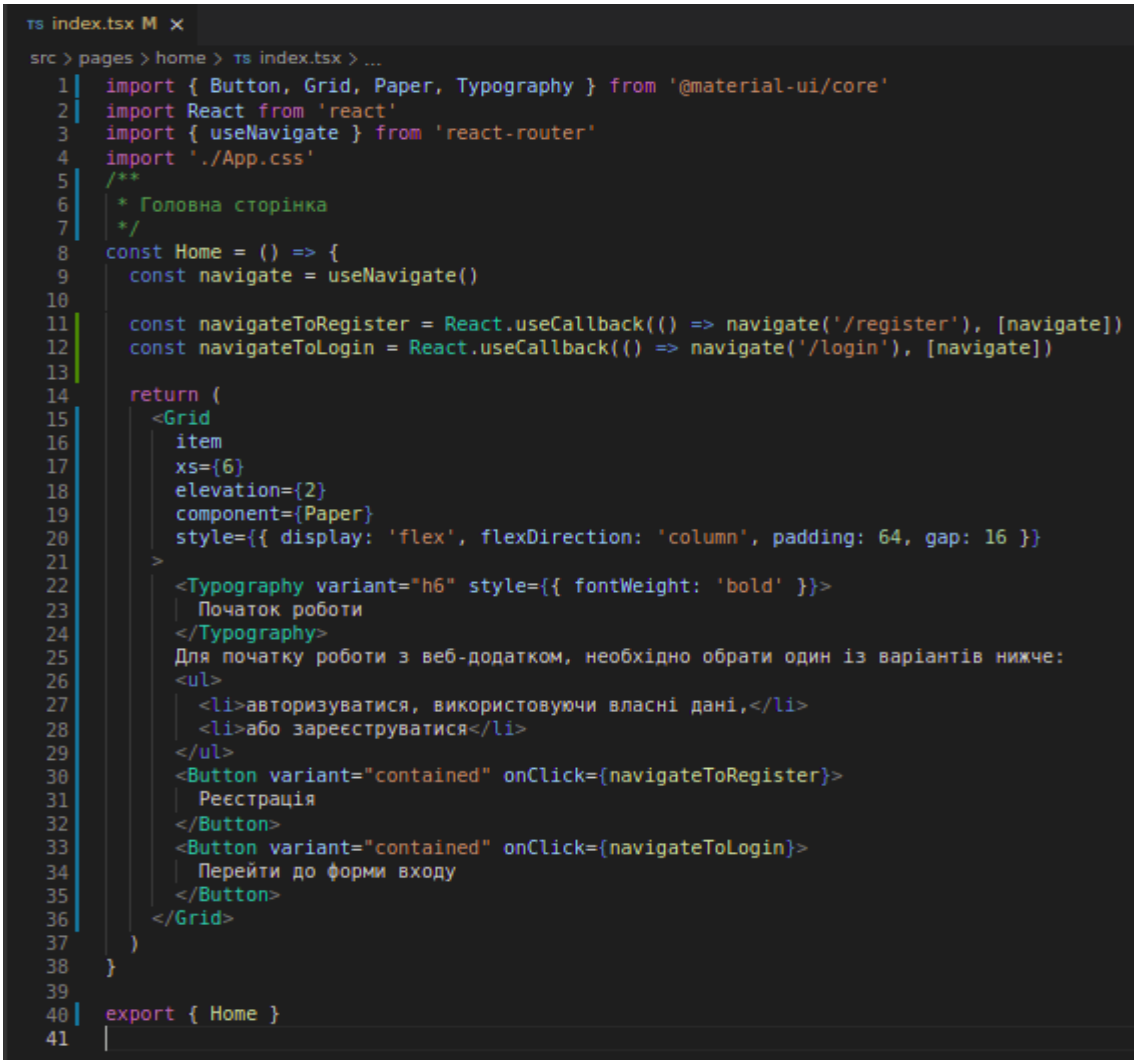

Рисунок 4.8 – Код класу Home

4.4 Редуктори

Редуктори є найважливішою концепцією Redux. Редуктор (його також можна назвати «функцією зменшення») - це функція, яка приймає акумулятор та значення, і повертає новий акумулятор. Вони використовуються для зменшення колекції значень до одного значення. Редуктори не є унікальними для Redux - вони є фундаментальною концепцією функціонального програмування. Навіть більшість мов, як-от JavaScript, мають вбудовані функції, подібні до зазначеної.

У Redux накопичене значення-акумулятор є об’єктом стану, а значення, що накопичуються, є діями. Редуктори обчислюють новий стан з урахуванням значення попереднього стану та переданої дії. Це повинні бути "чисті функції" - функції, які повертають однаковий вихід при заданні вхідних даних. Вони також не повинні мати побічних ефектів, тобто не повинні мати асинхронності тощо.

На рисунку 4.9 показано код об’єднуючого кореневого редуктора, який також містить конфігурацію Redux-Persist.

index.reducer.ts M

src > store > index.reducer.ts > ...

```ts
import { authReducer, templatesReducer } from '@/features'
import { combineReducers } from '@reduxjs/toolkit'
import { persistReducer } from 'redux-persist'
import storage from 'redux-persist/lib/storage'
/**
 * Конфігурація Redux-Persist
 */
const persistConfig = {
  key: 'root',
  storage,
}
/**
 * Загальний редуктор, який поєднує існуючі
 * та містить конфігурацію Redux-Persist
 */
const rootReducer = persistReducer(
  persistConfig,
  combineReducers({
    auth: authReducer,
    templates: templatesReducer,
  })
)

export type RootState = ReturnType<typeof rootReducer>

export default rootReducer
```

Рисунок 4.9 – Код класу RootReducer

Для поєднання React-Redux із Redux-Persist, використовується функція з бібліотеки, яка на вхід отримує об’єкт, що містить конфігурацію майбутнього стану, збереженого у локальному сховищі браузеру, а також комбінацію з усіх існуючих редукторів додатку. Один з яких редуктор authReducer, що відповідає за дії пов’язані з користувачем, а саме авторизацію, реєстрацію тощо, а також редуктор templatesReducer, який містить бізнес-логіку, відповідно до обробки шаблонів.

На рисунку 4.10 зображено приклад написання редуктора authReducer всередині допоміжної функції createSlice із пакету з набором корисних функцій для роботи з Redux під навою Redux-Toolkit, та зменшенню кількості коду, потрібного для досягнення певного результату, у разі їх відсутності.

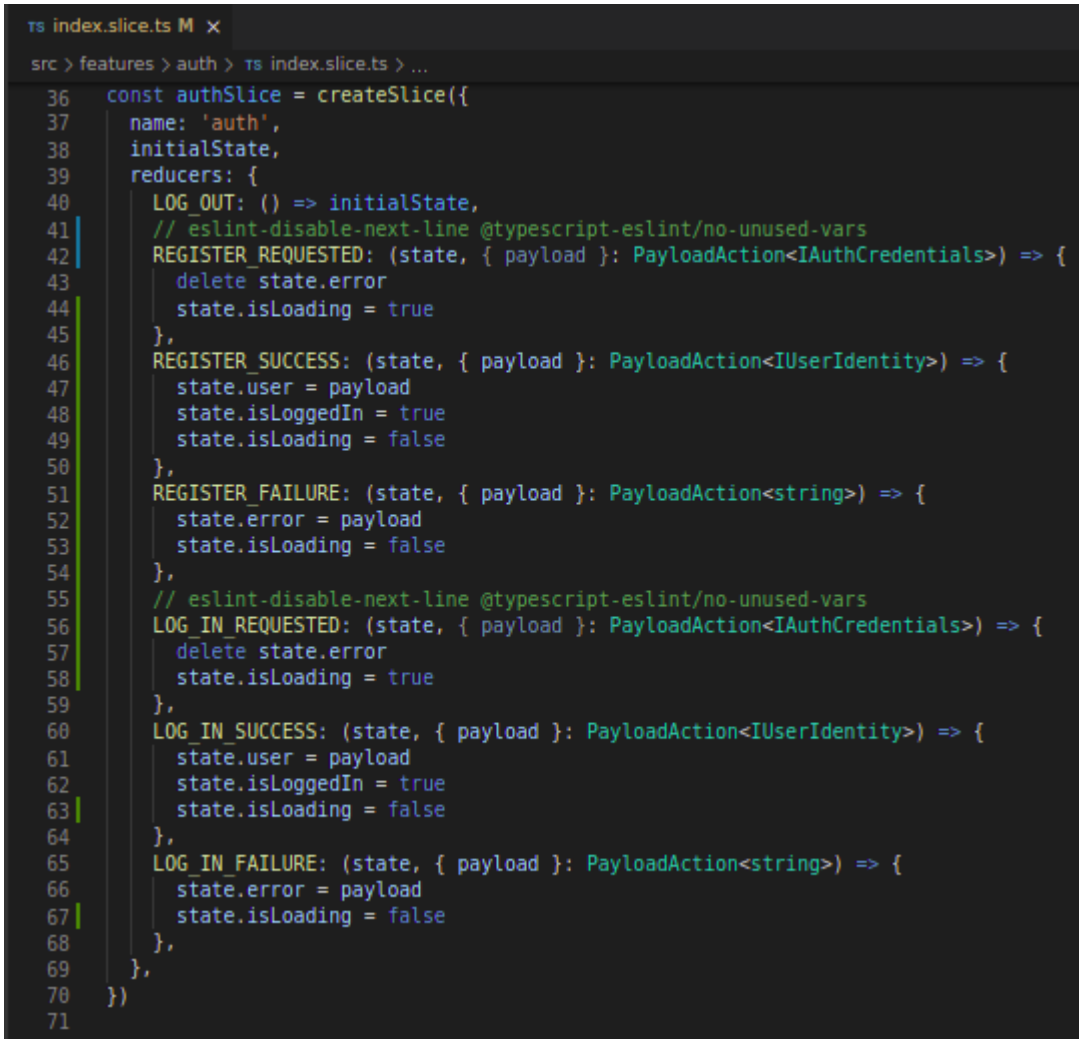

Рисунок 4.10 – Код класу AuthReducer

Всі “чисті функції” перераховані в атрибуті [reducers] та описані в таблиці 4.9.

Таблиця 4.9 – Редуктори AuthReducer

Назва	Опис
LOG_OUT	Повертає стан Auth до початкового.
REGISTER_REQUESTED	Редуктор, виклик якого буде перехоплений проміжним програмним забезпеченням Redux-Saga. Видаляє можливу існуючу помилку зі стану Redux та перемикає флаг із загрузкою isLoading у правдиву позицію.
REGISTER_SUCCESS	Редуктор, виклик якого ініціює встановлення стану змінної користувача до поточного. Перемикає флаг із загрузкою isLoading у неправдиву позицію, а флаг, що описує чи користувач авторизован isLoggedIn навпаки в правдиву.
REGISTER_FAILURE	Редуктор, виклик якого ініціює встановлення стану змінної помилки до поточного. Перемикає флаг із загрузкою isLoading у неправдиву позицію.
LOG_IN_REQUESTED	Редуктор, виклик якого буде перехоплений проміжним програмним забезпеченням Redux-Saga. Видаляє можливу існуючу помилку зі стану

	Redux та перемикає флаг із загрузкою isLoading у правдиву позицію.
LOG_IN_SUCCESS	Редуктор, виклик якого ініціює встановлення стану змінної користувача до поточного. Перемикає флаг із загрузкою isLoading у неправдиву позицію, а флаг, що описує чи користувач авторизован isLoggedIn навпаки в правдиву.
LOG_IN_FAILURE	Редуктор, виклик якого ініціює встановлення стану змінної помилки до поточного. Перемикає флаг із загрузкою isLoading у неправдиву позицію.

Корисна функція createSlice у своїй внутрішній реалізації використовує бібліотеку Immer. Immer можна використовувати в будь-якому контексті, в якому необхідно використовувати незмінні структури даних. Наприклад, у поєднанні з редукторами JavaScript, React або Redux

Незмінні структури даних дозволяють виявляти зміни ефективніше, у порівнянні зі змінними: якщо посилання на об’єкт не змінилося, сам об’єкт не змінився.

Крім того, це робить клонування об’єктів відносно незатратною операцією: незмінні частини дерева даних із вкладеною структурою не потрібно копіювати, адже можна спільно використовувати у пам’яті старіші версії того самого стану.

Завдяки цьому, можна не вдаватись у поглиблення знань про оперування незмінними даними та повторенні кодових патернів для їх

створення, адже редуктори, передані в атрибут [reducers] надають можливість прямої зміни Redux стану state без створення його копії.

4.5 Побічні ефекти Redux-Saga

Redux-Saga - це бібліотека, яка має на меті зробити побічні ефекти програмного застосунку (тобто асинхронні речі, як наприклад, отримання даних із серверу, або речі, які не можуть бути використані за ідеологією редукторів Redux, бо вони не відносяться до чистих функцій, як це буває з доступом до кешу браузера) легшими в управлінні, швидшими у виконанні, легкими при тестуванні та кращими при обробці помилок.

Ментальна модель полягає в тому, що сага - це як окремий потік у програмному забезпеченні, який відповідає виключно за побічні ефекти.

Redux-Saga - це проміжне програмне забезпечення для бібліотеки Redux, що означає, що цей потік можна запускати, призупиняти та скасовувати у регулярному потоці основної програми за допомогою звичайних дій Redux, воно має доступ до повного стану Redux, а також може самостійно передавати дії.

Для виконання асинхронного запиту до серверу, потрібна наявність деякого сервісу. Нижче представлено фрагмент коду із функціями авторизації та реєстрації, які у своїй реалізації використовують бібліотеку Axios та посилають POST-запити із вхідними даними у формі сигнатури користувача.

Наприклад, для авторизації потрібні атрибути [email] та [password]. Запити відправляються за шляхом, який обробляє активний сервер Express за спеціальними вузлами, прописаними на стороні серверу. Сервіс REST надає можливість знати про існування лише обмеженої кількості вузлів, які доступні для обробки. Логіка перелічених функцій для посилання запитів описана на рисунку 4.11.

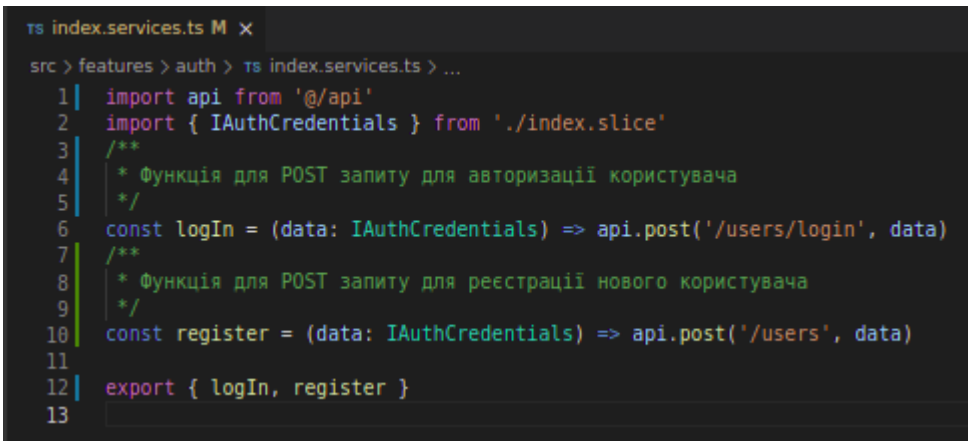

Рисунок 4.11 – Код класу AuthServices

Сага в свою чергу викликає створений сервіс та чекає на відповідь у асинхронному потоці, не перериваючи основну програму. Виклик сервісу всередині саги знаходиться в синтаксичній конструкції try…catch для його безпечного виконання.

У разі отримання позитивної відповіді зі сторони серверу, сага відправляє дію з даними, що свідчить про успішне відпрацювання функції.

У випадку, коли під час відправлення сервісом запиту трапилася будь-яка помилка, чи то зі сторони серверу, чи з клієнтської сторони, отримана помилка аналогічно до попереднього процесу, передає дію з повідомленням помилки, для зберігання в загальному сховищі програми (рисунок 4.12).

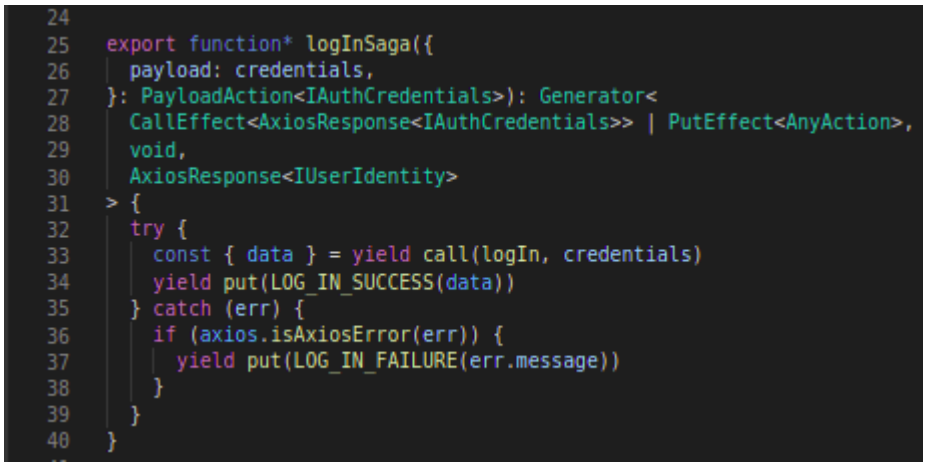

Рисунок 4.12 – Код саги logInSaga класу AuthSagas

Для дотримання одного з існуючих принципів програмування DRY, а також для зменшення часу написання репететивного коду, було вирішено створити абстрактну сагу apiCall для обробки асинхронних запитів, яка буде використовуватися при подальшій розробці додатку (рисунок 4.13).

```
15
16  export function* apiCall(
17    // eslint-disable-next-line @typescript-eslint/no-explicit-any
18    fn: (...props: any[]) => AxiosResponse,
19    {
20      successAction,
21      failureAction,
22    }: { successAction: ActionCreatorWithPayload<object>; failureAction:
23    action: PayloadAction<object>
24  ) {
25    try {
26      // eslint-disable-next-line @typescript-eslint/no-unsafe-assignment
27      const { data } = yield call(fn, action.payload)
28      yield put(successAction(data as object))
29    } catch (e) {
30      yield put(failureAction(e.message as AxiosError))
31    }
32  }
```

Рисунок 4.13 – Код класу Utils

Як показано на рисунку вище, сага apiCall використовується як контекст та отримує на вхід сервісну функцію для асинхронних запитів, дію у разі її успішного виконання, та дію у разі помилки.

Використання цієї корисної функції зображено на рисунку 4.14.

```
42  export default [
43    takeLeading(LOG_IN_REQUESTED, apiCall, logIn, {
44      successAction: LOG_IN_SUCCESS,
45      failureAction: LOG_IN_FAILURE,
46    }),
47    takeLeading(REGISTER_REQUESTED, apiCall, register, {
48      successAction: REGISTER_SUCCESS,
49      failureAction: REGISTER_FAILURE,
50    }),
51  ]
52
```

Рисунок 4.14 – Код експорту класу AuthSagas

Коренева сага, що містить усі саги додатку, представлена на рисунку 4.15.

```
TS index.sagas.ts M ×
src > store > TS index.sagas.ts > ...
1  import { authSagas, templatesSagas } from '@/features'
2  import { all } from 'redux-saga/effects'
3
4  export default function* rootSagas() {
5    yield all([...authSagas, ...templatesSagas])
6  }
7
```

Рисунок 4.15 – Код класу RootSagas

Весь потік даних, при використанні архітектурного підходу розробки Redux зображений на рисунку 4.16.

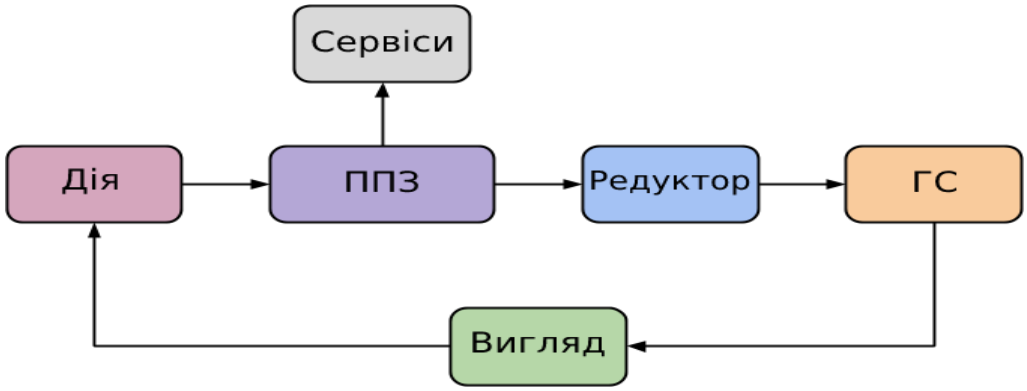

Рисунок 4.16 – Потік обробки даних додатку із використанням Redux

На рисунку вище можна помітити за своєю суттю круговий процес. Все починається з представлення, що створює дію. Створена дія в свою чергу може бути перехоплена ППЗ (проміжним програмним забезпеченням) із подальшим викликом сервісів.

Після, або під час (бо ППЗ може обробляти асинхронні події) виконання функції, результативна дія передається на обробку відповідному їй редуктору, після чого змінюється стан ГС (глобального сховища), а представлення отримує оновлений глобальний стан.

Щоб зв’язати представлення з глобальним сховищем Redux, а саме передачі дій в кореневий редуктор, отриманні даних з глобального сховища, зменшенні кількості коду при використанні у представленнях, створимо

допоміжну функцію useAuth, яка інкапсулює згадану щойно бізнес-логіку (рисунок 4.17)

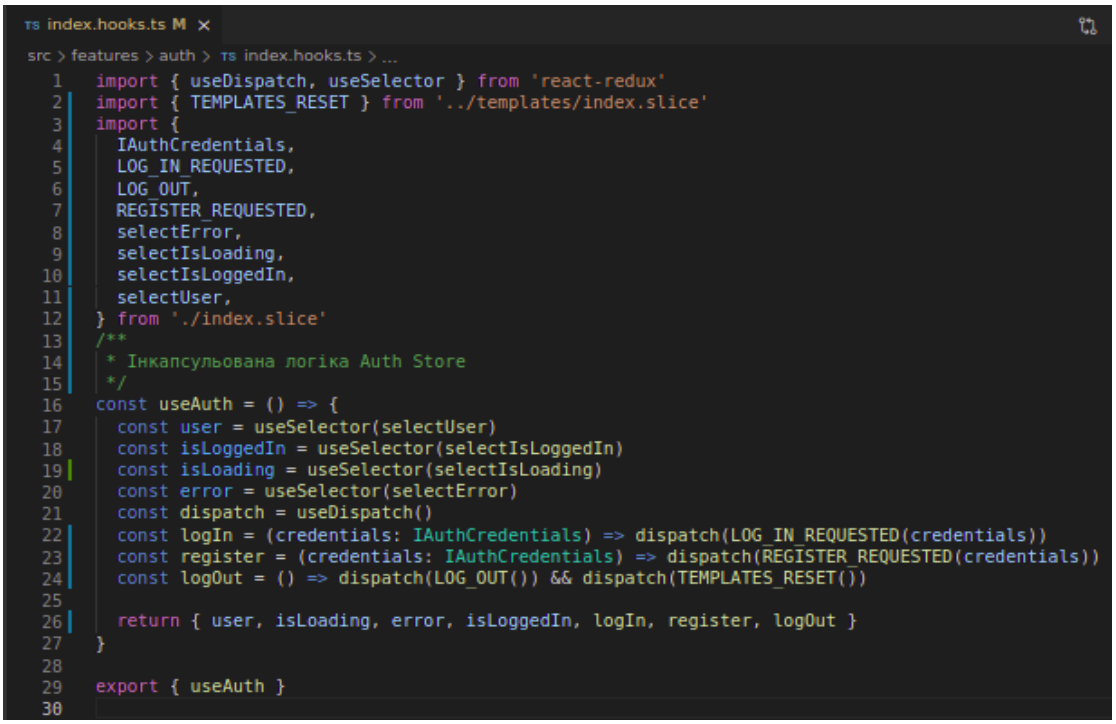

Рисунок 4.17 – Допоміжна функція з інкапсульованою бізнес-логікою

Функція useAuth повертає методи і властивості, що можуть бути використані у одному чи декількох представлень. Їх перелік з описом представлено в таблиці 4.10.

Таблиця 4.10 – Властивості та методи useAuth

Назва	Опис
user	Властивість, що містить об’єкт користувача із присутніми йому даними, а саме ім'я, прізвище, електронна пошта та масив шаблонів.
isLoggedIn	Властивість, що містить флаг, чи авторизован конкретний користувач, може бути як правдиве так і неправдиве значення.

isLoading	Властивість, що містить флаг, чи в даний момент відбувається загрузка, може бути як правдиве так і неправдиве значення.
error	Властивість, що містить повідомлення про помилку, може як існувати, так і бути відсутнім.
logIn	Метод, який посилає дію авторизації LOG_IN_REQUESTED в кореневий редуктор Redux.
register	Метод, який посилає дію реєстрації REGISTER_REQUESTED в кореневий редуктор Redux.
logOut	Метод, який посилає дію виходу користувача з поточної сесії LOG_OUT в кореневий редуктор Redux.

4.6 Налаштування бази даних MongoDB

Для того, щоб використовувати можливості NoSQL бази даних MongoDB необхідна строка підключення, яку можна знайти в особистому кабінету на сайті сервісу після створення кластеру з назвою.

Сервіс надає доступ до ресурсів MongoDB без додаткової плати. Безкоштовний кластер відрізняється від платного рядом обмежень, серед яких неможливість вибору апаратного обладнання, регіону кластеру та додаткових налаштувань. Вікно з формою створення нового кластеру показано на рисунку 4.18.

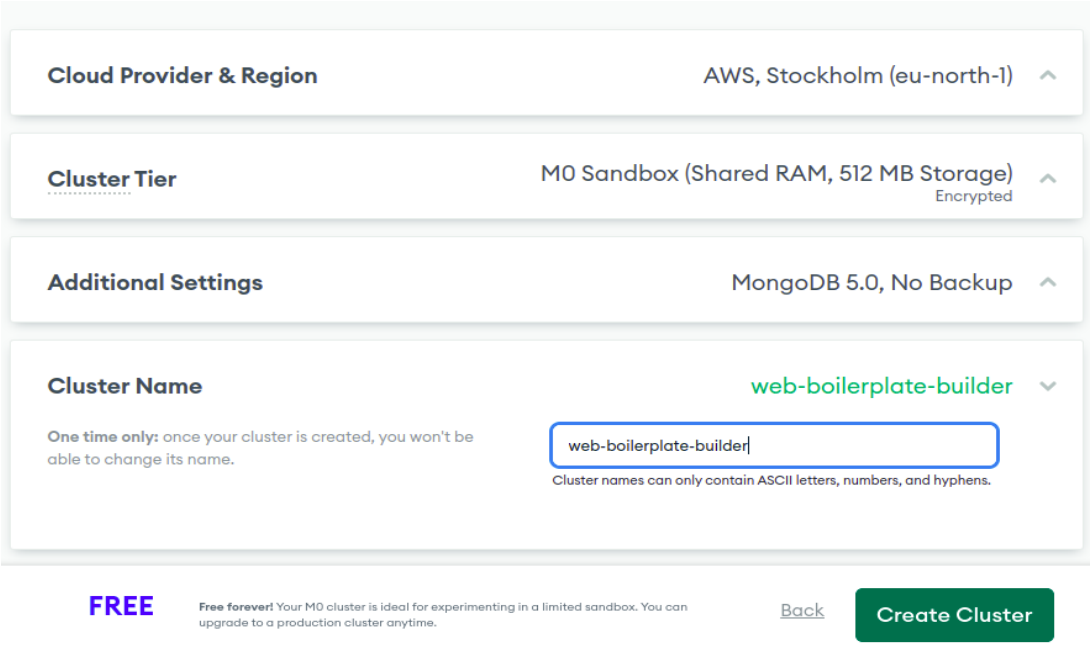

Рисунок 4.18 – Форма створення кластеру в сервісі Mongo

Строка з підключенням до бази даних містить приватну інформацію про користувача-ініціатора, а саме його ім’я та пароль. Завдяки цим даним відбувається подальша аутентифікація та авторизація користувача.

Доброю практикою при розробці веб-додатків є збереження приватної інформації, яка буде потрібна при роботі застосунку, у файлі з розширенням “.env”. Наповненість файлу показано на рисунку 4.19.

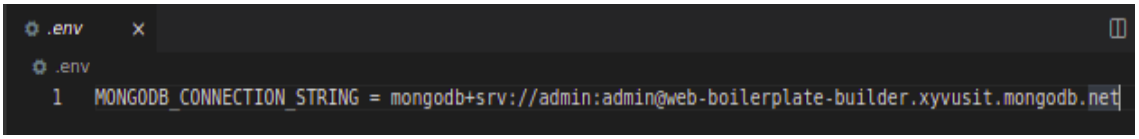

Рисунок 4.19 – Код файлу .env

Для ініціалізації підключення до бази даних MongoDB зі сторони серверу використаємо власну корисну функцію connectToDb (рисунок 4.20).

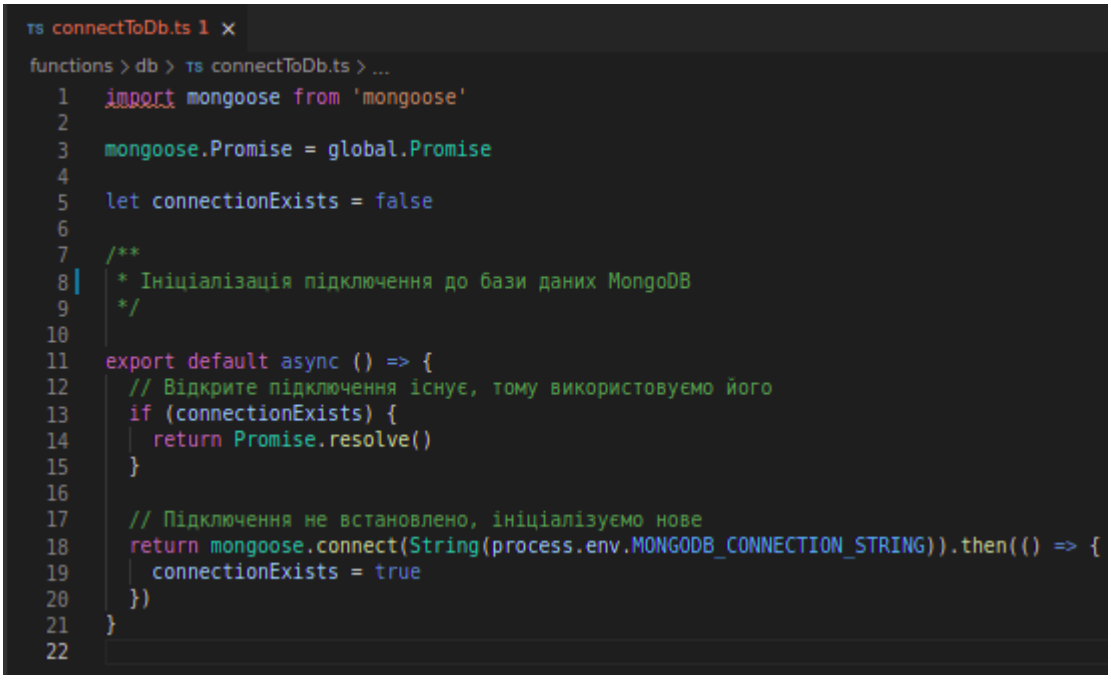

Рисунок 4.20 – Код класу connectToDb

Функція, задана на рисунку вище, містить бізнес-логіку, яка описує підключення нового користувача до NoSQL бази даних за допомогою бібліотеки Mongoose. На початковому етапі її виконання, ініціалізується булева змінна, що свідчить про поточний стан підключення користувача.

Під час виклику функції, перевіряється наявність правдивого значення в цій змінній, в разі виконання умови, асинхронний виклик нічого не повертає, а лише дозволяє виконання наступної функції.

В іншому випадку, встановлюється підключення завдяки бібліотеки Mongoose із строкою підключення, отриманою із файла з розширенням “.env”, і якщо підключення вдале, встановлюється значення флагу в правдиве значення.

4.7 Вузли серверу Express

Під час розробки REST API, використовуючи фреймворк Express для середовища виконання NodeJS, потрібно вказати деяку кількість вузлів (або шляхів) та назначити їм відповідні обробники. Наприклад, для авторизації

користувача був доданий обробник POST-запитів до конкретного вузла, як на рисунку 4.21.

Рисунок 4.21 – Код вузлу з авторизацією роутера Users

Загальний роутер users виступає в ролі акумулятора підписників подій, а саме REST запитів, із зазначеними обробниками до відповідних вузлів. Під час процесу авторизації користувача зі сторони серверу зчитується “тіло” отриманого запиту, а саме електронна пошта та пароль.

Далі відбувається ініціалізація підключення до бази даних в асинхронному режимі, але функція чекає на відповідь для виконання наступних шагів інструкції.

Після цього, відбувається пошук користувача із отриманими даними у базі даних MongoDB завдяки методу findOne. В разі відсутності користувача в базі даних, повертаємо відповідну помилку із кодом помилки стандарту протоколу HTTP та текстом повідомлення.

В іншому випадку, коли користувач з вхідними даними був знайдений, підготовлюємо відповідь - сутність користувача до формату JSON. Наприкінці авторизації, повертаємо об’єкт із самим користувачем, але з міркувань безпеки забираємо з форми відповіді пароль.

Фреймворк Express не має автоматичного підбору коду відповіді як реакцію на деяку подію, а отже розробник повинен сам вказати код відповіді та її тіло (файл, об’єкт, текст тощо). Перелік використаних відповідей під час розробки дослідницького проєкту зазначено в таблиці 4.11.

Таблиця 4.11 – Перелік HTTP кодів відповідей та їх пояснень

Код	Опис
400	Bad Request — Неправильний запит. Запит не може бути виконаний з причини невірного синтаксису
401	Unauthorized — Несанкціонований доступ. Використовується у разі необхідності аутентифікації користувача, яка відбулася невдало, або ще не відбулася перед запитом ресурсу.
404	Not Found — Не знайдено. Ресурс не знайдено, але він може бути доступний в майбутньому.
409	Conflict — Конфлікт. Вказує, що запит не може бути опрацьований через конфлікт у запиті, наприклад, конфлікт додавання існуючих даних.
500	Internal Server Error — Внутрішня помилка сервера. Будь-яка внутрішня помилка сервера, що не входить в рамки описаного класу
200	OK. Стандартна відповідь про успішне виконання HTTP запиту. Фактична відповідь буде залежати від методу запиту. Для запитів GET, відповідь буде містити зміст об'єкту чи сторінки у відповідності до зверненого ресурсу. Для запитів POST відповідь буде містити опис об'єкту чи результат операції.
201	Created — Створено. Запит був виконаний і в результаті нього був створений новий ресурс.

4.8 Перегляд бази даних

Після ініціалізації NoSQL бази даних MongoDB створені колекції можна побачити, зробивши відповідний запит на їх отримання вручну, або подивитися через інформаційну панель бази даних безпосередньо на сайті з сервісом. Останній варіант більш користувацько-орієнтований, адже має візуальний інтерфейс, та абстрагує розробника від написання запитів власноруч, економлячи час, виділений на перегляд, тестування та зміну записів в таблицях колекцій.

В інтерфейсі інформаційної панелі бази даних MongoDB є низка корисних функцій, що може знадобитися під час розробки застосунку з архітектурним рішенням клієнт-сервер, як наприклад сортування, пошук записів, візуалізація збережених даних, перегляд показників продуктивності, використання ресурсів, завантаженості кластеру тощо.

Приклад відображення документів колекції Users показано на рисунку 4.22.

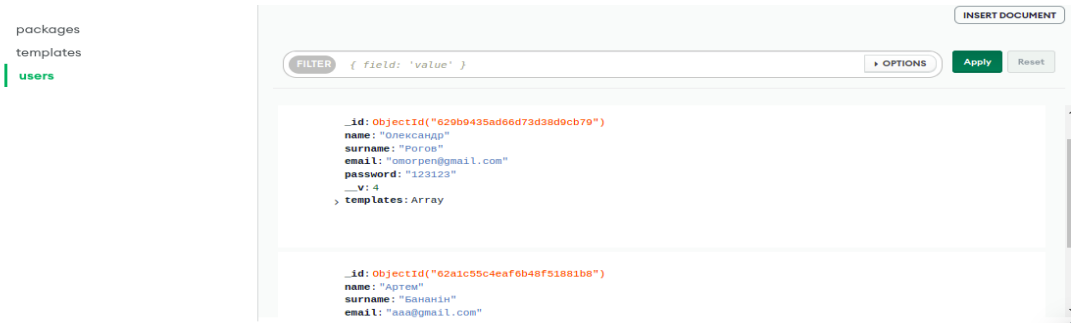

Рисунок 4.22 – Зміст колекції Users

В наведеному вище прикладі є документ, що відображає сутність користувача із відповідною йому моделлю даних. Розгорнутий вигляд документу можна побачити на рисунку 4.23.

Рисунок 4.23 – Розгорнутий вигляд документу Users

4.9 Діаграма залежностей

VS Code дозволяє перевіряти архітектурні залежності у своїх рішеннях за допомогою додаткових розширень, які доступні розробникам безкоштовно в величезній онлайн-колекції.

Можна використати розширення DependencyGraph, яке приймає на вхід стартовий файл, а потім генерує граф залежностей, заточений спеціально під конкретне рішення. Ця функція допомагає забезпечити дотримання розробниками архітектурних обмежень програми під час редагування коду.

Як і традиційна діаграма архітектури, діаграма залежностей визначає основні компоненти або функціональні одиниці конструкції та їх взаємозалежності. Кожен вузол на діаграмі, який називається шаром представляє логічну групу просторів імен, проєктів чи інших артефактів.

Діаграма залежностей до відповідної системи, що проєктується, наведена у документі ІА82.210БАК003 ДЗ.

4.10 Розгортання додатку на Netlify

Після проходження первинних шагів налаштування середовища розробки, необхідних бібліотек, розроблення програмної архітектури, появи деякої частини функцій додатку, а саме коли застосунок вже має робоче

підґрунтя для запуску, можна зайнятися розгортанням його на хостинг-сервісі.

Складність розгортання клієнт-серверного застосунку має під собою вагомий нюанс підтримки серверу на окремому хостингу. Netlify в свою чергу надає можливість розгорнути веб-додаток із серверним REST API, що знаходяться в одному репозиторії Github, завдяки Netlify Functions.

При перенесенні коду на хостинг, додатку автоматично надається унікальна адреса, за якою будь-який користувач може мати доступ до перегляду. Додатки до застосунку на Netlify, їх налаштування, або ж команди запуску самого застосунку тощо, можна вказати через візуальний інтерфейс на самому сайті сервісу, або ж через файл з конфігурацією, доданий до кореневої папки з проєктом.

Було обрано створити файл, що містить налаштування, в самому репозиторію. Це дозволить вільно мігрувати додаток, що розгортається, між акаунтами Netlify. Код файлу із конфігурацією зображено на рисунку 4.24.

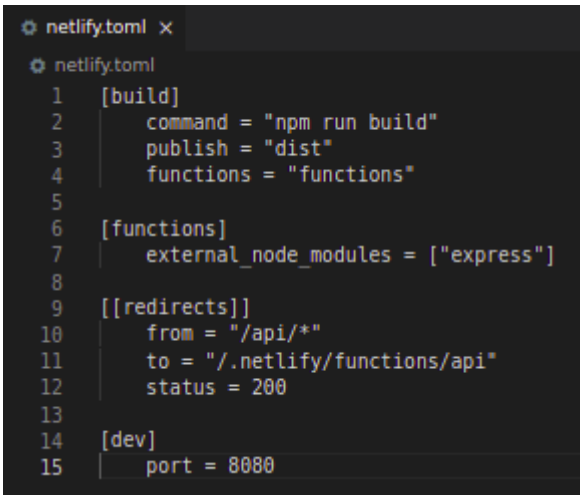

Рисунок 4.24 – Зміст файлу з конфігурацією Netlify

В файлі, згаданому вище, є декілька важливих аспектів налаштувань. По-перше, для успішного розгортання будь-якого веб-додатку, пріоритетною потребою є вказання команди побудови. Під час побудови відбувається внутрішня оптимізація ресурсів, обраний раніше інструмент Vite для

побудови, внутрішньо, використовує esbuild - сучасний швидкий комплектувальник залежностей та ресурсів.

Також, після цього потрібно вказати шлях до директорії, що містить Netlify Functions. Сервіс Netlify може самостійно обрати стратегію побудови при розгортанні, використовуючи деякі популярні фреймворки для створення статичних сайтів, як наприклад Gatsby.

В межах цього проєкту, шлях до папки з артефактами побудови, потрібно вказати явно, в конкретному випадку це папка functions.

Деякі залежності, що не містяться у вбудованому наборі Netlify Functions, як наприклад фреймворк Express, потрібно вказати як масив з назвами у властивості external node modules (від англ. “зовнішні модулі Node”).

Для комфортного використання розгорнутого REST API із власним шляхом до нього, потрібно вказати перенаправлення від стандартної адреси до тієї, що потрібна, в даному випадку це перенаправлення до шляху “/api”.

Це дозволить підвищити розуміння при створенні запитів та зменшити складність набору адреси, у порівнянні зі стандартною.

Для запуску Netlify Functions локально, найчастіше, під час розробки, потрібно вказати додаткову інформацію про порт, на якому функції будуть запускатися, в даному випадку це “port = 8080”.

Команди запуску застосунку локально, локально із Netlify функціями, та команда побудови вказані на рисунку 4.25.

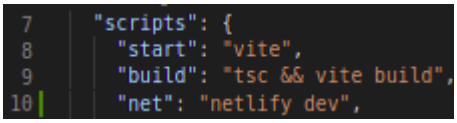

Рисунок 4.25 – Зміст скриптів застосунку в package.json

Виклик скрипту start через менеджер npm запускає застосунок локально, скрипт build ініціює побудову завдяки інструменту Vite, а команда net запускає додаток та піднімає сервер з Netlify Functions локально, тому

можна тестувати функціональність без оновлення розгорнутого сайту на хостинг-сервісі.

Лог сервісу Netlify після успішного розгортання додатку відображено на рисунку 4.26.

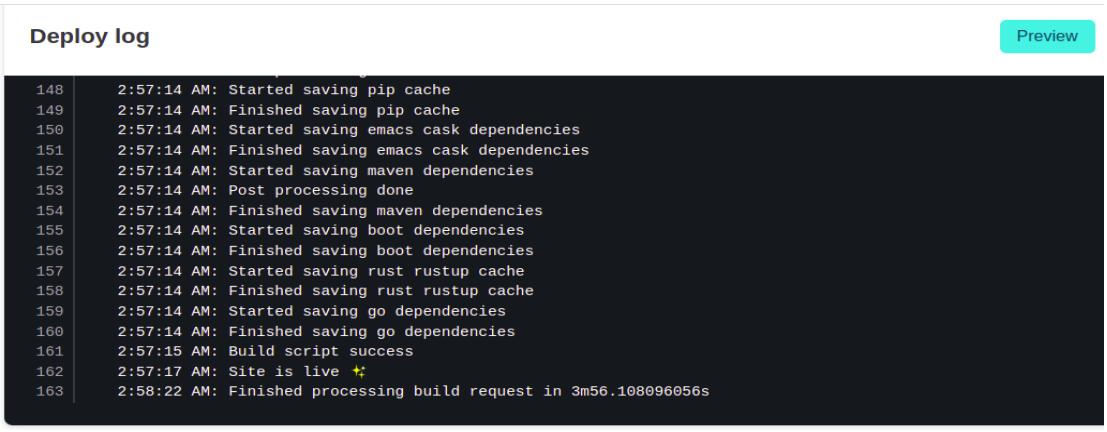

Рисунок 4.26 – Лог Netlify, що свідчить про успішне розгортання веб-додатку

Висновки до розділу 4

В розділі описано ключові аспекти розробки веб-додатку конструктора шаблонів для розробки веб-застосунків. Етапи розробки подані на прикладах побудови окремого функціоналу додатку та таких аспектів як проєктування архітектури клієнт-сервер з використанням React та Express відповідно, створення моделей та представлень, підключення серверної частини додатку до бази даних MongoDB і посилання запитів до неї за допомогою Mongoose, застосування корисних функцій глобального сховища Redux Toolkit і обробка асинхронності завдяки Redux Saga і бібліотеки Axios.

Клієнт-серверна частина додатку була розгорнута завдяки хостинг-сервісу Netlify та допоміжній технології Netlify Functions. Під час розробки було використано можливості статичної перевірки типів мови програмування Typescript і описано використання деяких принципів програмування. Також було обрано сучасні інструменти та підходи до веб-розробки. Розроблений

веб-додаток містить базовий функціонал, створену клієнт-серверну архітектуру (діаграма компонентів в документі ІА82.210БАК.003 Д2) та налаштовані її компоненти.

Завдяки сучасним інструментам підтримується на старіших версіях популярних браузерів і може бути запущений на будь-яких операційних системах, оскільки розміщується на сервері та не потребує попереднього встановлення.

5 ІНСТРУКЦІЯ КОРИСТУВАЧА

Після переходу по відповідній URL-адресі, де знаходиться додаток, відбувається встановлення підключення з сервером. Після успішного з’єднання, браузер починає завантаження сторінки. Коли сторінка готова до відображення зі сторони користувацького браузеру, можна побачити головну сторінку застосунку (рисунок 5.1).

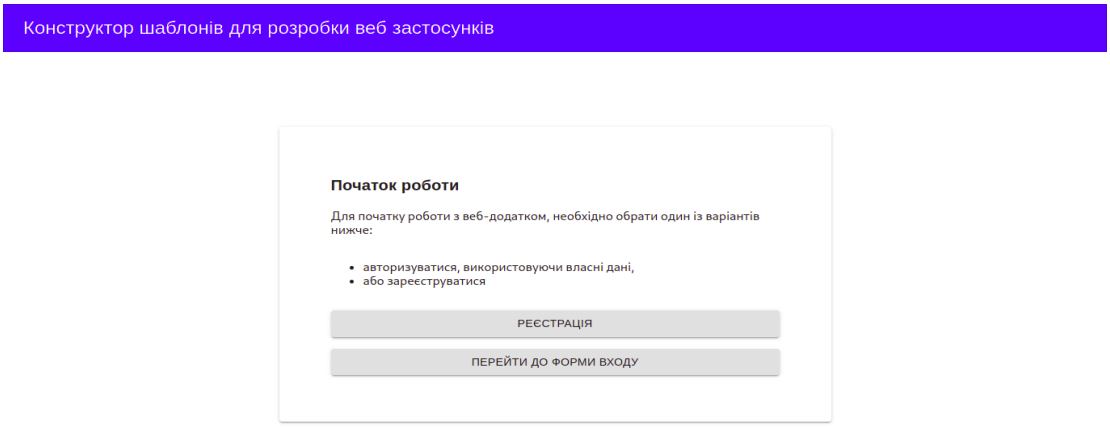

Рисунок 5.1 – Вигляд головної сторінки для неавторизованого користувача

На сторінці зображено повідомлення про початок роботи з застосунком, а також посилання на форму входа та реєстрації. За замовчуванням, меню з кнопками навігації до сторінок з аутентифікацією та реєстрацією показуються для неавторизованого користувача. У випадку, коли користувач вже авторизован, вигляд головної сторінки змінюється, як показано на рисунку 5.2.

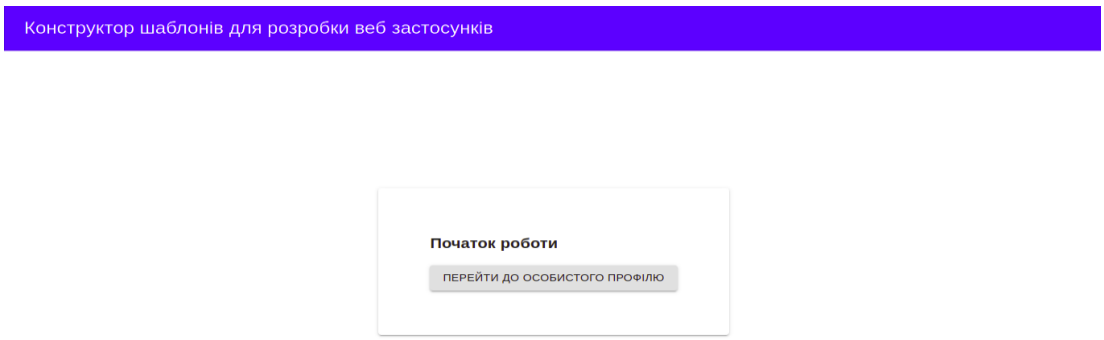

Рисунок 5.2 – Вигляд головної сторінки для авторизованого користувача

Як можна побачити із рисунку, наведеному вище, для авторизованого користувача відображається кнопка із переходом до особистого профілю, а повідомлення про авторизацію чи реєстрацію, і відповідні навігаційні елементи приховані.

Для початку роботи з додатком потрібно мати зареєстрованого користувача в системі. Після переходу із головної сторінки для форми реєстрації, з’являється форма реєстрації (рисунок 5.3).

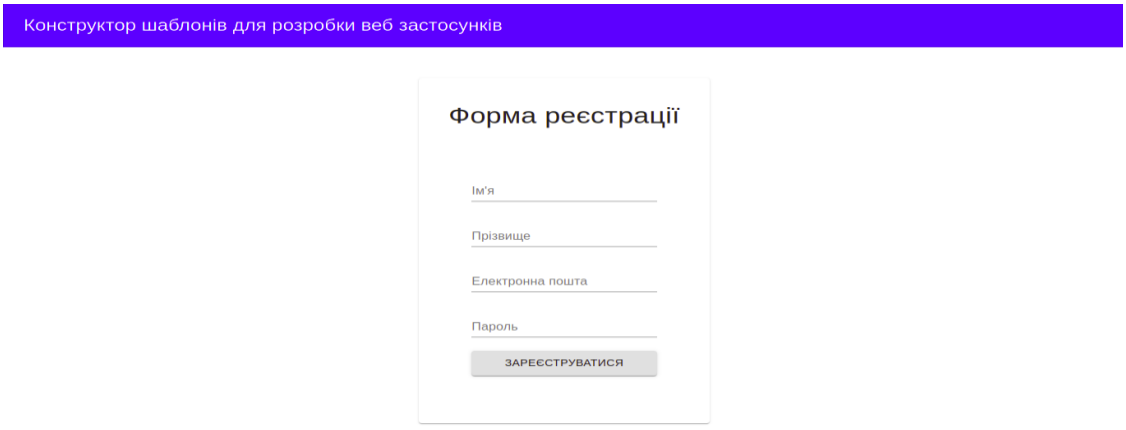

Рисунок 5.3 – Стандартний вигляд форми реєстрації

Форма реєстрації має декілька полів для вводу, серед яких ім'я, прізвище, електронна пошта, пароль, а також кнопка, при натисканні якої,

запит посилається на сервер де згодом обробляється, а клієнтський застосунок отримує відповідь.

При вводі даних у доступні поля форми реєстрації, відбувається миттєва перевірка валідності даних в режимі реального часу. Біля кожного з полей відображається поточний стан локальної валідації.

У разі спроби відправлення користувачем остаточної форми в тому самому вигляді що і ініціалізована, з’являться повідомлення про необхідність в заповненні полей, бо вони є обов’язковими, про що також свідчить зміна їх кольору на червоний (рисунок 5.4).

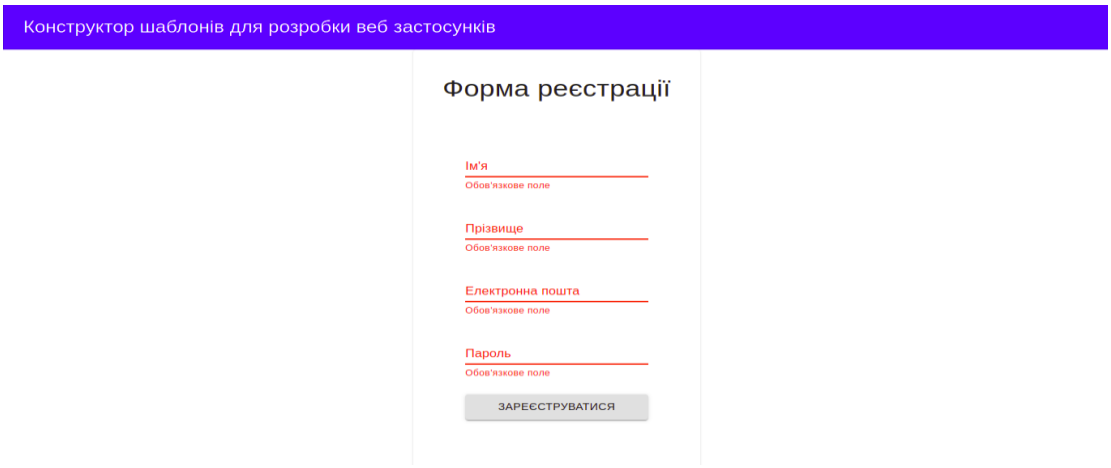

Рисунок 5.4 – Помилка валідації всіх полів форми реєстрації

Помилка валідації електронної пошти зображена на рисунку 5.5.

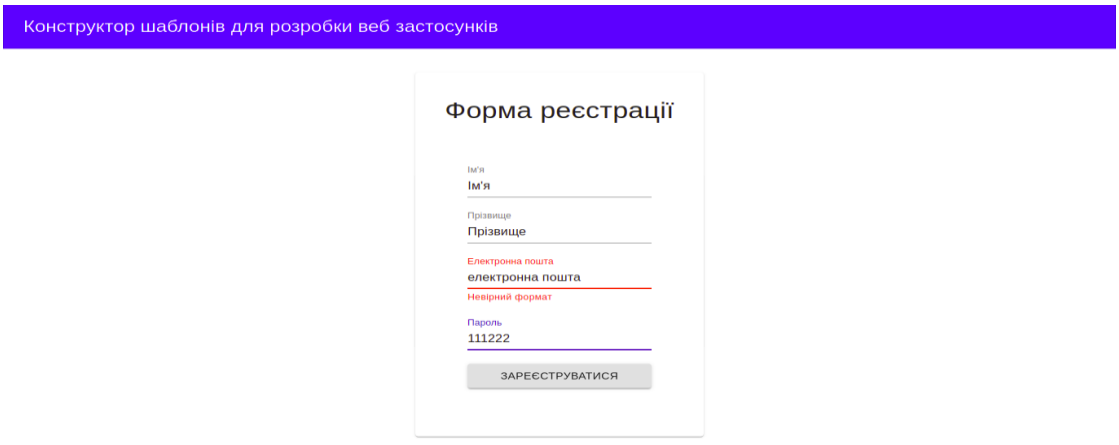

Рисунок 5.5 – Помилка валідації електронної пошти форми реєстрації

Поле, яке приймає на вхід від користувача електронну пошту, має унікальне правило валідації, воно повинно відповідати формату стандартної електронної пошти з латинськими символами та/або цифрами, одним розділенням через “собачку” (@), а після неї наявність розділення через точку при вказанні доменного імені поштового провайдера.

Нижче представлено вигляд сторінки з формою авторизації користувача (рисунок 5.6).

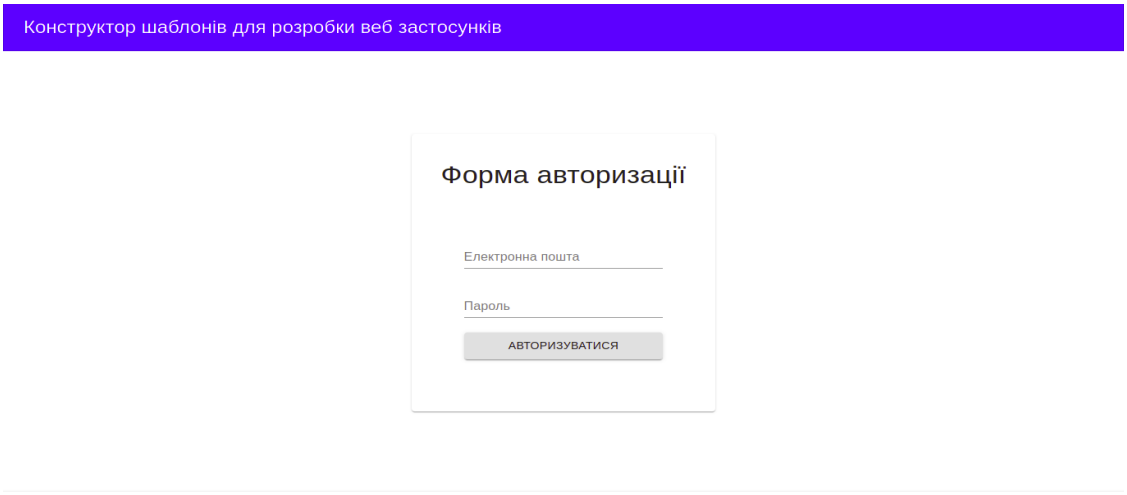

Рисунок 5.6 – Стандартна форма авторизації

Якщо користувач вже існує в базі даних, то відповідний шаг з реєстрацією може бути пропущено і перейдено одразу до форми з авторизацією. Для поля з електронною поштою встановлена валідація за форматом, аналогічним з відповідним полем у формі реєстрації.

У разі введення користувачем правильних облікових даних, але за відсутності інтернет з’єднання на стороні клієнта, чи появі помилки на стороні серверу, користувач побачить повідомлення про помилку, яка щойно з’явилася. Це вікно може містити декілька типів повідомлень.

У разі отримання будь-якої помилки від серверу, відображає найактуальнішу. Користувач має можливість закрити вікно з повідомлення про помилку натиснувши на одне з полів форми, або почекавши деякий час,

в даному випадку 4 секунди, після чого закриття відбудється в автоматичному режимі. Приклад помилки зображено на рисунку 5.7.

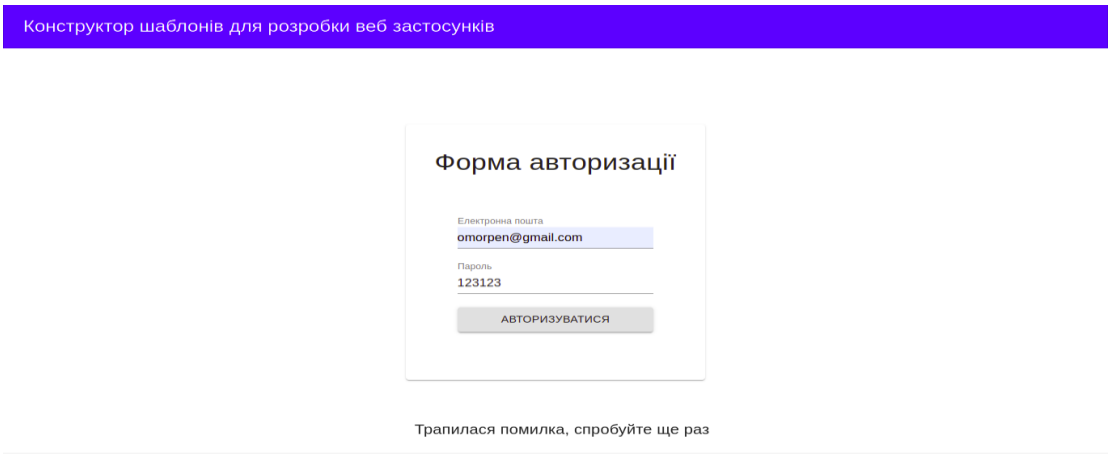

Рисунок 5.7 – Повідомлення про помилку на сторінці з формою авторизації

Сторінка зі списком шаблонів користувача зображено на рисунку 5.8.

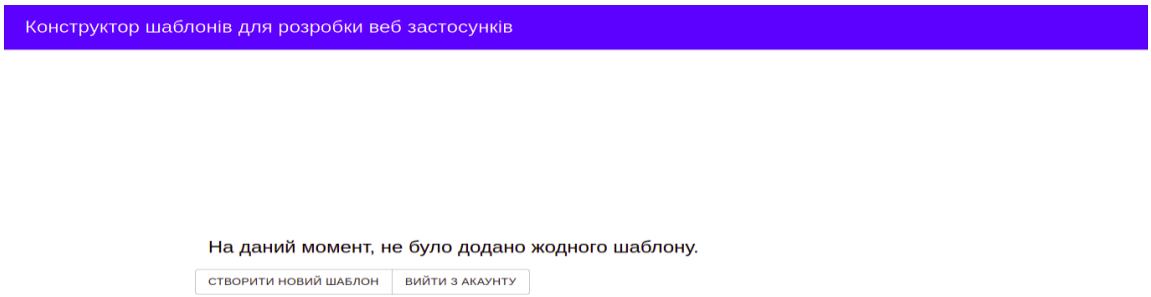

Рисунок 5.8 – Сторінка з пустим списком шаблонів користувача

Після першої авторизації, користувач попадає до сторінці його профілю, яка містить пустий список шаблонів, а також повідомлення про те, що не було додано жодного шаблону до бази даних.

Також, є візуальне представлення дії виходу з акаунту у вигляді кнопки, при натисканні якої відбувається перенаправлення до сторінки з авторизацією, а дані про користувача, що зберігалися локально, знищуються.

Якщо користувач обрав іншу кнопку, то він отримує форму зі створенням нового шаблону (рисунок 5.9).

Рисунок 5.9 – Форма для створення нового шаблону

Вигляд сторінки зі списком доданих шаблонів показано на рисунку 5.10. Біля кожного з шаблонів розташовані елементи управління. При натисканні кнопки «Завантажити», користувач отримує запит на завантаження шаблону на локальний накопичувач.

При натисканні кнопки «Детальніше», користувач перенаправляється на сторінку з детальною інформацією про шаблон.

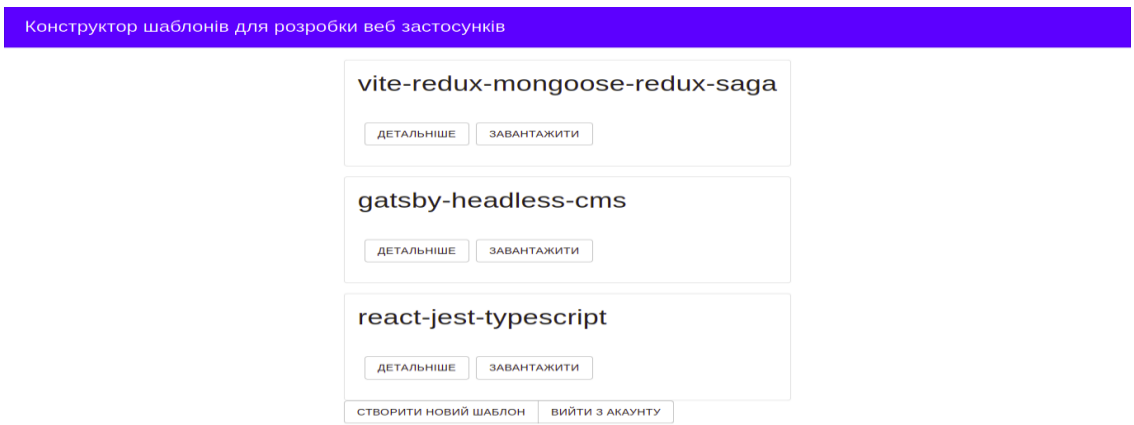

Рисунок 5.10 – Сторінка зі списком доданих шаблонів користувача

Сторінка з детальною інформацією про шаблон містить список залежностей шаблону, кнопки «Додати нову залежність, змінити конфігурацію залежності «Змінити конфігурацію», та кнопку видалення «Видалити залежність» (рисунок 5.11).

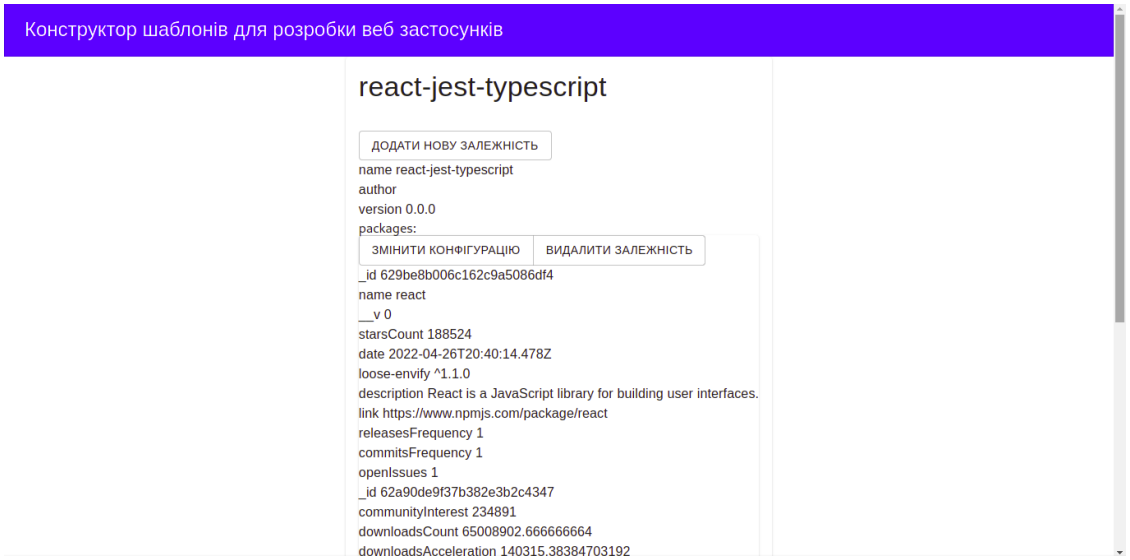

Рисунок 5.11 – Сторінка з інформацією про шаблон

Нижче на рисунку 5.12 зображені елементи управління конкретним шаблоном.

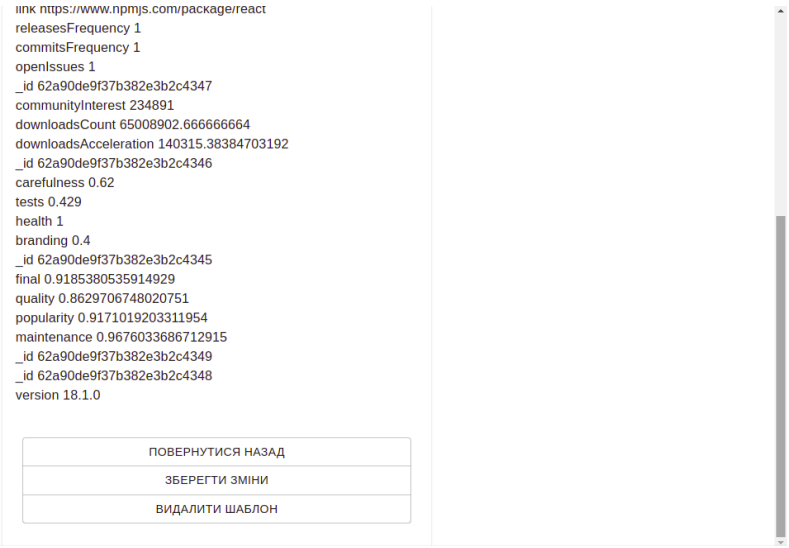

Рисунок 5.12 – Елементи управління шаблоном

Під інформацією про шаблон розташовані кнопки, при натисканні яких користувач може повернутися назад до загального списку шаблонів «Повернутися назад», зберегти внесені зміни про шаблон «Зберегти зміни» і видалити шаблон «Видалити шаблон».

Висновки до розділу 5

В розділі описано користувацький інтерфейс розробленого веб-застосунку. Візуальне представлення надає можливість користування основним функціоналом розробленої системи зі сторони кінцевого користувача. Користувач отримує доступ до усієї необхідної інформації про шаблон на відповідній сторінки додатку.

Доступ до відповідних функцій надається за публічною веб-адресою, що веде на головну сторінку веб-застосунку.

Також реалізовано виведення помилок в разі введення некоректних даних, помилок при реєстрації чи авторизації. Створений графічний інтерфейс простий для використання та освоєння.

ВИСНОВКИ

В ході виконання даного дослідницького проєкту був спроєктований та розроблений конструктор для розробки вебзастосунків. Реалізований додаток містить базовий робочий функціонал за обраною темою, а підібрана сучасна архітектура дозволяє скоріше впровадження нових функцій в майбутньому.

З використанням додатку користувачу надається можливість створювати, редагувати, переглядати та видаляти записи про шаблони та їх пакетні залежності, завантажувати шаблон на локальний накопичувач.

З огляду на існуючі аналоги розроблений додаток має ряд переваг, оскільки акцентує увагу на створенні шаблонів саме для вебзастосунків, має можливість налаштування конфігурацій залежностей, доступний на старіших версіях браузерів та має простий користувацький інтерфейс.

Також, надає можливість кінцевому користувачу зберегти створений шаблон у базі даних серверу, а за необхідністю завантажити на локальний накопичувач.

Додаток був створений з використанням сучасних інструментів розробки, підходів проєктування архітектури та програмної реалізації, що в свою чергу дає можливість подальшого розширення функціоналу системи та її модернізації.

Отримані результати відповідають завданню, поставленому на проєктування дослідницького проєкту.

На даному етапі розроблений додаток може бути використаний для вивчення досліджуваної проблематики та отримання більшого розуміння про проблему користувача, яку може вирішити конструктор шаблонів для розробки вебзастосунків.

ПЕРЕЛІК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Додаток для створення шаблонів Java застосунків. URL: https://start.spring.io/
2. Ініціалізатор React застосунку – Create React App. URL: https://create-react-app.dev/
3. Єрьоміна Н. В. Проєктування баз даних / Н. В. Єрьоміна. – Київ: Київський національний економічний університет, 1998. – 49 с
4. Функціональне програмування в JavaScript. URL: https://dou.ua/forums/topic/37548/
5. Що таке функціональне програмування? URL: https://uk.theastrologypage.com/functional-programming/
6. Create a React App from Scratch in 2021. URL: https://javascript.plainenglish.io/create-a-react-app-from-scratch-in-2021-8e9948602e9c/
7. TypeScript vs JavaScript: Which One Should You Choose? URL: https://radixweb.com/blog/typescript-vs-javascript/
8. TypeScript: JavaScript With Syntax For Types. URL: https://www.typescriptlang.org/
9. MongoDB | Build Faster. Build Smarter. URL: https://www.mongodb.com/docs/
10. An overview of HTTP. URL: https://developer.mozilla.org/en-US/docs/Web/HTTP/Overview/
11. How to Design a REST API. URL: https://restfulapi.net/rest-api-design-tutorial-with-example/
12. Getting started with React. URL: https://reactjs.org/docs/getting-started.html
13. Welcome to Netlify. URL: https://docs.netlify.com/
14. Getting started with NodeJS. URL: https://nodejs.org/en/docs/guides/getting-started-guide/

15. Basic Routing – Express. URL: https://expressjs.com/en/starter/basic-routing.html

16. What is Axios? URL: https://axios-http.com/docs/intro/

17. Getting started with Redux. URL: https://redux.js.org/introduction/getting-started/

18. Recipes – Redux Saga. URL: https://redux-saga.js.org/docs/recipes/

19. Usage Guide – Redux Toolkit. URL: https://redux-toolkit.js.org/usage/usage-guide/

20. What Is the MERN Stack? Introduction & Examples. URL: https://www.mongodb.com/mern-stack/

ДОДАТОК А

Код програми

```
import mongoose from 'mongoose'
mongoose.Promise = global.Promise

let connectionExists = false

/**
* Ініціалізація підключення до бази даних MongoDB
*/

export default async () => {
// Відкрите підключення існує, тому використовуємо його
if (connectionExists) {
return Promise.resolve()
}

// Підключення не встановлено, ініціалізуємо нове
return mongoose.connect(String(process.env.MONGODB_CONNECTION_STRING)).then(() => {
connectionExists = true
})
}

import mongoose from 'mongoose'
import { IPackageEntity } from './types'

const qualitySchema = new mongoose.Schema({
carefulness: Number,
tests: Number,
health: Number,
branding: Number,
})

const popularitySchema = new mongoose.Schema({
communityInterest: Number,
downloadsCount: Number,
downloadsAcceleration: Number,
})

const maintenanceSchema = new mongoose.Schema({
```

```
releasesFrequency: Number,
commitsFrequency: Number,
openIssues: Number,
})

const scoreDetailSchema = new mongoose.Schema({
quality: Number,
popularity: Number,
maintenance: Number,
})

const scoreSchema = new mongoose.Schema({
final: Number,
detail: scoreDetailSchema,
})

const packageSchema = new mongoose.Schema({
name: String,
version: String,
description: String,
date: Date,
link: String,
starsCount: Number,
dependencies: {
type: Map,
of: Object,
},
quality: qualitySchema,
popularity: popularitySchema,
maintenance: maintenanceSchema,
score: scoreSchema,
})

/**
* Модель сутності пакету
*/

export default mongoose.model<IPackageEntity>('Package', packageSchema)

import mongoose from 'mongoose'
import Package from './Package'
```

```
import { ITemplateEntity } from './types'

const templateSchema = new mongoose.Schema({
name: {
type: String,
required: [true, "Назва - обов'язкове поле для заповнення"],
minlength: [1, 'Назва має містити більше 1 символу'],
maxlength: [128, "Ім'я має містити не быльше 128-х символів"],
},
author: {
type: String,
default: '',
required: false,
maxlength: [32, 'Автор має містити не быльше 32-х символів'],
},
version: {
type: String,
default: '0.0.0',
required: false,
validate: (value: string) => /([0-9]\.){2}[0-9]/.test(value),
},
packages: {
type: [mongoose.Schema.Types.ObjectId],
ref: Package,
},
})

/**
* Модель сутності шаблону
*/

export default mongoose.model<ITemplateEntity>('Template', templateSchema)

// Інтерфейс сутності пакету
export interface IPackageEntity {
name: string
version: string
description: string
lastPublishDate: Date
dependencies: { [dependencyName: string]: string }
devDependencies: { [devDependencyName: string]: string }
```

```
peerDependencies: { [peerDependencyName: string]: string }
starsCount: number
popularity: {
communityInterest: number
downloadsCount: number
}
score: {
final: number
detail: {
quality: number
popularity: number
maintenance: number
}
}
}

// Інтерфейс сутності шаблону
export interface ITemplateEntity {
user: string
name: string
version: string
author: string
scripts: { [scriptName: string]: string }
packages: Array<IPackageEntity>
}

// Інтерфейс сутності користувача
export interface IUserEntity {
name: string
surname: string
email: string
password: string
templates: Array<ITemplateEntity>
}

import mongoose from 'mongoose'
import { IUserEntity } from './types'

const userSchema = new mongoose.Schema({
name: {
type: String,
```

```
required: [true, "Ім'я - обов'язкове поле для заповнення"],
minlength: [2, "Ім'я має містити більше 2-х символів"],
maxlength: [32, "Ім'я має містити не быльше 32-х символів"],
},
surname: {
type: String,
required: [true, "Прізвище - обов'язкове поле для заповнення"],
minlength: [2, 'Прізвище має містити більше 2-х символів'],
maxlength: [64, 'Прізвище має містити не быльше 64-х символів'],
},
email: {
type: String,
required: [true, "Електронна пошта - обов'язкове поле для заповнення"],
validate: {
validator: (value: string) => /^[a-z1-9]+@[a-z1-9]+\.[a-z1-9]+$/.test(value),
message: 'Електронна пошта невірного формату',
},
},
password: {
type: String,
required: [true, "Пароль - обов'язкове поле для заповнення"],
minlength: [2, 'Пароль має містити більше 2-х символів'],
},
templates: {
type: [mongoose.Schema.Types.ObjectId],
ref: 'Template',
},
})

/**
* Модель сутності користувача
*/

export default mongoose.model<IUserEntity>('User', userSchema)

import compression from 'compression'
import cors from 'cors'
import express from 'express'

/**
* Проміжне програмне забезпечення.
```

```
*/

export default [
// для успішного виконання запитів на зовнішні URL
cors(),

// GZIP компресія відповідей
compression(),

// парсинг JSON запитів
express.json(),

// парсинг Query параметрів
express.urlencoded({ extended: true }),
]

import { RequestHandler } from 'express'

/**
* Проміжний обробник при зверненні до неіснуючого шляху
*/

export default ((req, res) => {
// Повертаємо статус: Not Found - не знайдено
res.status(404).send(`<h1>За заданим URL: <u>${req.path}</u> - немає жодної відповіді</h1>`)
}) as RequestHandler

import express from 'express'
import './packages'
import * as routers from './_exports'

/**
* Загальний роутер. Поєднує існуючі.
*/

const rootRouter = express.Router()

Object.values(routers).forEach((router) => rootRouter.use(router))

export default rootRouter
```

```
import express from 'express'
import fetch from 'node-fetch'
import connectToDb from '../db/connectToDb'
import Package from '../db/Package'
import Template from '../db/Template'

/**
* Обробка шляхів пов'язаних із модульними пакетами.
*/

const router = express.Router()

router
.route('/packages/:name')
.get(async ({ params }, res) => {
const { name } = params

let response
try {
response = await fetch(`https://api.npms.io/v2/package/${name}`)
} catch (e) {
console.log(e)
}

if (!response || response.status !== 200) {
// Повертаємо статус: Not Found - не знайдено
return res.status(404).send('Жодного пакету не знайдено')
}
const {
collected: {
metadata: { version, description, date, links, dependencies },
github: { starsCount },
},
evaluation,
score,
} = await response.json()

// Ініціалізуємо підключення до бази даних
await connectToDb()
console.log(1)
try {
```

```
// Зберегаємо результат пошуку пакету у базі даних
const existingPackage = await Package.findOneAndUpdate(
{ name },
{
name,
version,
description,
date,
link: links?.npm || links?.homepage || links?.repository,
dependencies,
starsCount,
quality: evaluation.quality,
popularity: evaluation.popularity,
maintenance: evaluation.maintenance,
score,
},
{ upsert: true, new: true }
)
// Повертаємо статус: OK - все добре
return res.status(200).send(existingPackage)
} catch (e) {
return res.status(500).send(`Внутрішня помилка ${String(e.message)}`)
}
})
.post(async ({ params, body }, res) => {
const { name } = params
const { templateName } = body as { templateName: string }

// Ініціалізуємо підключення до бази даних
await connectToDb()

// Шукаємо пакет за наданими даними
const existingPackage = await Package.findOne({ name }).exec()

if (!existingPackage) {
// Повертаємо статус: Not Found - не знайдено
return res.status(404).send('Нажаль, жодного пакету із введеними даними не існує')
}

// Шукаємо шаблон за наданими даними
const existingTemplate = await Template.findOne({ name: templateName }).exec()
```

```
if (!existingTemplate) {
// Повертаємо статус: Not Found - не знайдено
return res.status(404).send('Нажаль, жодного шаблону із введеними даними не існує')
}

existingTemplate.packages.push(existingPackage)

await existingTemplate.save()

// Повертаємо статус: OK - все добре
return res.status(200).send(existingTemplate)
})

export { router as packages }

import express from 'express'
import connectToDb from '../db/connectToDb'
import Template from '../db/Template'
import { IUserEntity } from '../db/types'
import User from '../db/User'

/**
* Обробка шляхів пов'язаних із шаблонами.
*/

const router = express.Router()

router
.route('/templates')
.get(async ({ query }, res) => {
const { _id } = query as { _id: string }

// Ініціалізуємо підключення до бази даних
await connectToDb()

// Шукаємо користувача за наданими даними
const existingUser = await User.findById(_id).exec()

if (!existingUser) {
// Повертаємо статус: Not Found - не знайдено
```

```
return res.status(404).send('Нажаль, жодного користувача із введеними даними не існує')
}

// Заповнюємо поле із шаблонами існуючими за їхніми "_id"
const { templates } = await existingUser.populate({
path: 'templates',
populate: {
path: 'packages',
model: 'Package',
},
})

// Повертаємо статус: OK - все добре, та існуючі шаблони користувача
return res.status(200).send(templates)
})
.post(async ({ body }, res) => {
const { name } = body as ITemplateEntity

// Ініціалізуємо підключення до бази даних
await connectToDb()

// Шукаємо користувача за наданими даними
const existingUser = await User.findById(body.userId).exec()

if (!existingUser) {
// Повертаємо статус: Not Found - не знайдено
return res.status(404).send('Нажаль, жодного користувача із введеними даними не існує')
}

// Шукаємо шаблон за наданими даними
const existingTemplate = await Template.findOne({ name }).exec()

if (existingTemplate) {
// Повертаємо статус: Conflict - шаблон вже існує у базі даних
return res.status(409).send("Шаблон із введеним ім'ям вже існує")
}

let newTemplate
try {
// Додаємо шаблон до бази даних
newTemplate = await new Template(body.data).save()
```

```
} catch (e) {
return res.status(500).send(e)
}

// Додаємо шаблон до відповідного масиву користувачу
existingUser.templates.push(newTemplate)

// Зберігаємо змінені дані користувача
await existingUser.save()

// Повертаємо статус: OK - все добре
return res.status(200).send('Шаблон був успішно додан')
})
.put(async ({ query, body }, res) => {
const { name } = query
const { email, password } = body as IUserEntity

// Ініціалізуємо підключення до бази даних
await connectToDb()

// Шукаємо користувача за наданими даними
const existingUser = await User.findOne({ email, password }).exec()

if (!existingUser) {
// Повертаємо статус: Not Found - не знайдено
return res.status(404).send('Нажаль, жодного користувача із введеними даними не існує')
}

// Шукаємо шаблон за наданими даними
const existingTemplate = await Template.findOne({ name }).exec()

if (existingTemplate) {
// Повертаємо статус: Conflict - шаблон вже існує у базі даних
return res.status(409).send("Шаблон із введеним ім'ям вже існує")
}

// Додаємо шаблон до бази даних
const newTemplate = await new Template(query).save()

// Додаємо шаблон до відповідного масиву користувачу
existingUser.templates.push(newTemplate)
```

```
// Зберігаємо змінені дані користувача
await existingUser.save()

// Повертаємо статус: OK - все добре
return res.status(200).send('Шаблон був успішно додан')
})

router.route('/templates/download/:templateId').get(async ({ params }, res) => {
const { templateId } = params

// Ініціалізуємо підключення до бази даних
await connectToDb()

// Шукаємо користувача за наданими даними
const existingTemplate = await Template.findById(templateId).exec()

if (!existingTemplate) {
// Повертаємо статус: Not Found - не знайдено
return res.status(404).send('Нажаль, жодного шаблону із введеними даними не існує')
}

// Заповнюємо поле із шаблонами існуючими за їхніми "_id"
const { packages, _id, __v, ...populatedTemplate } = await (
await existingTemplate.populate('packages')
).toJSON()
var text = 'Hello world!'
res.attachment('filename.txt')
res.type('txt')
res.send(text)
// res.setHeader('Content-type', 'application/octet-stream')
// res.setHeader('Content-disposition', `attachment; filename=${existingTemplate.name}.json`)
// res.status(200).send(JSON.stringify({ ...populatedTemplate, dependencies: packages }))

// const zip = new JSZip()
// zip.file('s.txt', 'sss')

// // const zipToSend = Buffer.from(base64, 'base64')
// zip.generateAsync({ type: 'base64' }).then((base64) => {
//   let zip = Buffer.from(base64, 'base64')
//   res.writeHead(200, {
```

```
//     'Content-Type': ['application/zip'],
//     'Content-disposition': `attachment; filename=${populatedTemplate.name}.zip`,
//   })
//   res.end(zip)
// })
})

router
.route('/templates/:templateId')
.get(async ({ params }, res) => {
const { templateId } = params

// Ініціалізуємо підключення до бази даних
await connectToDb()

// Шукаємо користувача за наданими даними
const existingTemplate = await Template.findById(templateId).exec()

if (!existingTemplate) {
// Повертаємо статус: Not Found - не знайдено
return res.status(404).send('Нажаль, жодного шаблону із введеними даними не існує')
}

// Заповнюємо поле із шаблонами існуючими за їхніми "_id"
const populatedTemplate = await existingTemplate.populate('packages')

// Повертаємо статус: OK - все добре
return res.status(200).send(populatedTemplate)
})
.delete(async ({ params }, res) => {
const { templateId } = params

// Ініціалізуємо підключення до бази даних
await connectToDb()

// Шукаємо користувача за наданими даними
const existingTemplate = await Template.findById(templateId).exec()

if (!existingTemplate) {
// Повертаємо статус: Not Found - не знайдено
return res.status(404).send('Нажаль, жодного шаблону із введеними даними не існує')
```

```
}

// Заповнюємо поле із шаблонами існуючими за їхніми "_id"
await existingTemplate.delete()

// Повертаємо статус: OK - все добре
return res.status(200).send(templateId)
})

export { router as templates }

import express from 'express'
import connectToDb from '../db/connectToDb'
import { IUserEntity } from '../db/types'
import User from '../db/User'

/**
* Обробка шляхів пов'язаних із користувачами
*/

const router = express.Router()

router
.route('/users')
.get(async (req, res) => {
// Ініціалізуємо підключення до бази даних
await connectToDb()

// Запитуємо всіх користувачів, але не включаємо поле з паролем
const allUsers = await User.find({}, { password: 0 })

if (allUsers.length < 0) {
// Повертаємо статус: Not Found - Нічого не знайдено
return res.status(404).send('Наразі жодного користувача до бази даних не додано')
}

// Повертаємо статус: OK - Все добре
return res.status(200).send(allUsers)
})
.post(async ({ body }, res) => {
const { email } = body as Pick<IUserEntity, 'email'>
```

```
// Ініціалізуємо підключення до бази даних
await connectToDb()

// Шукаємо користувача за наданими даними
const existingUser = await User.findOne({ email }).exec()

if (existingUser) {
// Повертаємо статус: Conflict - користувач вже існує у базі даних
return res.status(409).send('Користувач із введеними даними вже існує')
}

/**
* У разі відсутності користувача із ідентичними властивостями:
* Ім'я, Прізвище та Електронна пошта, створюємо нову сутність користувача
* із переданими даними та зберігаємо її у базі даних із відповідною сигнатурою.
* Інакше - передаємо у відповідь помідомлення про помилку.
*/
try {
// Додаємо користувача до бази даних
const user = await new User(body).save()

// Повертаємо статус: Created - користувач був доданий до бази даних
return res.status(201).send(user)
} catch (err) {
return res.status(400).send(err.message)
}
})

// Авторизація користувачів
router.route('/users/login').post(async ({ body }, res) => {
const { email, password } = body as Pick<IUserEntity, 'email' | 'password'>

// Ініціалізуємо підключення до бази даних
await connectToDb()

// Шукаємо користувача за наданими даними
const existingUser = await User.findOne({ email, password }).exec()

if (!existingUser) {
// Повертаємо статус: Unauthorized - потрібна аутентифікація
```

```
        return res.status(401).send('Нажаль, жодного користувача із введеними даними не існує')
        }

        // Задля безпеки, не передаємо пароль у відповідь
        const { password: passwordOfExistingUser, ...existingUserWithoutPassword } =
existingUser.toJSON()

        // Повертаємо статус: OK - все добре
        return res.status(200).send(existingUserWithoutPassword)
        })

        export { router as users }

        import express from 'express'
        import serverless from 'serverless-http'
        import middlewares from './middlewares'
        import notFoundHandler from './middlewares/notFoundHandler'
        import rootRouter from './routers'

        // Ініціалізуємо сервер Express
        const app = express()

        /**
        * Додаємо проміжне програмне забезпечення
        */

        app.use(middlewares)

        // Позначаємо шлях до Express REST API
        app.use('/api', rootRouter)

        // Обробка запитів неіснуючих шляхів
        app.use(notFoundHandler)

        /**
        * Сервер буде розгорнутий на Serverless Netlify Functions,
        * тому додаємо обгортку до експорту
        */

        export const handler = serverless(app as unknown as serverless.Application
```