Хмельницький національний університет
Факультет інформаційних технологій
Кафедра інженерії програмного забезпечення

КВАЛІФІКАЦІЙНА РОБОТА

Метод поєднання технологій Redux-Toolkit та
Redux-Saga для роботи з API вебресурсів

Рівень вищої освіти Другий (магістерський)

Галузь знань 12 «Інформаційні технології»

Спеціальність 121 «Інженерія програмного забезпечення»

Освітня програма Освітньо-професійна програма «Інженерія програмного забезпечення»

Шифр ДРІПЗ.190160.01.09.ПЗ

Виконав студент 2 курсу група ІПЗм-22-1 _____ Владислав СОБКО
Підпис Ім`я, ПРІЗВИЩЕ

Керівник д-р фіз.-мат. наук, професор _____ Леонід БЕДРАТЮК
Науковий ступінь, звання Підпис Ім`я, ПРІЗВИЩЕ

Нормоконтролер канд. техн.наук, доцент _____ Галина РАДЕЛЬЧУК
Підпис Ім`я, ПРІЗВИЩЕ

До захисту допускаю:
Завідувач кафедри інженерії
програмного забезпечення _____ Леонід БЕДРАТЮК
Підпис Ім`я, ПРІЗВИЩЕ

7 грудня 2023 р.

Хмельницький 2023

ХМЕЛЬНИЦЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ

Факультет Інформаційних технологій
Кафедра Інженерії програмного забезпечення
Рівень вищої освіти Другий (магістерський)
Галузь знань 12 «Інформаційні технології»
Спеціальність 121 «Інженерія програмного забезпечення»
Освітня програма Освітньо-професійна програма «Інженерія програмного забезпечення»

ЗАТВЕРДЖУЮ
Завідувач кафедри
Л. П. Бедратюк
01 09 2023 р.

ЗАВДАННЯ
НА КВАЛІФІКАЦІЙНУ РОБОТУ

Собку Владиславу Вадимовичу
Прізвище, ім'я, по батькові студента

1. Тема проєкту (роботи) Метод поєднання технологій Redux-Toolkit та Redux-Saga для роботи з API вебресурсів

Керівник проєкту (роботи) Бедратюк Леонід Петрович, д-р фіз.-мат. наук, професор
Прізвище, ім'я, по батькові, науковий ступінь, вчене звання

Затверджена наказом ректора університету від 15.08.2023 р. № 30
2. Строк подання студентом проєкту (роботи) на кафедру 01.12.2023 р.
3. Вихідні дані до проєкту (роботи) Матеріали науково-дослідної практики
4. Зміст пояснювальної записки (перелік питань, які потрібно розробити)
1 Дослідження предметної області та постановка задачі
2 Концепції, моделі та методи вирішення задачі
3 Технології вирішення задачі
4 Реалізація та тестування програмного засобу
5. Перелік графічного матеріалу (із зазначенням обов'язкових креслень)
Презентаційні матеріали (слайди)

6. Консультанти розділів кваліфікаційної роботи

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		завдання видав	завдання прийняв
Антиплагіат	Форкун Ю.В., доцент	23.11.2023 р.	01.12.2023 р.
Нормоконтроль	Радельчук Г.І., доцент	23.11.2023 р.	01.12.2023 р.

7. Дата видачі завдання «01» вересня 2023 р.

КАЛЕНДАРНИЙ ПЛАН

Назва етапів (розділів) кваліфікаційної роботи	Строк виконання етапів проєкту (роботи)	Примітка
1 Вивчення предметної області; формулювання мети та задач дослідження; визначення об'єкта та предмета дослідження; формування логістичної структури кваліфікаційної роботи	01.09-10.09.2023	
2 Робота над розділом 1 кваліфікаційної роботи – вивчення літературнихта Інтернет-джерел; аналіз відомих моделей, методів та засобів за темою роботи; визначення методологічних підходів до вирішення задачі; висновки до розділу та постановка задач дослідження	11.09-25.09.2023	
3 Робота над розділом 2 кваліфікаційної роботи – розробка моделей,методів та алгоритмів вирішення задачі; висновки до розділу	26.09-10.10.2023	
4 Робота над науковими статтями	11.10-30.10.2023	
5 Робота над розділом 3 кваліфікаційної роботи – розробка інформаційноїтехнології вирішення задачі (аналіз вимог до програмного засобу та його проектування, аналіз та вибір засобів реалізації програмного засобу тощо); висновки до розділу	11.10-26.10.2023	
6 Робота над розділом 4 кваліфікаційної роботи – програмна реалізація спроектованих рішень, результати експериментів та їх аналіз; дослідження ефективності запропонованих рішень; висновки до розділу	27.10-17.11.2023	
7 Попередній захист кваліфікаційної роботи	Листопад (згідно графіка)	
8 Узгодження постановки задачі, отриманих результатів та висновків; написання вступу, загальних висновків, оформлення джерел посилання та додатків; оформлення пояснювальної записки та графічних матеріалів згідно вимог чинних стандартів	18.11-30.11.2023	
9 Перевірка роботи на наявність плагіату; нормоконтроль; брошурування пояснювальної записки; підготовка супровідних документів	01.12-04.12.2023	
10 Підготовка до захисту кваліфікаційної роботи	з 01.12.2023 р.	

Студент _____ Владислав. СОБКО
Підпис Ім'я, ПРІЗВИЩЕ

Керівник проекту (роботи) _____ Леонід БЕДРАТЮК
Підпис Ім'я, ПРІЗВИЩЕ

РЕФЕРАТ

Тема кваліфікаційної роботи: «Метод поєднання технологій Redux-Toolkit та Redux-Saga для роботи з API вебресурсів.»

Автор роботи: Собко Владислав Вадимович.

Керівник роботи: Бедратюк Леонід Петрович.

Пояснювальна записка: 130 с., 42 рис., 2 табл., 3 дод., 31 джерела.

API, ПАТЕРН, REDUX, REACT, ПРОГРАМНА АРХІТЕКТУРА, ПРОГРАМНИЙ ПРОДУКТ, ВЕБСЕРВІС.

Об'єкт дослідження: технологічні підходи до розробки вебзастосунків з використанням інструментів Redux-Toolkit та Redux-Saga для роботи з API вебресурсів.

Мета дослідження: розробка та апробація методу поєднання технологій Redux-Toolkit та Redux-Saga для підвищення ефективності та зручності роботи з API вебресурсів в масштабованих вебзастосунках.

У роботі використані наступні методи дослідження та апаратура:

– аналіз літературних джерел та документації з використання Redux-Toolkit та Redux-Saga;

– експериментальне тестування розроблених вебзастосунків з використанням розглянутих технологій;

– порівняльний аналіз з існуючими підходами до роботи з API;

– програмування вебзастосунків на базі розглянутих технологій.

Проведено дослідження практичної застосовності поєднання Redux-Toolkit та Redux-Saga для роботи з API вебресурсів. Оцінено переваги цього підходу у порівнянні з традиційним використанням Redux та Redux-Saga окремо. Представлено приклади реалізації та інструкції щодо використання цього методу.

Розроблений метод дозволяє покращити структуру та ефективність вебзастосунків, спрощуючи управління станом застосунку та асинхронними запитами до API вебресурсів. Використання Redux-Toolkit сприяє зменшенню

зайвого коду та покращенню читабельності. Реалізація Redux-Saga дозволяє зручно обробляти асинхронні запити та виконувати складні сценарії управління станом.

Дослідження показує, що розроблений метод спрощує процес розробки вебзастосунків та допомагає підвищити продуктивність розробників. Результати апробації підтверджують покращення продуктивності та зменшення помилок у вебзастосунках, розроблених за допомогою поєднання Redux-Toolkit та Redux-Saga. Цей підхід може бути рекомендований для використання в розробці вебзастосунків, особливо в масштабованих проектах.

Обсяг проведеного дослідження та розробленого методу підтверджує його значущість і внесок у покращення якості розробки вебзастосунків.

Підпис

01.12.2023

Дата

ABSTRACT

Master’s thesis: «Method of integrating Redux-Toolkit and Redux-Saga technologies for working with web resource APIs».

Author: Sobko Vladyslav.

Head of research: Bedratyuk Leonid.

Master’s thesis consists of: 130 p., 42 pc., 2 tb., 3 add., 31 srs.

API, PATTERN, REDUX, REACT, SOFTWARE ARCHITECTURE, SOFTWARE PRODUCT, WEB SERVICE.

Research Object: Technological approaches to web application development using Redux-Toolkit and Redux-Saga tools for working with web resource APIs.

Research Objective: Development and validation of a method that combines Redux-Toolkit and Redux-Saga technologies to improve the efficiency and convenience of working with web resource APIs in large-scale web applications.

The following research methods and equipment were used in this work: analysis of literature sources and documentation related to the use of Redux-Toolkit and Redux-Saga;

– experimental testing of web applications developed using the discussed technologies;

– comparative analysis with existing approaches to API interaction;

– programming of web applications based on the discussed technologies.

Practical applicability of the combination of Redux-Toolkit and Redux-Saga for web resource API interaction was explored. The advantages of this approach in comparison to the traditional use of Redux and Redux-Saga separately were assessed. Examples of implementation and instructions for using this method are provided.

The developed method enhances the structure and efficiency of web applications by simplifying state management and asynchronous API requests. The use of Redux-Toolkit reduces redundant code and improves code readability. The implementation of

Redux-Saga facilitates the handling of asynchronous requests and the execution of complex state management scenarios.

The research shows that the developed method simplifies the web application development process and aids in improving developer productivity. The results of the validation confirm increased productivity and reduced errors in web applications developed using the combination of Redux-Toolkit and Redux-Saga. This approach can be recommended for use in web application development, particularly in large-scale projects.

The scope of the research and the developed method underscores its significance and contribution to enhancing the quality of web application development.

Signature

01.12.2023

Date

ЗМІСТ

ПЕРЕЛІК СКОРОЧЕНЬ

БД	–	база даних
ПЗ	–	програмний засіб
ПС	–	програмна система
СКБД	–	система керування базами даних
DNS	–	Domain Name System
HTTP	–	Hyper Text Transfer Protocol
API	–	Application Programming Interface
REST	–	Representational State Transfer
SOAP	–	Simple Object Access Protocol

ВСТУП

Сучасний світ програмної інженерії невпинно розвивається, створюючи нові виклики та можливості для розробників. В епоху інформаційних технологій, коли вебзастосунки стають все більш функціональними та інтерактивними, важливість ефективного управління станом застосунків значно зростає. Redux, як визнаний інструментарій для управління станом у React-застосунках, пропонує структурований та передбачуваний спосіб управління даними. Разом з тим, у складних застосунках з'являється потреба в асинхронних операціях та побічних ефектах, які Redux сам по собі не вирішує, тому розробники часто вдаються до використання додаткових мідлварів, як-от Redux-Saga − бібліотека, яка дозволяє легко та ефективно управляти асинхронними операціями та побічними ефектами.

Ця кваліфікаційна робота присвячена дослідженню методу поєднання двох потужних технологій: Redux-Toolkit та Redux-Saga. Redux-Toolkit спрощує процес налаштування та використання Redux, надаючи більш чистий та менш бойлерплейтний код, а Redux-Saga дозволяє елегантно управляти асинхронними діями, що забезпечує кращу обробку складних потоків даних та побічних ефектів.

Метою даної роботи є аналіз та розробка оптимальної методології використання цих двох технологій разом, що дозволить розробникам не тільки підвищити ефективність розробки та легкість підтримки коду, але й забезпечить більшу масштабованість та гнучкість у роботі з API вебресурсів. Таке поєднання суттєво полегшить реалізацію складних вимог до стану застосунків, спростити управління потоками даних та оптимізувати взаємодію з серверною частиною.

Для досягнення мети розробки оптимальної методології використання технологій Redux-Toolkit та Redux-Saga разом, основні завдання включають:

– дослідження та аналіз існуючих патернів управління станом − вивчення найкращих практик та патернів, що використовуються у сучасній розробці вебзастосунків для управління станом і асинхронними операціями;

– аналіз можливостей Redux-Toolkit та Redux-Saga − оцінка функціональності, яку пропонують обидві технології, та їх придатності для різних типів задач, що пов’язані з взаємодією з API;

– експериментування з інтеграцією технологій − реалізація прототипів, що використовують Redux-Toolkit і Redux-Saga разом, для практичного вивчення їх взаємодії;

– оптимізація процесу розробки − розробка методів та інструментів для спрощення процесу кодування, зменшення кількості бойлерплейту, та поліпшення структури проекту;

– розробка єдиної архітектури − створення архітектурного шаблону, що поєднує Redux-Toolkit для управління станом та Redux-Saga для обробки побічних ефектів;

– підвищення масштабованості та гнучкості − впровадження практик, які дозволять легко масштабувати застосунок та ефективно адаптуватися до змін в API вебресурсів;

– тестування та валідація − проведення комплексного тестування для перевірки стабільності та ефективності розробленої методології;

– аналіз продуктивності − оцінка впливу впроваджених рішень на продуктивність застосунку та його відгук на користувацькі запити;

– забезпечення готовності до змін − розробка системи, що зможе ефективно адаптуватися до змін в специфікації API без необхідності переписування значної частини коду.

Розв’язання цих завдань дозволить розробникам створити міцну основу для використання Redux-Toolkit та Redux-Saga, що сприятиме кращій організації коду, поліпшенню процесів розробки та забезпеченню гнучкості та масштабованості взаємодії з вебресурсами.

Об'єктом дослідження в темі «Метод поєднання технологій Redux-Toolkit та Redux-Saga для роботи з API вебресурсів» є процеси і методики розробки клієнтської частини вебзастосунків, а саме управління станом та асинхронними операціями, що взаємодіють з вебAPI.

Предметом дослідження є конкретні механізми та підходи до інтеграції Redux-Toolkit і Redux-Saga як інструментів для створення ефективної архітектури управління станом вебзастосунків, оптимізації процесу роботи з серверними даними через API та обробки побічних ефектів, зокрема:

– розробка акшенів, редюсерів та селекторів за допомогою Redux-Toolkit;

– використання мідлвари Redux-Saga для управління асинхронними запитами до API;

– створення ефективних Saga-воркерів та вотчерів для оптимізації асинхронної логіки;

– способи зменшення бойлерплейту та покращення читабельності коду за допомогою Redux-Toolkit;

– техніки забезпечення стабільності та надійності асинхронної взаємодії з API;

– аналіз та оптимізація продуктивності вебзастосунків при використанні цих технологій.

В ході цієї роботи буде проведено аналіз сучасних підходів до управління станом у великих вебзастосунках, визначено ключові вимоги до систем управління станом, а також розглянуто основні проблеми, з якими розробники зустрічаються при інтеграції Redux-Toolkit та Redux-Saga. Буде розроблено прототип вебзастосунку, який демонструє практичне застосування обраної методології, і проведено аналіз його ефективності.

Робота містить теоретичну та практичну частини, які об'єднують глибокий теоретичний аналіз з реальною практичною розробкою. Значна увага приділяється визначенню кращих практик та патернів проектування, які можуть бути корисні для розробників, що прагнуть оптимізувати процес роботи зі станом великих та складних вебзастосунків.

Таким чином, ця кваліфікаційна робота сприяє глибшому розумінню та ефективному використанню сучасних інструментів управління станом у вебзастосунках, а також вносить свій вклад у вдосконалення бестпрактисів розробки програмного забезпечення.

1 ДОСЛІДЖЕННЯ ПРЕДМЕТНОЇ ОБЛАСТІ ТА ПОСТАНОВКА ЗАДАЧІ

1.1 Аналіз предметної області, останніх досліджень та джерел

Модернізація веброзробки вимагає впровадження передових інструментів для поліпшення ефективності та удосконалення коду. Розглянемо сучасні підходи до використання технологій, як-от Redux-Toolkit та Redux-Saga, в роботі з вебресурсами. Redux-Toolkit та Redux-Saga є популярними інструментами в екосистемі React для управління станом застосунку та ефектами (побічними ефектами) відповідно. Redux-Toolkit є розширенням для Redux, що забезпечує більш зручну та лаконічну розробку завдяки вбудованим утилітам, таким як управління станом. Цей інструмент включає в себе декілька корисних функцій, таких як createSlice та createAsyncThunk, що дозволяють оптимізувати та спрощувати роботу зі станом і асинхронними діями. Redux-Toolkit спрощує роботу з Redux, надаючи набір інструментів, що зменшують кількість шаблонного коду та сприяють більш ефективній розробці. Redux-Saga, з іншого боку, є бібліотекою для ефективного управління асинхронними діями, які можуть включати доступ до API, і пропонує механізми для розробки складних сценаріїв бізнес-логіки. Ця бібліотека використовує генератори JavaScript для управління асинхронними діями, такими як запити до API.

Застосування Redux-Saga дає змогу створити ефективні механізми для управління запитами до API, включно з GET, POST, PUT і DELETE. Redux-Toolkit спрощує створення структурованого стану та взаємодію з API.

Роль Redux-Toolkit полягає у створенні легкої конфігурації загального сховища даних ПС, організації редюсерів, дій та селекторів. Redux-Saga, у свою чергу, дозволяє відділити побічні ефекти від логіки редюсерів, спрощуючи тестування та керування складними потоками даних.

Інтеграція Redux-Toolkit та Redux-Saga в процес розробки відкриває нові горизонти для роботи з об'ємними даними і API, покращуючи взаємодію з сервером та оптимізуючи обробку результатів.

Актуальні дослідження у цій області часто фокусуються на підвищенні продуктивності та зменшенні складності управління станом застосунків SPA (Single Page Applications). Останні досягнення та прогрес у використанні Redux-Toolkit і Redux-Saga сприяють розширенню можливостей розробників і підвищенню якості продукту. Основна увага приділяється таким аспектам, як:

– покращення читабельності коду та зменшення кількості boilerplate коду;

– ефективне управління асинхронними процесами;

– спрощення тестування компонентів та логіки управління станом;

– інтеграція з іншими бібліотеками та фреймворками;

– автоматизація рутинних задач за допомогою утиліт та мідлварів.

Також недавні дослідження щодо використання Redux-Toolkit та Redux-Saga виявили, що ці інструменти мають значний потенціал для підвищення продуктивності розробників і забезпечення стабільності застосунків. Стало відомо, що використання Redux-Toolkit допомагає зменшити обсяг коду на приблизно 30%, що сприяє більш ефективній роботі розробників. Крім того, застосування Redux-Saga дозволяє ефективніше управляти асинхронними діями та легше впоратися з помилками завдяки створенню спеціалізованих сценаріїв для взаємодій з сервером.

Рекомендації, що випливають з цих досліджень, включають наступне:

– застосування Redux-Toolkit для оптимізації роботи із станом застосунку та зниження складності коду, що потрібен для написання;

– використання Redux-Saga для кращої обробки асинхронних операцій, що надає можливість створювати більш чистий і зрозумілий код сценаріїв взаємодії з API;

– інтеграція createAsyncThunk з Redux-Toolkit для простішої обробки асинхронних запитів, що дозволяє розробникам легко обробляти помилки і статуси запитів.

Ці знахідки допомагають розробникам вдосконалити їх підходи до роботи з API вебресурсів та сприяють розвитку їхніх навичок у сфері веброзробки. Це

важливо для створення ефективних, надійних застосунків, що можуть обробляти складні взаємодії з користувачами та серверами.

Існують також розширені практики та підходи, наприклад, атомарний управління станом, принципи реактивного програмування та патерни проектування для оптимізації взаємодії з API.

Основними джерелами інформації є офіційна документація Redux (redux.js.org) та Redux-Saga (redux-saga.js.org), а також велика кількість статей, блогів та відеоматеріалів від розробників та спільноти. Додаткові інформаційні ресурси включають Stack Overflow для розв'язання проблем, та Medium, де експерти діляться своїми знаннями та досвідом. Досвідчені розробники також можуть поділитися реальними проектами та фрагментами коду на GitHub, що стане цінним доповненням до теоретичних знань. Dev.to та Hashnode.com також є популярними платформами, де розробники діляться своїми досвідами та найкращими практиками. Великі технічні конференції, такі як React Conf, часто включають доповіді про передові методики роботи з Redux-Toolkit і Redux-Saga. Блоги, такі як Airbnb Engineering & Data Science, Facebook Engineering, Netflix Technology Blog часто публікують статті про внутрішнє використання та оптимізацію цих технологій. Udemy, Coursera та Egghead.io надають платні та безплатні курси, де досвідчені розробники ведуть детальні лекції про Redux-Toolkit та Redux-Saga.

Метод поєднання Redux-Toolkit та Redux-Saga для роботи з API вебресурсів може значно поліпшити ефективність розробки та якість кінцевого продукту. Сучасні дослідження та розробки в цій області продовжують зосереджуватися на покращенні взаємодії між різними частинами застосунку та зовнішніми сервісами. Навчання та використання Redux-Toolkit і Redux-Saga є важливими для розробників, які хочуть підвищити свою кваліфікацію у сфері веброзробки та створювати продуктивні вебсервіси. Постійна еволюція технологій і взаємодія зі спільнотою сприяють вдосконаленню практик та підходів до управління станом великих застосунків.

1.2 Аналіз існуючих методів та засобів для роботи з API вебресурсів

Аналіз методів і засобів для роботи з API вебресурсів вимагає розуміння основних концепцій та доступних інструментів. API (Application Programming Interface) − це набір правил і протоколів для створення та інтеграції програмного забезпечення, що дозволяє різним програмам ефективно комунікувати один з одним.

Основні види API:

– REST (Representational State Transfer);
– SOAP (Simple Object Access Protocol);
– GraphQL;
– WebSocket.

Архітектурний стиль Representational State Transfer (REST) є підходом до розробки вебзастосунків, що базується на кількох ключових принципах. Даний підхід використовує стандартні HTTP-методи, такі як GET, POST, PUT, DELETE та інші, для взаємодії з ресурсами, що ідентифікуються за допомогою унікальних URI (уніфікованих ідентифікаторів ресурсу).

У даній архітектурі, кожен ресурс представляє собою індивідуальний об'єкт чи набір даних, доступ до якого здійснюється через відповідний URI. Важливою характеристикою є те, що REST не зберігає жодної інформації про стан між окремими запитами, що робить його «stateless» − без стану.

Для обміну даними REST часто використовує формати JSON або XML, які є популярними серед розробників для передачі структурованої інформації через мережу. Це дозволяє системам взаємодіяти та обмінюватися даними незалежно від мов програмування та платформи, що використовується.

Загалом, REST забезпечує простий та ефективний спосіб створення вебсервісів, де ресурси, їхні представлення та взаємодія з ними організовані згідно з чіткими принципами, що сприяє розширюваності та доступності систем.

Протокол Simple Object Access Protocol (SOAP) є одним із стандартів для обміну повідомленнями між компонентами програмного забезпечення. SOAP використовує мову розмітки XML для форматування та передачі структурованих даних між різними системами.

Однією з ключових характеристик SOAP є його строге дотримання стандартів безпеки та повідомлень. Він має вбудовані механізми для автентифікації, шифрування та контролю цілісності даних, що дозволяє забезпечити безпеку під час обміну інформацією між різними системами.

Однак SOAP часто сприймається як більш складний для розуміння та інтеграції порівняно з іншими протоколами, такими як REST. Це зумовлено великою кількістю деталей та нормативних вимог, які потрібно враховувати при роботі з SOAP-протоколом. Незважаючи на це, його стандарти забезпечують надійність та безпеку під час обміну даними між різними системами.

GraphQL − це запитова мова та середовище виконання, розроблене компанією Facebook, яке надає можливість клієнтам визначати структуру запитуваних даних. На відміну від традиційних REST API, де сервер визначає структуру відповіді, GraphQL дозволяє клієнтам зазначати, які саме поля та дані вони хочуть отримати, що робить запити більш гнучкими та специфічними.

Однією з переваг GraphQL є зменшення кількості запитів до сервера та об'єму пересиланих даних. Клієнти можуть запросити лише необхідні для них дані, отримуючи лише ту інформацію, яка їм потрібна, що може суттєво зменшити обсяг передачі даних та підвищити швидкість відповіді від сервера.

Також GraphQL сприяє покращенню гнучкості та ефективності взаємодії з API. Він дозволяє клієнтам отримувати точно той тип даних, який їм необхідний, і уникати отримання зайвої інформації, що полегшує розробку та підтримку клієнтських застосунків.

Усі ці функції роблять GraphQL потужним інструментом для розробників, які прагнуть покращити продуктивність, швидкість та ефективність своїх вебзастосунків та API.

WebSocket − це технологія, яка надає можливість для забезпечення двостороннього зв'язку між вебклієнтом і сервером в реальному часі через одне постійне з'єднання. На відміну від традиційних HTTP-запитів, які мають характер запит-відповідь (request-response), WebSocket дозволяє встановлювати постійне з'єднання, що дає змогу обмінюватися даними між клієнтом та сервером в обидва напрямки без перерви.

Ця технологія особливо корисна для вебзастосунків, які потребують швидкого та миттєвого обміну даними в реальному часі. Наприклад, чати, онлайн-ігри, системи сповіщень та інші сценарії, де важливо отримувати оновлення чи повідомлення миттєво, без затримок.

WebSocket дозволяє вебзастосункам підтримувати постійний потік даних між клієнтом та сервером, що сприяє зменшенню затримок, покращенню продуктивності та взаємодії між користувачами в реальному часі. Це дозволяє створювати більш динамічні та реактивні вебзастосунки, які забезпечують більш зручний та інтерактивний користувацький досвід.

Переваги REST:

– універсальність: REST використовує стандартні HTTP-операції, що робить його легким для розуміння та інтеграції;

– безстаничність**:** кожен запит незалежний, що полегшує масштабування;

– формати даних**:** REST може використовувати JSON, XML, YAML або інші легкі формати обміну даними;

Недоліки REST:

– управління станом: для операцій, які вимагають багатьох послідовних запитів, управління станом може бути складним;

– overfetching/underfetching**:** клієнти можуть отримувати більше даних, ніж потрібно, або їм може бракувати даних, що вимагає додаткових запитів;

Переваги SOAP:

– стандарти безпеки**:** SOAP підтримує WS-Security, що забезпечує високий рівень безпеки;

– транзакції**:** підтримка складних транзакцій і надійної обробки помилок;

Недоліки SOAP:

– складність**:** SOAP часто вимагає більшої кількості коду та більш складної логіки, ніж REST;

– продуктивність**:** XML, який використовує SOAP, може бути більш громіздким, ніж JSON, що використовується в REST, що призводить до більших об'ємів даних для передачі;

Переваги GraphQL**:**

– гнучкі запити: клієнти можуть запитувати саме ті дані, які їм потрібні, що знижує навантаження на мережу;

– оптимізація запитів: здатність зробити складні запити за один раунд-тріп;

Недоліки GraphQL**:**

– складність кешування: через динамічність запитів, кешування може бути складнішим;

– висока навантаження на сервер: комплексні запити можуть вимагати значних ресурсів сервера для обробки;

Переваги WebSocket:

– двостороння комунікація: підтримує постійне з'єднання для обміну даними в реальному часі;

– продуктивність: зменшує затримки, порівняно з постійним створенням нових HTTP-з'єднань;

Недоліки WebSocket:

– складність управління: потребує більш складного управління підключеннями та станом з'єднання;

– підтримка: може бути неповною або неоднозначною в різних браузерах.

Інструменти для роботи з API.

– клієнтські бібліотеки та фреймворки: Axios, Fetch API, jQuery AJAX;

– інструменти для тестування API: Postman та cURL;

– платформи для управління API: Apigee, AWS API Gateway.

Переваги Postman:

– інтерфейс: легкий та інтуїтивно-зрозумілий у використанні для тестування та відладки API;

– автоматизація тестування: підтримка автоматичних тестів та сценаріїв;

Недоліки Postman:

– ресурсоємність: може споживати багато системних ресурсів;

Переваги cURL**:**

– гнучкість: дозволяє виконувати запити до різних протоколів;

– широкий діапазон використання: підтримка багатьох операційних систем;

Недоліки cURL**:**

– командний рядок: може бути складним для користувачів без досвіду роботи з командним рядком;

– Swagger або OpenAPI: для тестування та документації REST API;

– Insomnia: альтернатива Postman з меншим споживанням ресурсів;

Переваги Apigee**:**

– функціональність: великий набір можливостей для аналітики, моніторингу та оптимізації API;

– масштабування: підтримка великої кількості API і трафіку;

Недоліки Apigee**:**

– вартість: може бути дорогим для малого бізнесу або стартапів;

Переваги AWS API Gateway:

– інтеграція з AWS: легке інтегрування з іншими сервісами AWS.

– масштабування**:** автоматично масштабується для підтримки збільшення кількості запитів.

Недоліки AWS API Gateway**:**

– комплексність: може вимагати часу для розуміння всіх можливостей та кращих практик.

– залежність від постачальника: тісно пов'язаний з екосистемою AWS, що може утруднити перенесення на інші платформи.

Кращі практики:

– аутентифікація та авторизація: використовувати стандарти OAuth, OpenID Connect або API ключі;

– кешування: зменшити навантаження на сервер і покращити час відповіді;

– обмеження частоти запитів (Rate Limiting): запобігання зловживанням і DDoS-атакам;

– версіонування: ясне управління версіями API для підтримки зворотної сумісності;

– документація: чітка та актуальна документація API, що полегшує інтеграцію і використання.

Застосування цих методів і засобів залежить від конкретного випадку та потреб проекту, а також від переваг та вимог розробників.

1.3 Методологічні підходи до роботи з API вебресурсів

Робота з API (Application Programming Interface) вебресурсів передбачає використання набору стандартизованих методів та протоколів для взаємодії з вебсервісами. API дозволяє програмам звертатися до служб і отримувати дані або виконувати певні дії без необхідності знання внутрішньої структури сервісу. Існує кілька методологічних підходів до роботи з API вебресурсів:

– опрацювання документації API: перш за все, необхідно детально вивчити документацію API, скільки документація містить інформацію про доступні методи, формат запитів, параметри, очікувані відповіді та можливі помилки;

– автентифікація та авторизація: розробка безпечної системи аутентифікації та авторизації є ключовою при роботі з API, найпоширеніші методи включають OAuth, API ключі, JWT (JSON Web Tokens);

– вибір підходящої архітектури: залежно від завдання, можна використовувати різні архітектурні стилі API, такі як REST, SOAP, GraphQL або gRPC;

– модульність: компонентна архітектура (React, Vue.js, Angular): сучасні фронтенд фреймворки та бібліотеки пропонують компонентний підхід, де кожен компонент взаємодіє з API незалежно, що підвищує повторне використання коду та полегшує тестування;

– асинхронність: Promises та Async/Await: ці конструкції дозволяють організувати асинхронні запити до API так, щоб вони не блокували рендеринг інтерфейсу;

– стейт-менеджмент: Redux, Vuex, NgRx: використання бібліотек для управління станом застосунку дозволяє централізовано керувати даними, отриманими з API, забезпечує їх синхронізацію між різними компонентами та відслідковування змін;

– робота з API-запитами: Fetch API, Axios: використання зовнішніх бібліотек для роботи з HTTP-запитами полегшує відправку запитів, обробку відповідей і конфігурацію необхідних заголовків;

– управління версіями: управління версіями API дозволяє забезпечити сумісність з існуючими клієнтськими застосунками під час оновлення або впровадження нових функцій;

– тестування: тестування API включає перевірку роботи кожного ендпойнта, валідацію відповідей і обробку помилок, для цього використовують спеціалізовані інструменти, такі як Postman, Swagger або власноруч написані скрипти;

– обробка помилок: коректне і чітке інформування про помилки − важливий аспект інтерфейсів API, статус-коди HTTP можуть допомогти клієнтам розуміти, чому запит не вдався;

– моніторинг і логування: ведення логів та моніторинг стану API дозволяє своєчасно виявляти та вирішувати проблеми, а також аналізувати поведінку користувачів для оптимізації сервісу;

– кешування: кешування даних може суттєво знизити навантаження на сервер та прискорити відгук API;

– обмеження частоти запитів: розробка систем контролю частоти запитів (rate limiting) захищає API від зловживань та DDoS-атак;

– сек'юрність: забезпечення безпеки API через застосування протоколу HTTPS, валідацію вводу, обмеження доступу та інші механізми безпеки є критично важливим.

Робота з API вебресурсів вимагає уважного планування, розуміння потреб бізнесу, технічної експертизи та постійного моніторингу та оновлення власних знань про новітні технології та кращі практики. Все це допомагає створити надійні, масштабовані та ефективні інтеграції з вебсервісами.

1.4 Висновки. Постановка задачі

В ході проведеного аналізу предметної області було досліджено останні наукові розробки та джерела, що стосуються використання API вебресурсів. Було виявлено, що сучасні вебзастосунки все більше покладаються на ефективний обмін даними з сервером, що робить оптимізацію роботи з API одним з ключових аспектів розробки. Аналіз існуючих методів і засобів показав, що інструменти, такі як Redux-Toolkit та Redux-Saga, є потужними засобами для управління станами великих вебзастосунків та асинхронними ефектами відповідно.

Методологічний аналіз підходів до роботи з API вебресурсів показав, що поєднання Redux-Toolkit та Redux-Saga може сприяти створенню більш ефективних, масштабованих та легко підтримуваних вебзастосунків. Redux-Toolkit забезпечує структурований та лаконічний спосіб управління станом застосунку, в той час як Redux-Saga дозволяє ефективно управляти складними асинхронними процесами та бічними ефектами.

Враховуючи проведений аналіз, можна зробити висновок, що розробка методу поєднання Redux-Toolkit та Redux-Saga для роботи з API вебресурсів може мати значний вплив на поліпшення процесу розробки вебзастосунків, зокрема:

- забезпечення більш чистої та підтримуваної кодової бази;

– зниження складності управління асинхронними операціями та станами;

– підвищення продуктивності розробників за рахунок зменшення кількості шаблонного коду;

– вдосконалення відповідності застосунків сучасним вимогам до швидкості відгуку та масштабованості.

На основі викладених висновків можна сформулювати наступну задачу:

Розробка методу поєднання технологій Redux-Toolkit та Redux-Saga для оптимізації роботи з API вебресурсів

Задача передбачає виконання наступних підзадач:

– розробка концептуальної моделі взаємодії Redux-Toolkit та Redux-Saga в контексті управління станом вебзастосунків та асинхронними операціями;

– аналіз та вибір найбільш ефективних практик використання цих технологій;

– проектування архітектури вебзастосунку, який інтегрує в себе Redux-Toolkit та Redux-Saga для роботи з API;

– реалізація прототипу вебзастосунку з використанням розробленого методу;

– тестування та оцінка ефективності методу на практиці;

– оформлення рекомендацій щодо використання поєднання Redux-Toolkit та Redux-Saga для розробників вебзастосунків.

Ціль даної роботи полягає у створенні методики, яка інтегрує Redux-Toolkit та Redux-Saga для оптимізації роботи з API вебсервісів. Це передбачає вдосконалення існуючих підходів, розроблення та тестування програмного продукту, заснованого на цій методиці.

Дослідження акцентує увагу на аналізі доступних рішень для взаємодії з API, визначенні ключових вимог до удосконаленої методики, як-то автоматичне скорочення кількості помилок та здатності адекватно реагувати на їх виникнення із встановленням причин.

У новоствореній методиці буде реалізовано такі новації:

– автоматичне розпізнавання помилок у процесі запитів до API, що знижує загальну кількість помилок;

– автоматичне повторення запитів до API у випадку помилок, що збільшує стійкість застосунку;

– введення модуля аудиту, який слідкує за успішністю запитів і збіром даних про помилки, що виникають.

Результати даного дослідження мають на меті спростити процес інтеграції з API вебресурсів, підвищити ефективність коду та забезпечити більшу адаптивність вебзастосунків до мінливих вимог користувачів та ринку.

2 КОНЦЕПЦІЇ, МОДЕЛІ ТА МЕТОДИ ВИРІШЕННЯ ЗАДАЧІ

2.1 Концепції програмних систем з використанням API

Клієнтська частина програмної системи − це та частина програми, з якою безпосередньо взаємодіє користувач. Вона включає в себе інтерфейси, графіку, а також клієнтську логіку, яка відповідає за обробку даних, введених користувачем, і відображення даних, отриманих з сервера або інших джерел.

У контексті фронтенду, API зазвичай використовується для комунікації з бекендом (серверною частиною) або з іншими сервісами, наприклад, соціальними мережами, платіжними системами тощо.

Основними концепціями фронтенду, які використовують API, є:

– запити до сервера;
– обробка відповідей;
– асинхронність;
– формати даних;
– аутентифікація та авторизація;
– крос-доменні запити;
– застосування фреймворків та бібліотек;
– WebSockets;
– GraphQL;

Запити до сервера: фронтенд використовує API для надсилання запитів до сервера. Це можуть бути запити на отримання даних (GET), відправлення нових даних (POST), оновлення існуючих даних (PUT/PATCH) або видалення даних (DELETE).

Проблема: помилки мережі або довгий час відповіді сервера.

Рішення: використання таймаутів для запитів, повторні спроби запитів, відображення повідомлень про помилки або статусів для користувача.

На рисунку 2.1 показана схема роботи концепції запитів до сервера.

Схема показує, що фронтенд використовує API для надсилання запиту до сервера. Запит містить інформацію про тип запиту, ресурс, до якого потрібно

звернутися, і будь-які дані, які потрібно передати. Сервер отримує запит і обробляє його. У відповідь сервер повертає дані фронтенду.

GET-запит використовується для отримання даних з ресурсу. Наприклад, вебсайт може використовувати GET-запит, щоб отримати список продуктів з бази даних. GET-запит використовується для отримання ресурсів з сервера, таких як HTML-сторінки, зображення, файли, або будь-які дані, які доступні через веб. Під час виконання GET-запиту дані не модифікуються на сервері, оскільки цей метод призначений лише для отримання інформації;

POST-запит використовується для відправлення нових даних на сервер. Наприклад, вебсайт може використовувати POST-запит, щоб додати новий продукт до бази даних;

PUT/PATCH-запит використовується для оновлення існуючих даних на сервері. Наприклад, вебсайт може використовувати PUT-запит, щоб змінити ціну продукту в базі даних;

DELETE-запит використовується для видалення даних з сервера. Наприклад, вебсайт може використовувати DELETE-запит, щоб видалити продукт з бази даних.

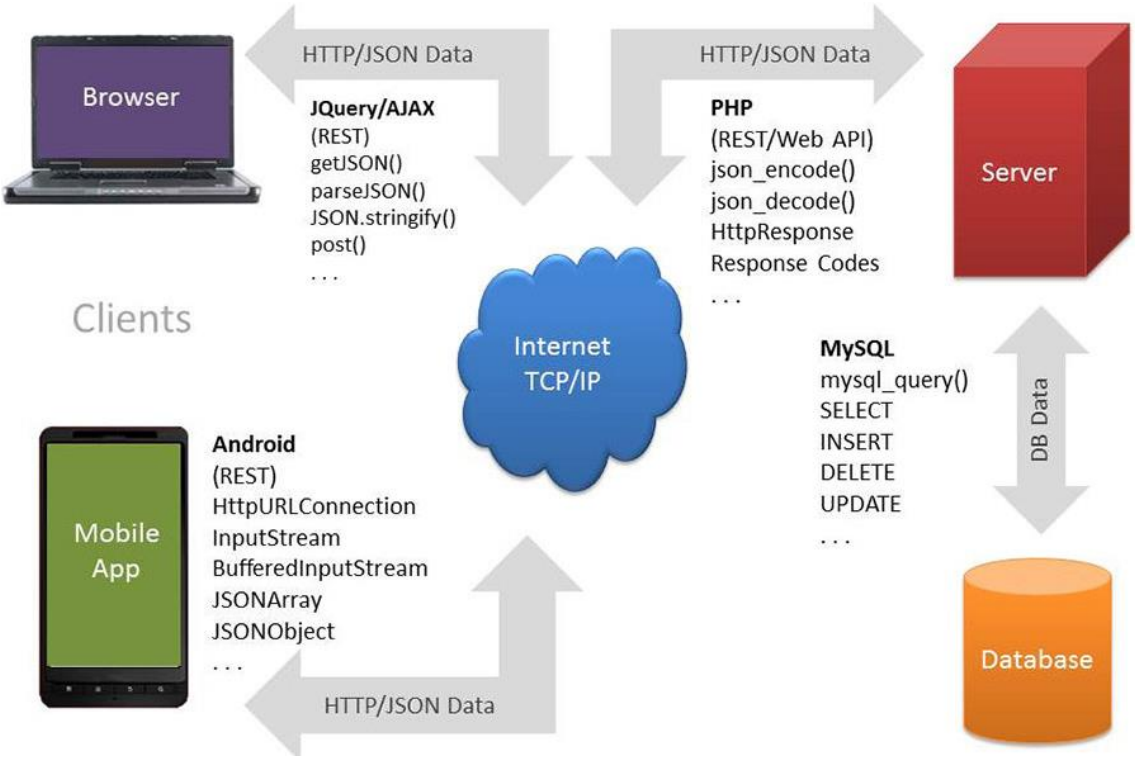


Рисунок 2.1 − Схема роботи концепції запитів до сервера

Обробка відповідей: після відправлення запиту, фронтенд очікує і обробляє та виводить на екран відповідь від сервера в зрозумілому для користувача форматі. Відповідь може містити дані, статус операції, а також повідомлення про помилки, тощо.

Проблема:

Непередбачувані відповіді сервера або формати даних.

Рішення:

Валідація даних перед їх використанням, обробка помилок сервера, використання типізації даних (наприклад, TypeScript).

На рисунку 2.2 показана схема отримання успішної відповіді від сервера, а на малюнку 2.3 – варіант, коли сервер відповів з помилкою

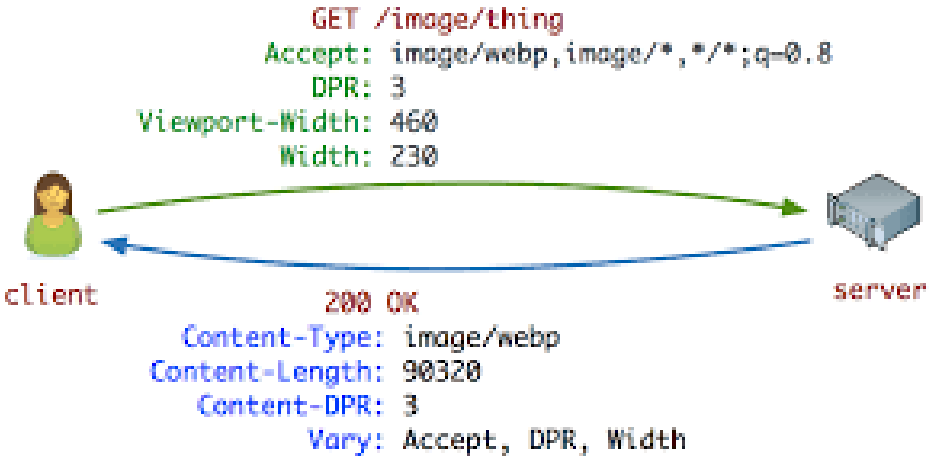


Рисунок 2.2 – Успішна відповідь від сервера

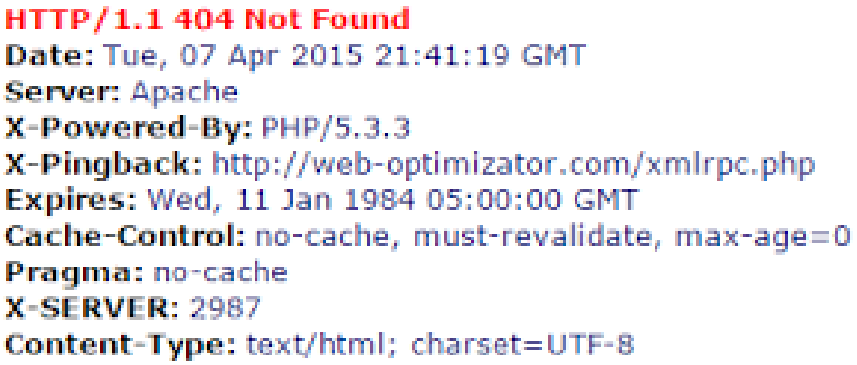


Рисунок 2.3 – Відповідь від сервера з помилкою

Асинхронність: більшість запитів API в фронтенді виконуються асинхронно, що означає, що користувачеві інтерфейс залишається відгуковим, поки чекає на відповідь від сервера. Для цього використовуються такі поняття, як проміси (promises), async/await тощо.

Проблема:

«Callback Hell» або складність управління асинхронним кодом.

Рішення:

Використання промісів (promises), async/await для спрощення та структуризації асинхронного коду.

На рисунку 2.4 показана схема роботи концепції асинхронності.

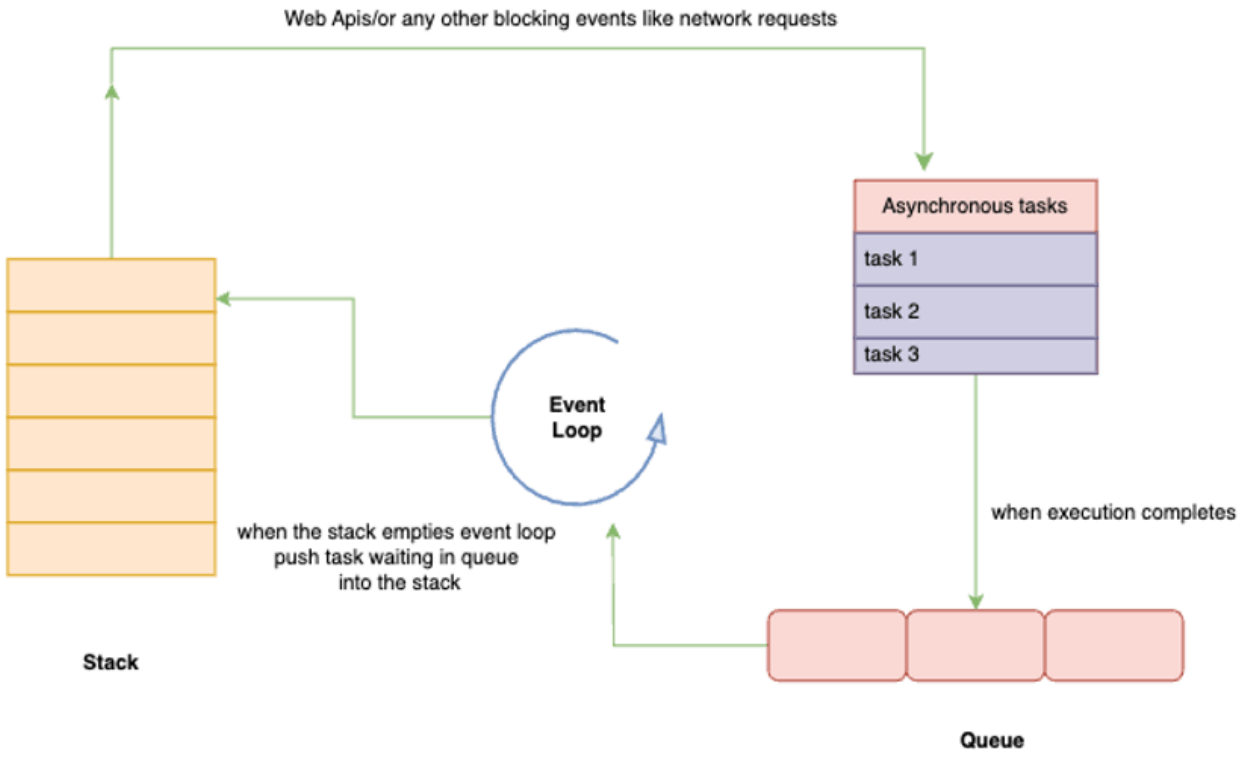


Рисунок 2.4 – Схема роботи концепції асинхронності

Формати даних: дані, які передаються між фронтендом і бекендом через API, зазвичай мають структурований формат, найчастіше JSON або XML.

Проблема:

Несумісність форматів даних між фронтендом і бекендом.

Рішення:

Узгодження форматів даних, використання серіалізації/десеріалізації даних (наприклад, JSON.parse() або JSON.stringify()).

На рисунку 2.5 показані найбільш розповсюджені формати даних, а саме JSON та XML і можна побачити різницю між ними.

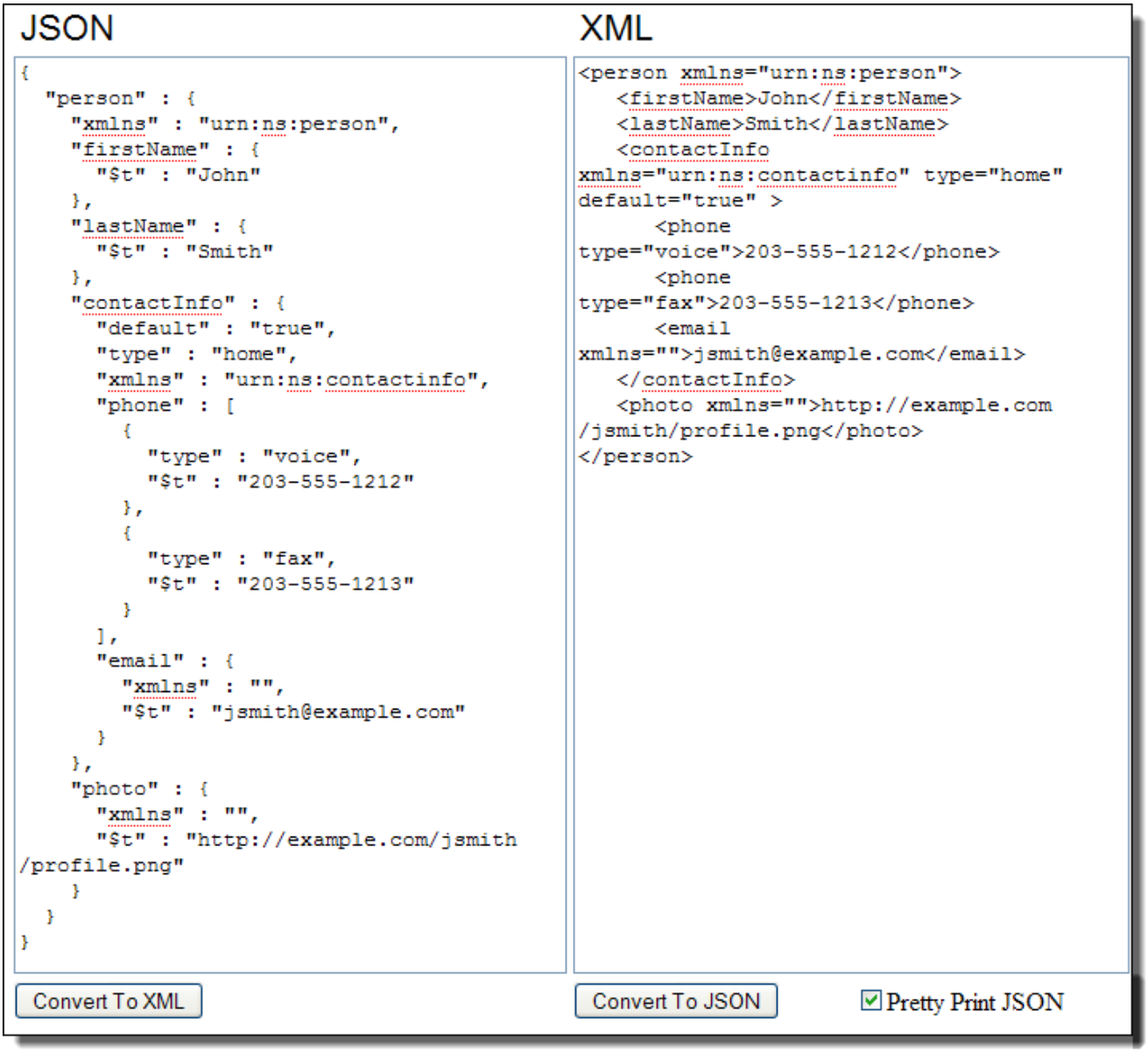


Рисунок 2.5 – Схема роботи концепції форматів даних

Аутентифікація та авторизація: для забезпечення безпеки, API часто вимагають аутентифікації (підтвердження особи користувача) та авторизації (перевірки прав на виконання певних дій). Це може бути реалізовано через токени, OAuth, сесії тощо.

Проблема:

Витік або крадіжка облікових даних, небезпека CSRF або XSS атак.

Рішення:

Використання HTTPS, токенів, які зберігаються безпечно (наприклад, в httpOnly cookies), впровадження політик CORS, використання анті-CSRF токенів.

На рисунку 2.6 показана схема роботи концепції аутентифікації та авторизації. На цій схемі видно, що з початку програма користувача відправляє ключ програми і секретні дані на сторінку входу в систему на сервері аутентифікації. Якщо пройдено аутентифікацію, сервер аутентифікації повертає користувачеві токен доступу (авторизації). Токен доступу (авторизації) упакований у параметр запиту перенаправлення відповіді (302) на запит. Перенаправлення надсилає запит користувача назад на сервер ресурсів (сервер API). Потім користувач надсилає запит на сервер ресурсів (сервер API). Токен доступу (авторизації) додається в заголовок запиту API зі словом Bearer, за яким слідує рядок токена. Сервер API перевіряє токен доступу (авторизації) у запиті користувача і вирішує, чи автентифікувати користувача. Токени доступу (авторизації) не тільки забезпечують аутентифікацію для сторони, що запитує, але й визначають права користувача на використання API. Крім того, токени доступу (авторизації) зазвичай закінчуються через деякий час і вимагають від користувача повторного входу до системи.

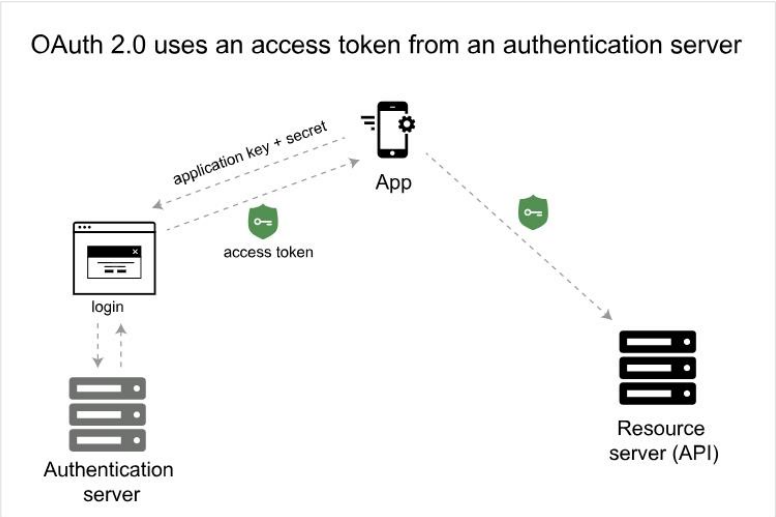


Рисунок 2.6 − Схема роботи концепції аутентифікації та авторизації

Крос-доменні запити: через політику одного джерела (same-origin policy), фронтенд-застосунки, які розміщені на одному домені, не можуть безпосередньо виконувати запити до API, що розміщені на іншому домені. Для цього використовуються техніки, як CORS (Cross-Origin Resource Sharing).

Проблема:

Браузери блокують крос-доменні запити через політику одного джерела.

Рішення:

Налаштування заголовків CORS на сервері, використання JSONP (якщо все ще потрібно), проксі-сервери або налаштування сервера для відправлення відповідних заголовків.

На рисунку 2.7 показана схема роботи концепції крос-доменних запитів. Схема показує, що кросдоменний запит складається з наступних етапів:

– клієнт надсилає запит на сервер;

– клієнт надсилає запит на сервер, який містить інформацію про домен, з якого він надсилається;

– сервер перевіряє дозвіл на доступ;

– сервер перевіряє дозвіл на доступ до домену, з якого надсилається запит;

– сервер виконує запит;

– сервер виконує запит і повертає відповідь клієнту.

Також ця схема ілюструє взаємодію між JavaScript, який запускає запит fetch(), браузером і сервером.

Браузер спочатку відправляє preflight запит методом OPTIONS до сервера, щоб перевірити, чи дозволяє сервер такі запити від джерела запиту. Він включає заголовки, які вказують джерело (Origin) та тип запиту (Access-Control-Request-Method) і запитувані заголовки (Access-Control-Request-Headers).

Якщо сервер дозволяє запити з цього джерела, він відповідає зі статусом 200 OK та включає заголовки Access-Control-Allow-Origin, Access-Control-Allow-Methods, Access-Control-Allow-Headers, та Access-Control-Max-Age, які підтверджують, що запит дозволений.

Після отримання дозволу від сервера, браузер (клієнт) відправляє основний HTTP-запит.

Сервер обробляє запит та відправляє основну HTTP-відповідь з заголовком Access-Control-Allow-Origin, який підтверджує, що джерело, з якого було відправлено запит, має дозвіл на отримання відповіді.

На схемі також вказані два потенційні результати: «якщо дозволено: успіх, інакше — помилка», що означає, що якщо сервер не надає дозвіл, браузер блокує запит, і JavaScript отримує помилку замість відповіді.

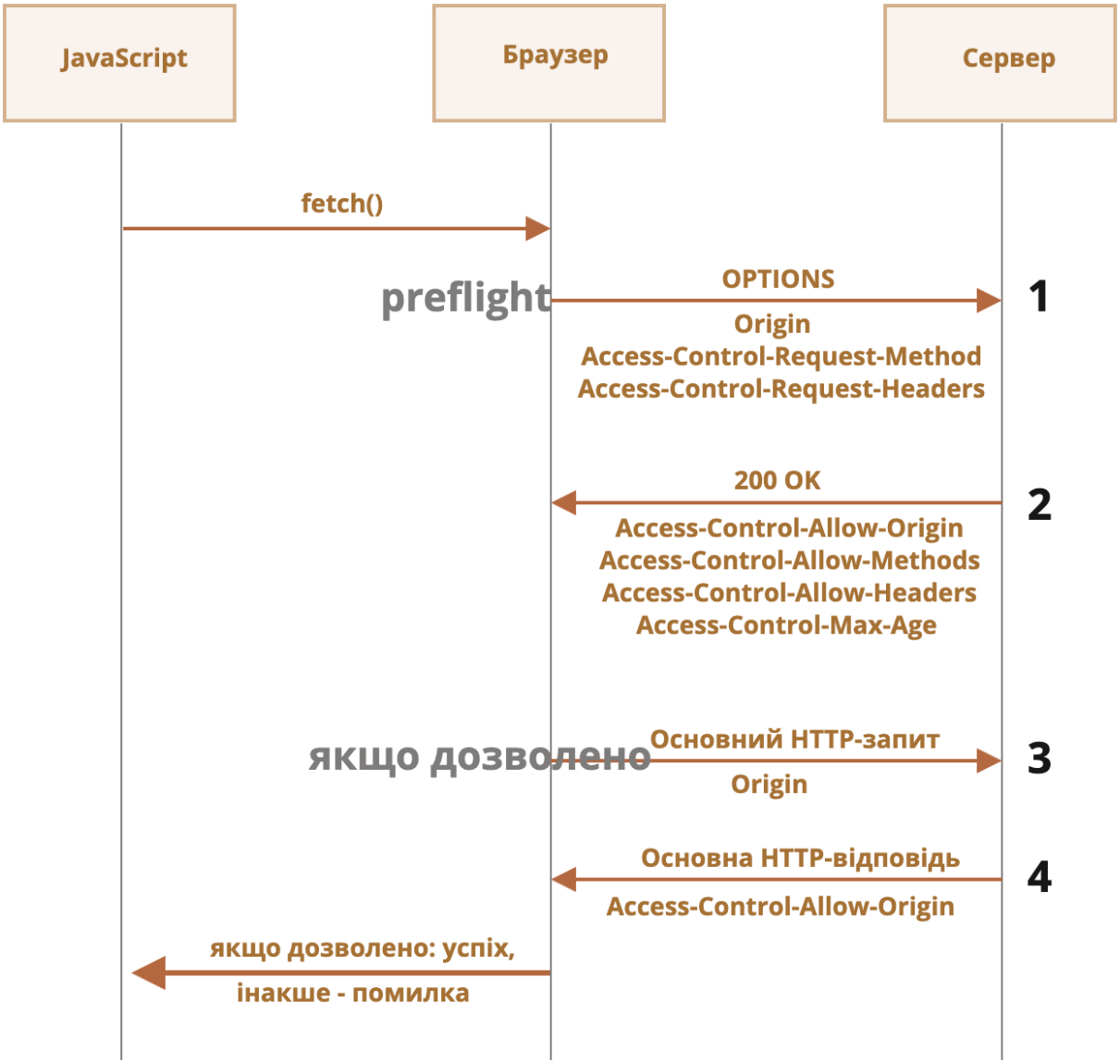


Рисунок 2.7 – Схема роботи концепції крос-доменних запитів

Застосування фреймворків та бібліотек: для спрощення роботи з API, розробники часто використовують різноманітні фреймворки та бібліотеки, такі як Axios, Fetch API, jQuery (для AJAX запитів) та багато інших.

Проблема:

Вибір неправильного інструменту, який не відповідає потребам проекту.

Рішення:

Аналіз вимог проекту перед вибором фреймворків або бібліотек, врахування розміру спільноти, документації та підтримки.

На рисунку 2.8 показані основні фронтенд-фреймворки та бібліотеки, які допомагають працювати з різними API.

Рисунок 2.8 − Схема роботи концепції застосування фреймворків та бібліотек

WebSockets: для реалізації двосторонньої комунікації в реальному часі між фронтендом і сервером можуть використовуватися WebSockets, що дозволяє серверу відправляти дані без необхідності попереднього запиту від клієнта.

Проблема:

Складності зі стабільністю з'єднання, масштабуванням і безпекою.

Рішення:

Використання надійних бібліотек, таких як Socket.IO, реалізація стратегій повторного підключення, використання шифрування та аутентифікації.

На рисунку 2.9 показана схема роботи концепції WebSockets. Схема показує, що WebSockets складається з двох компонентів: клієнт та сервер.

Клієнт — це програмний компонент, який створює WebSocket-з'єднання з сервером. Клієнтом може бути веббраузер, вебзастосунок або будь-яка інша програма, яка підтримує WebSockets.

Сервер — це програмний компонент, який приймає WebSocket-з'єднання від клієнта. Сервер може бути вебсервером, API-сервером або будь-якою іншою програмою, яка підтримує WebSockets.

WebSocket-з'єднання встановлюється за допомогою HTTP-запиту. Клієнт надсилає HTTP-запит з типом запиту GET і параметром Upgrade: websocket. Сервер відповідає на цей запит, надсилаючи HTTP-заголовок Upgrade: websocket.

Після встановлення з'єднання клієнт і сервер можуть обмінюватися даними в режимі реального часу. Дані передаються у вигляді текстових повідомлень.

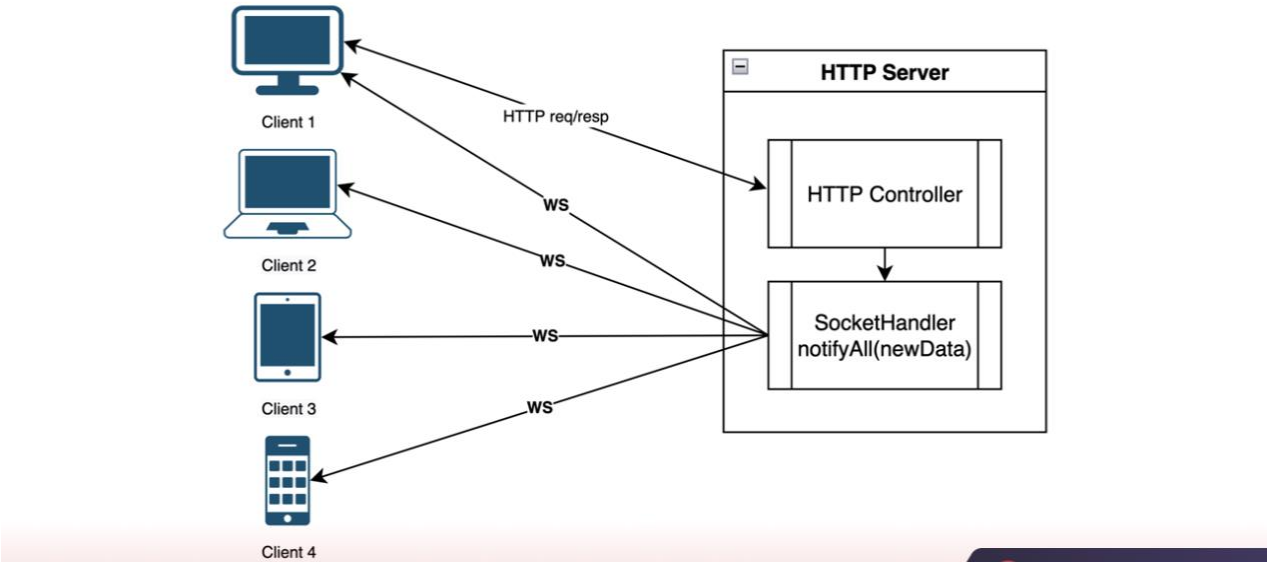


Рисунок 2.9 − Схема роботи концепції WebSockets

GraphQL: це мова запитів для API, яка дозволяє клієнтам запитувати точно ті дані, які їм потрібні, що може зменшити кількість запитів і кількість переданих даних.

Проблема:

Складність запитів, потенційна перевантаженість сервера через складні або занадто великі запити.

Рішення:

Використання інструментів для обмеження складності запитів (наприклад, depth limiting), пагінація, кешування запитів.

На рисунку 2.10 показана схема роботи концепції GraphQL. Схема показує, що GraphQL складається з трьох основних компонентів: клієнт, сервер та шлюз.

Клієнт − це програмний компонент, який генерує запити GraphQL і отримує відповіді від сервера. Клієнтом може бути веббраузер, вебзастосунок або будь-яка інша програма, яка підтримує GraphQL.

Сервер − це програмний компонент, який обробляє запити GraphQL і повертає відповіді. Сервер може бути вебсервером, API-сервером або будь-якою іншою програмою, яка підтримує GraphQL.

Шлюз − це програмний компонент, який виконує перетворення між протоколом HTTP і протоколом GraphQL. Шлюз може бути необхідним, якщо сервер не підтримує HTTP.

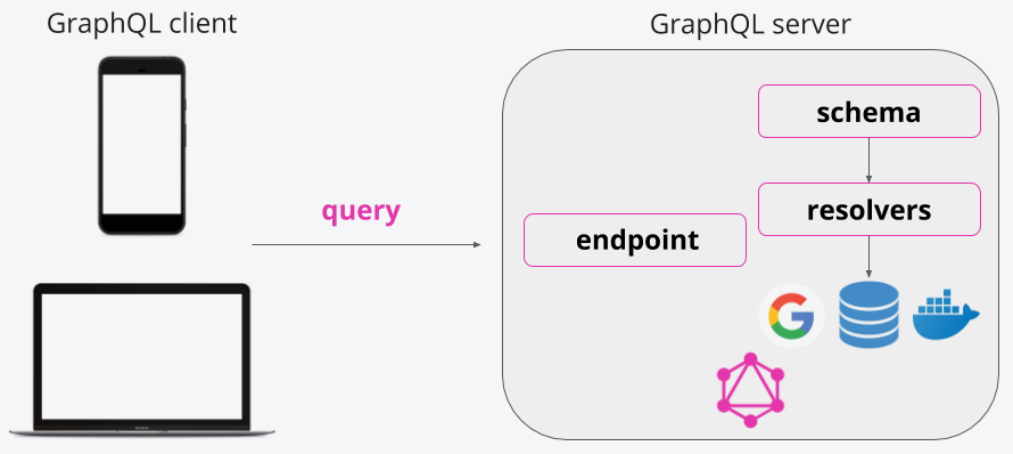


Рисунок 2.10 − Схема роботи концепції GraphQL

Фронтенд-розробка з використанням API є важливою частиною сучасної веброзробки, оскільки це дозволяє створювати багатофункціональні, адаптивні та ефективні вебзастосунки.

Загалом, при роботі з API важливо ретельно планувати архітектуру фронтенду, передбачати можливі проблеми та розробляти стратегії їх вирішення. Це допоможе створити надійні, безпечні та ефективні вебзастосунки.

2.2 Моделі та методи роботи з API вебресурсів

Redux-Toolkit та Redux-Saga – це інструменти, які використовуються для управління станом та побічними ефектами в JavaScript-застосунках, зазвичай з React або React Native.

Redux Toolkit представляє собою комплект інструментів, який полегшує процес розробки з використанням Redux, слугуючи офіційним і комплексним рішенням для управління станом в застосунках. Він включає функцію configureStore, яка розширює можливості createStore, надаючи спрощені варіанти налаштування і оптимальні значення за замовчуванням. Ця функція дозволяє автоматично комбінувати редюсери сегментів, інтегрувати вказане Redux middleware і підтримувати розширення Redux DevTools. На рисунку 2.11 зображено приклад базовий приклад того, як можна використовувати метод configureStore();

```
import { configureStore } from '@reduxjs/toolkit';
import counterReducer from './counterSlice';

export const store = configureStore({
  reducer: {
    counter: counterReducer,
  },
  middleware: (getDefaultMiddleware) => getDefaultMiddleware().concat(/* тут мідлвари */),
  // Налаштування DevTools
  devTools: process.env.NODE_ENV !== 'production',
});
```

Рисунок 2.11 – базовий приклад того, як можна використовувати configureStore

Крім того, createSlice – це інструмент, який спрощує створення сегментів стану, приймаючи початковий стан, набір функцій редюсера і назву сегмента,

автоматично створюючи action creators і типи дій, які відповідають заданим редюсерам і стану. На рисунку 2.12 зображено приклад базовий приклад використання createSlice для управління станом в простому лічильнику.

```
import { createSlice } from '@reduxjs/toolkit';

// Початковий стан
const initialState = {
  value: 0,
};

const counterSlice = createSlice({
  name: 'counter',
  initialState,
  reducers: {
    // Редюсер для збільшення значення лічильника
    increment: state => {
      state.value += 1;
    },

  },
});

// Експорт генерованих дій
export const { increment } = counterSlice.actions;

// Експорт редюсера
export default counterSlice.reducer;
```

Рисунок 2.12 – Базовий приклад управління станом за допомогою createSlice()

У сфері асинхронної логіки та middleware, Redux Toolkit надає createAsyncThunk. Ця функція приймає строку, що ідентифікує тип дії, і callback, який створює навантаження, повертаючи обіцянку. В результаті вона створює thunk, який керує діями залежно від стадії виконання обіцянки, що дозволяє ефективно реагувати на різні етапи асинхронних операцій, наприклад, при завантаженні даних користувача з API. На рисунку 2.13 зображено приклад базовий приклад використання createAsyncThunk() для створення асинхронної дії, яка завантажує деталі користувача з API.

```
import {createAsyncThunk} from '@reduxjs/toolkit';

export const fetchUserById = createAsyncThunk(
  'users/fetchByIdStatus', // ідентифікатор дії
  async (userId, thunkAPI) => {
    const response = await fetch(
      `https://jsonplaceholder.typicode.com/users/${userId}`,
    );
    const data = await response.json();

    if (response.ok) {
      return data;
    } else {
      return thunkAPI.rejectWithValue(data);
    }
  },
);
```

Рисунок 2.13 – приклад використання методу createAsyncThunk()

Redux-Saga – це бібліотека, що полегшує управління побічними ефектами в застосунках, такими як асинхронні операції отримання даних або доступ до кешу браузера. Вона допомагає робити ці процеси більш керованими, ефективними, легкими для тестування та краще адаптованими до помилок.

У контексті Redux-Saga, ефекти є наказами для saga middleware, такі як take, put, call, fork, які в собі не мають функціональності, а повертають об'єкти JavaScript, що описують потрібні дії.

Воркери(робітники) – це функції-генератори, що виконують реальну логіку саги, використовуючи ефекти для декларативної взаємодії з зовнішнім середовищем. На рисунку 2.14 зображено приклад Worker Saga, яка використовує декілька різних ефектів, а саме call та put.

```
import {call, put, takeEvery, fork} from 'redux-saga/effects';

// Воркер сага, яка виконує асинхронний запит
function* fetchUser(action) {
  try {
    const user = yield call(
      fetchApi,
      `https://jsonplaceholder.typicode.com/users/${action.payload.userId}`,
    );
    // Диспетчеризація дії з результатами запиту
    yield put({type: 'FETCH_USER_SUCCESS', payload: user});
  } catch (e) {
    // Диспетчеризація дії помилки, якщо запит не вдалий
    yield put({type: 'FETCH_USER_FAILURE', payload: e.message});
  }
}
```

Рисунок 2.14 – приклад Worker Saga та використання її ефектів

Вотчери(спостерігачі) – це саги-генератори, що стежать за діями, які диспетчеризуються в Redux, та активують робітників саг у відповідь на певні дії, часто використовуючи takeEvery або takeLatest для визначення, як і коли відповідати на дії. Така структура дозволяє ефективно управляти складними процесами в застосунку. На рисунку 2.15 зображено приклад саги-спостерігача, яка використовує декілька різних ефектів, а саме call та put:

```
import {call, put, takeEvery, fork} from 'redux-saga/effects';

// Спостерігач сага, яка слухає дії FETCH_USER_REQUESTED
function* watchFetchUser() {
  yield takeEvery('FETCH_USER_REQUESTED', fetchUser);
}
```

Рисунок 2.15 – приклад саги-спостерігача

Потоки: дозволяють управляти одночасними завданнями. Це важливо для ситуацій, коли потрібно координувати виконання кількох саг одночасно. Два найбільш використовувані потоки – all та race. На рисунку 2.16 зображено

приклад використання потоку all – це потік, який виконує кілька ефектів паралельно і чекає завершення всіх них, а на рисунку 2.17 зображено приклад використання потоку race – це потік, який запускає кілька ефектів одночасно і виконує той із них, який завершиться першим.

```
import { all, call } from 'redux-saga/effects';

function* fetchAll() {
  const [users, posts] = yield all([
    call(fetchApi, '/users'),
    call(fetchApi, '/posts')
]);
}
```

Рисунок 2.16 – використання all()

```
import { race, take, put, call } from 'redux-saga/effects';

function* timeoutSaga() {
  const { posts, timeout } = yield race({
    posts: call(fetchApi, '/posts'),
    timeout: call(delay, 1000) // delay - це функція, що створює затримку
  });

  if (posts) {
    yield put({ type: 'POSTS_FETCHED', payload: posts });
  } else {
    yield put({ type: 'FETCH_TIMEOUT' });
  }
}
```

Рисунок 2.17 – використання race()

Розглянемо спільне використання Redux-Toolkit та Redux-Saga. У типовому налаштуванні, що використовує Redux-Toolkit та Redux-Saga:

– Redux-Toolkit буде використовуватися для налаштування сховища, редюсерів та синхронних action creators;

– Redux-Saga буде використовуватися для обробки складної асинхронної логіки та побічних ефектів. Вона спостерігає за діями, диспетчеризованими action creators Redux-Toolkit, і виконує необхідні асинхронні операції.

Ось приклад високого рівня того, як вони можуть бути використані разом:

- визначити createSlice з Redux-Toolkit для синхронних оновлень стану;
- використовувати createAsyncThunk для визначення асинхронних операцій, які генерують дії;
- налаштувати сагу з takeEvery, що спостерігає за цими асинхронними діями;
- у сазі використовувати метод call() для виконання асинхронної роботи та метод put() для диспетчеризації дій на основі результатів асинхронної роботи.

Ці методи та моделі поєднують структуроване управління станом Redux з потужним управлінням побічними ефектами, створюючи надійну та масштабовану рамку для складних застосунків. Завдяки такому підходу, розробники отримують потужний інструментарій для створення більш чистого, підтримуваного та ефективного коду в своїх проектах.

2.3 Висновки

У даному розділі детально проаналізовано концепції, моделі та методики, які застосовуються при використанні потужної комбінації Redux-Toolkit та Redux-Saga для управління станом та асинхронними операціями в контексті взаємодії з API вебресурсів. Було встановлено, що використання Redux-Toolkit сприяє зменшенню бойлерплейту та складності коду, забезпечує більш зрозумілі та стандартизовані підходи до структуризації стану застосунків. Водночас, інтеграція Redux-Saga дозволяє ретельно управляти асинхронними процесами, включаючи дії, пов'язані з обробкою даних API, з урахуванням різноманітних сценаріїв відмовостійкості.

За допомогою Saga Effects було створено гнучку систему, яка здатна реагувати на складні послідовності дій користувачів, ефективно обробляти

помилки та відокремлювати нестабільні частини системи з метою підтримки безперервної роботи застосунку. Також була розроблена модель програмного забезпечення, яка впроваджує патерн для розподілу стану та запитів між різними сегментами, що дозволило ізолювати потенційно небезпечні або високонавантажені частини застосунку та оптимізувати використання ресурсів.

Додатково, було визначено, що стратегічне використання ефектів Redux-Saga, таких як takeLatest та takeEvery, може значно покращити відповідь на часті запити користувачів, зменшуючи навантаження на сервери та запобігаючи втраті даних через перевантаження. Це дозволило оптимізувати потоки даних, забезпечуючи стабільність та надійність системи при високій конкуренції запитів та інтенсивному обміні даними.

З урахуванням вищесказаного, наступним етапом роботи стане розробка детальної специфікації вимог до системи, яка інтегрує Redux-Toolkit та Redux-Saga. Це включатиме проєктування архітектури, декомпозицію компонентів, визначення взаємодії між частинами стану та побічними ефектами, а також аналіз шляхів масштабування та забезпечення високої доступності сервісу. Окрім того, буде здійснено вибір оптимальних засобів реалізації, які найкраще відповідатимуть заданим вимогам, з метою створення ефективного, гнучкого та відмовостійкого вебзастосунку.

3 ТЕХНОЛОГІЯ РЕАЛІЗАЦІЇ УДОСКОНАЛЕНОГО МЕТОДУ РОБОТИ З API ПРОГРАМНИХ СИСТЕМ

3.1 Аналіз вимог до програмного засобу

Поєднуючи Redux-Toolkit із Redux-Saga, ми можемо використовувати простоту Redux-Toolkit для більшості синхронної логіки, в той час як Redux-Saga може бути використаний для управління більш складними асинхронними операціями. Тому було сформульовано такі вимоги до програмного забезпечення, яке використовує обидві технології:

– управління станом і розділення логіки: забезпечити, щоб сховище Redux було єдиним джерелом істини для стану, тоді як Sagas обробляють логіку асинхронних операцій і побічних ефектів;

– обробка дій: визначити чіткі шаблони для надсилання дій. Sagas слухають надіслані дії та виконують побічні ефекти, потім надсилають додаткові дії на основі результатів запитів до API;

– обробка помилок: розробити міцні стратегії обробки помилок в Sagas для невдалих запитів до API, потенційно використовуючи createAsyncThunk від Redux-Toolkit або інші механізми для оновлення стану при помилкових умовах;

– ефективність та продуктивність: оптимізувати Sagas, щоб уникнути непотрібних запитів до API і забезпечити, щоб компоненти перерендерювалися тільки коли це необхідно;

– обслуговуваність: спільне використання Redux-Toolkit та Redux-Saga повинно привести до кодової бази, яка легко обслуговується і масштабується, з чітким розподілом завдань і добре визначеними потоками логіки;

– тестування: як Redux-Toolkit slices, так і Redux-Sagas повинні бути легкими для тестування з передбачуваними результатами, використовуючи модульні тести для редукторів і інтеграційні тести для Sagas;

– типізація та валідація: у випадку використання TypeScript, належно типізувати всі дії, стан та Sagas, щоб уникнути помилок під час виконання та забезпечити якість коду;

– ініціалізація стану модулів: кожен модуль у Redux-Toolkit повинен мати initial state з дефолтними значеннями. шо гарантує, що стан застосунку може бути передбачуваним від самого початку, і забезпечує базовий рівень консистенції перед тим, як будь-які дії вплинуть на стан;

– обробка ключових подій запиту у слайсах: для кожного асинхронного запиту до API, слайси мають обробляти чотири ключові події: trigger (спрацювання запиту), success (успішне завершення запиту), error (виникнення помилки під час запиту), та fulfilled (запит завершено, незалежно від результату). Це забезпечує структуровану обробку запитів до API і дозволяє реагувати на різні стани запиту;

– селектори: відповідні селектори повинні бути створені для кожного слайса стану, вони дозволяють компонентам вибирати необхідні частини стану без необхідності знання про внутрішню структуру стану слайса, що не тільки спрощує доступ до даних, але й покращує перевикористання та тестування коду;

– стандартизація дій та редюсерів: потрібно використовувати консистентні назви для дій, що полегшує розуміння та відлагодження коду, а також ред'юсери повинні бути чистими функціями і не містити побічних ефектів;

– модульність та повторне використання: прагніть до того, щоб кожен слайс був максимально незалежним та легко повторно використовуваним у різних частинах застосунку;

Підсумовуючи, вимоги до програмного забезпечення, що поєднує Redux-Toolkit та Redux-Saga, повинні зосереджуватися на чіткому розділенні завдань, ефективному управлінні станом, міцній обробці помилок та обслуговуванності. Належне поєднання цих двох технологій може привести до масштабованої, обслуговуваної та ефективної архітектури фронтенду для управління складним станом застосунків та побічними ефектами. Такий інтегрований підхід забезпечує високу гнучкість у реагуванні на зміни вимог та легкість додавання нових функцій без зайвого компрометування вже існуючого коду. Він також сприяє кращому розумінню потоків даних у застосунках, що є критично важливим для команд розробників, які працюють над складними проектами. Крім того,

використання такої архітектури спрощує тестування та відлагодження коду, оскільки воно дозволяє ізолювати компоненти та процеси, роблячи можливим більш детальний аналіз та контроль якості.

3.2 Проектування програмного засобу

3.2.1 Структура типового модуля

Розглянемо приклад структури одного модуля. Модуль складається з шести файлів: saga.ts, selectors.ts, services.ts, slice.ts, state.ts та types.ts.

Saga.ts: зазвичай містить Redux-Saga watchers і workers, які використовуються для асинхронних дій, таких як здійснення API-запитів. Він слухає визначені дії, відправлені до store, і запускає саги для обробки цих дій;

Selectors.ts: використовується для вилучення певних частин стану з Redux store. Цей модуль містить функції, які приймають повний стан застосунка і повертають лише ту частину, яка потрібна для компонентів;

Services.ts: модуль, у якому зазвичай знаходяться функції, що використовуються для взаємодії з зовнішніми API, такі як отримання або відправлення даних. Це може включати різні HTTP-запити (get, post, put, delete) та інші функції;

Slice.ts: в Redux Toolkit є функцією, яка створює reducer і пов'язані з ним дії. Цей модуль містить логіку для обробки змін в стані, пов'язаних з певною функцією або компонентом (у цьому випадку, профілем користувача);

State.ts: тут описано початковий стан (initial state) для певної частини застосунка. Він визначає, як виглядатиме стан відразу після ініціалізації застосунка, перш ніж будь-які дії змінять його;

Types.ts: містить визначення TypeScript інтерфейсів та типів для суворої типізації стану, дій та інших частин Redux-екосистеми. Таке визначення типів допомагає уникнути помилок під час розробки;

Отже, у якості початкового об’єкта визначено та протипізовано такі стани: data, loading, error та done.

Data: цей стан використовується для зберігання відповіді від API. Коли дані успішно завантажені, вони зберігаються у цьому стані, звідки компоненти можуть їх отримати та відобразити;

Loading: цей булевий стан використовується для індикації процесу виконання запиту. Коли запит ініціюється, loading стає true, що може активувати індикатор завантаження в UI;

Error: це поле призначене для зберігання помилки, якщо вона виникає під час запиту. Якщо запит завершується з помилкою, відповідне повідомлення зберігається в цьому стані;

Done: цей стан може використовуватися для індикації того, що запит було повністю оброблено, незалежно від того, чи було це успішно, чи з помилкою.

За обробку дій, пов'язаних з цими станами відповідають редюсери у файлі slice.ts: fetchDataTrigger, fetchDataSuccess, fetchDataFailed та fetchDataFulfilled.

FetchDataTrigger: цей редюсер викликається для ініціації запиту. Він встановлює loading в true та error в null, підготовлюючи стан до нового запиту;

FetchDataSuccess: коли дані успішно отримані, цей редюсер зберігає ці дані в стані data та встановлює done в true;

FetchDataFailed: у разі помилки запиту, цей редюсер зберігає інформацію про помилку в стані error;

FetchDataFulfilled: цей редюсер використовується для встановлення стану loading в false після того, як запит був повністю оброблений.

Селектори в selectors.ts дають змогу вилучити специфічну інформацію зі стану. Було створено такі селектори: selectData, selectFetchLoading, selectFetchDone та selectFetchError.

SelectData: повертає частину стану data, яка містить інформацію профілю;

SelectFetchLoading: повертає булеве значення стану loading, що індикує, чи запит наразі виконується;

SelectFetchDone: повертає булеве значення стану done, що індикує, чи запит було повністю оброблено;

SelectFetchError: повертає значення стану error, яке містить помилку запиту.

Дії в saga.ts визначають асинхронну логіку обробки запитів. Було створено такі дії: запуск запиту, виконання API-запиту, успішний результат, обробка помилки, завершення запиту.

Запуск запиту: запит ініціюється за допомогою fetchDataTrigger, який встановлює стан loading в true, щоб індикувати початок завантаження;

Виконання API-запиту: сага використовує ефект call для виклику функції getData з модуля сервісів, яка робить HTTP-запит до API для отримання даних профілю;

Успішний результат: якщо запит завершується успішно, сага використовує ефект put для відправлення дії fetchDataSuccess, передаючи отримані дані як payload. Ця дія встановить стан data з отриманими даними та змінить done на true;

Обробка помилки: у разі виникнення помилки під час запиту, сага ловить помилку і використовує ефект put для відправлення дії fetchDataFailed, передаючи інформацію про помилку як payload. Це встановить стан error з відповідною інформацією пропомилку;

Завершення запиту: незалежно від результату, сага завжди використовує ефект put для відправлення дії fetchDataFulfilled в кінці процесу. Ця дія встановить стан loading назад у false, що сигналізує додатку про завершення процесу завантаження.

Отже, кожен модуль відіграє певну роль у структурі Redux-базованого застосунка, дозволяючи розділити логіку і зробити код більш читабельним та легким для підтримки. Така послідовність дій дозволяє ефективно керувати станами завантаження, успіху, помилки та завершення завантаження, забезпечуючи надійний механізм для асинхронних запитів в Redux-базованому застосунку. Загальна схема роботи модуля зображена на рисунку 3.1 (лише послідовність подій) та 3.2 (послідовність подій з демонстрацією того, як виглядає стан модуля у кожний момент).

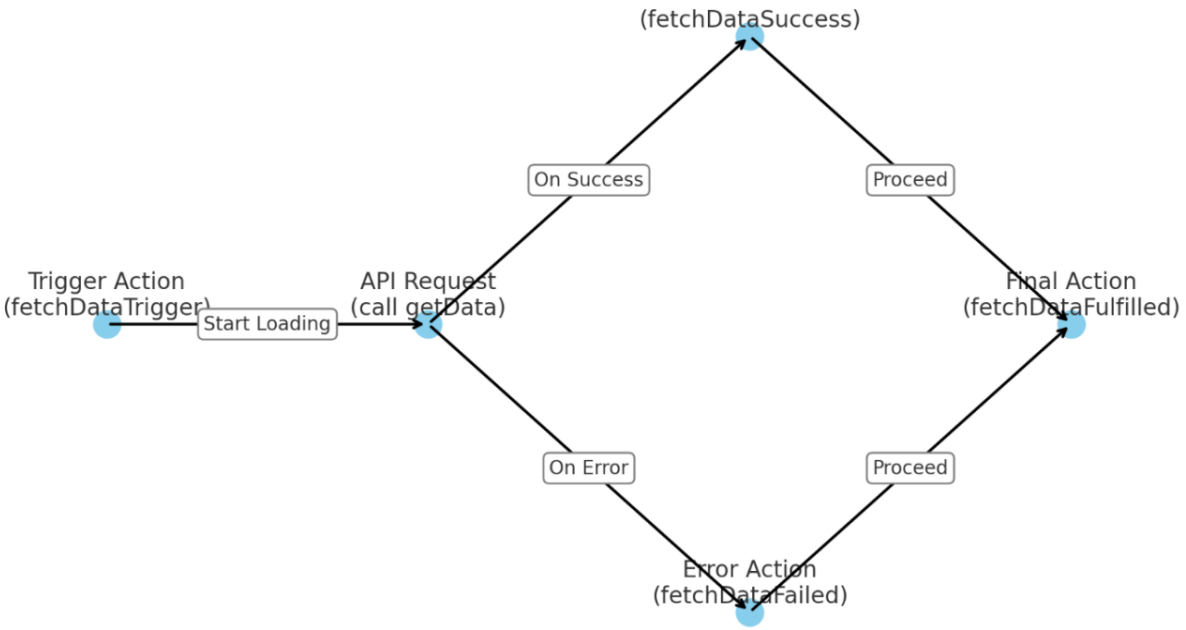


Рисунок 3.1 − Схема послідовності подій у модулі

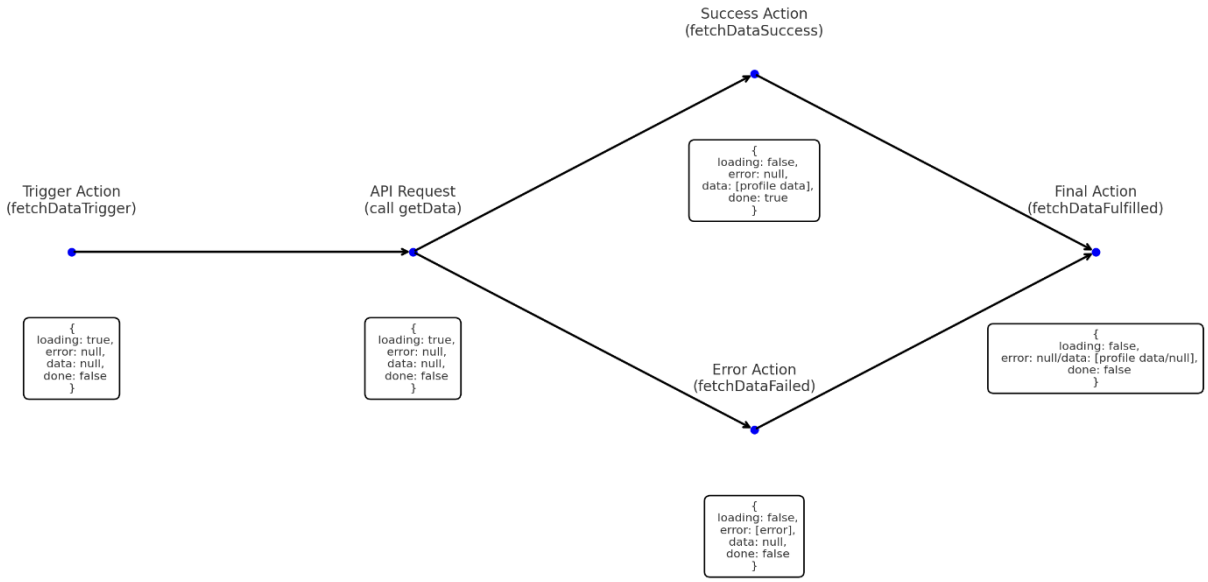


Рисунок 3.2 − Детальна схема станів модуля при кожній події

3.2.2 Структура взаємодії модулів

Розроблювана програмна система повинна працювати у якості окремого самостійного проекту. У зв’язку з цим розроблюваний програмний засіб буде

мати клієнт-сервісну архітектуру та окремі модулі (нижче детально описано кожен модуль):

Клас «Role»

Атрибути:

– Id (унікальний ідентифікатор): унікальний ідентифікатор ролі;
– Name (рядок): назва ролі (наприклад, «адміністратор», «користувач»).

Клас «User»

Атрибути:

– Id (унікальний ідентифікатор): унікальний ідентифікатор користувача;
– Surname (рядок): прізвище користувача;
– FirstName (рядок): ім'я користувача;
– Email (рядок): електронна пошта користувача;
– Password (рядок): пароль користувача;
– FavoriteArticles (масив): масив улюблених статей користувача;

Зв'язки:

– RoleId (зв'язок з класом «Role»): вказує на роль користувача;

Клас «Article»

Атрибути:

– Id (унікальний ідентифікатор): унікальний ідентифікатор статті;
– Title (рядок): назва статті;
– Text (рядок): текст статті;
– Images (масив): масив зображень, пов'язаних зі статтею;

Зв'язки:

– Може бути доданий адміністратором;

Клас «Images»

Атрибути:

– Id (унікальний ідентифікатор): унікальний ідентифікатор зображення;
– Uri (рядок): посилання на фотографію.

Ці класи відображаються на діаграмі з відповідними зв'язками і атрибутами. Клас «User» має зв'язок з класом «Role» через RoleId. Клас «Article» може мати зв'язки з класом «Images» для представлення фотографій, пов'язаних зі статтею.

Далі виконаємо декомпозицію модулів на компоненти, кожний з яких відповідатиме за конкретну логічну функцію. На рисунку 3.1 представлена логічна структура програми у вигляді діаграми класів.

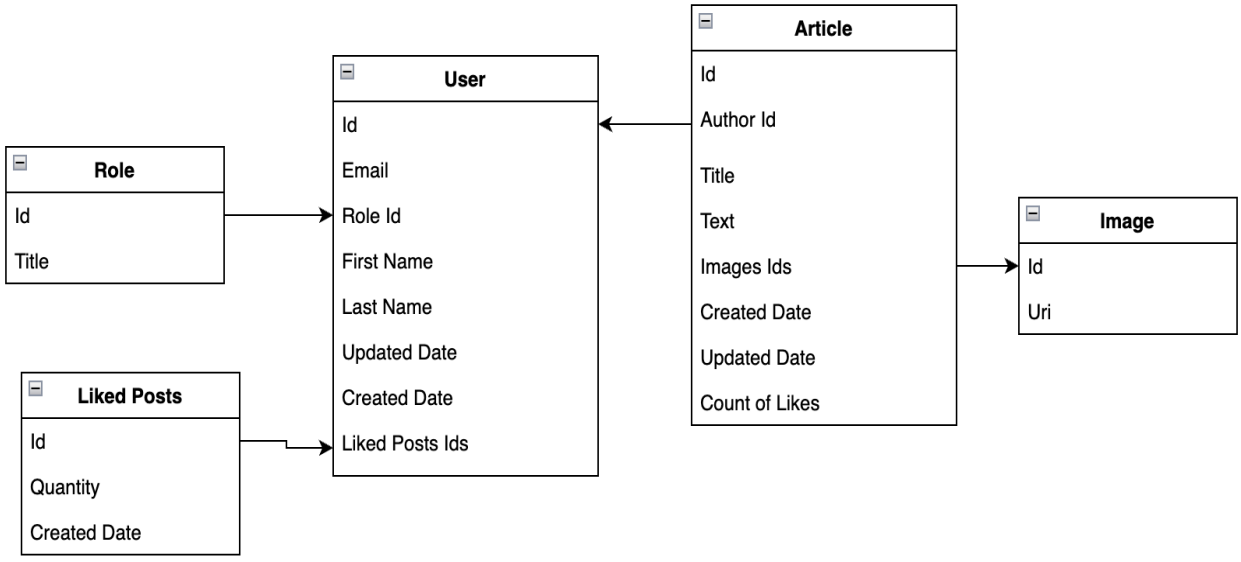


Рисунок 3.3 – Діаграма класів

3.3 Аналіз та вибір засобів програмної реалізації методу

При створенні програмного забезпечення важливо включити додаткові критерії:

– інструментарій повинен бути адаптивним, тобто здатним до швидкого розширення;

– повинна бути можливість безпроблемної інтеграції інструменту в різні середовища;

– необхідно забезпечити надійний захист і конфіденційність даних користувачів відповідно до існуючих законодавчих норм щодо обробки та зберігання особистих відомостей;

– інструмент повинен ефективно обробляти запити в негайному режимі, уникаючи будь-яких затримок.

З огляду на це, було вирішено використовати такі технології для створення програмного забезпечення:

Мова програмування:

JavaScript або TypeScript (зазвичай використовується TypeScript для забезпечення типізації в коді, що може підвищити якість коду і зменшити кількість помилок у довгостроковій перспективі).

Фронтенд фреймворк:

React (як найбільш поширений вибір для роботи з Redux, має велику спільноту та розгалужену екосистему).

Стан управління:

Redux Toolkit (покращує роботу з Redux, зменшуючи кількість шаблонного коду, та вносить оптимізації та додаткові функції, такі як налаштовані слайси та енкапсуляція логіки).

Основні переваги Redux Toolkit включають:

– спрощення конфігурації Redux: Redux Toolkit надає зручний шлях для створення Redux store, уникнення повторення коду і автоматизації процесів;

– скорочення шаблонного коду: використання Redux Toolkit дозволяє уникнути великої кількості рутинних дій, таких як створення action creators, reducers тощо, завдяки використанню спеціальних функцій, таких як createSlice;

– налаштовані слайси (Customizable Slices): Redux Toolkit використовує поняття «слайсів» (slices), що дозволяє групувати action creators та reducers, пов'язані з певним станом застосунку, це допомагає управляти логікою для конкретної частини стану;

– енкапсуляція логіки: використання Redux Toolkit сприяє кращій організації та розподілу логіки Redux у проєкті, а також полегшує його розуміння та підтримку;

– вбудована підтримка DevTools: Redux Toolkit поставляється з вбудованою підтримкою для Redux DevTools Extension, що полегшує відлагодження та візуалізацію стану застосунку.

Засіб для управління асинхронними операціями:

Redux-Saga (для управління складними сайд-ефектами, такими як асинхронні запити до API, обробка вебсокетів тощо, забезпечуючи детальний контроль над асинхронними операціями).

Основні принципи та можливості Redux-Saga включають:

– generator-функції: Redux-Saga використовує JavaScript generator-функції для створення саг (sagas), які представляють собою спеціальні функції для обробки асинхронних операцій. Це дозволяє легко керувати послідовністю асинхронних дій;

– події та ефекти: саги реагують на певні події (actions) у Redux та запускають асинхронні операції (ефекти) на їх основі. Наприклад, вони можуть слідкувати за певними actions і відправляти запити до API або виконувати інші асинхронні дії;

– контроль та обробка паралельних процесів: Redux-Saga надає можливість запускати паралельні процеси та контролювати їх виконання та завершення;

– управління складністю асинхронної логіки: вона дозволяє розміщувати складну асинхронну логіку в окремих сагах, що полегшує управління та збереження цієї логіки;

– тестування: Redux-Saga надає інструменти для тестування саг, що дозволяє перевіряти їх поведінку та переконатися, що вони працюють правильно.

Бекенд фреймворк:

Node.js з Express.js (для створення RESTful API, що є простим у використанні і широко підтримується).

База даних:

MongoDB (документо-орієнтована база даних, яка легко інтегрується з Node.js та ефективно працює з JSON, що є зручним для JS-орієнтованих стеків).

Інтерфейс користувача:

Ant Design або інші React UI бібліотеки (для прискорення розробки інтерфейсу з використанням готових компонентів).

CSS-рішення:

Styled Components або Emotion (для використання CSS-in-JS, які забезпечують потужні можливості для стилізації компонентів).

Управління залежностями:

NPM (для управління бібліотеками та пакетами в проекті).

Система контролю версій:

Git (за умовчанням для більшості проектів, може бути інтегрована з різними сервісами хостингу коду).

Розгортання та хостинг:

- Docker (для контейнеризації та легкого розгортання);
- AWS, або Heroku (залежно від вимог до інфраструктури та бюджету).

Вибір цих технологій базується на широкій підтримці спільноти, великій кількості ресурсів для навчання, і, що важливо, на гнучкості та масштабованості для майбутнього росту застосунку. Важливо також врахувати зручність роботи з цими інструментами та їх взаємну інтеграцію.

3.4 Висновки

У цьому розділі було розглянуто комплексні вимоги до програмного засобу, включаючи його адаптивність, інтегрованість, безпеку та ефективність. Було представлено структуру типового модуля, яка забезпечує зручність у розширенні та модифікації системи, а також деталізовано структуру взаємодії модулів, що гарантує надійну та ефективну комунікацію в межах програми.

Аналізуючи та обираючи засоби для реалізації визначеного методу, було зосереджено увагу на поєднанні Redux-Toolkit і Redux-Saga для управління

станами та асинхронними процесами у вебзастосунках. Redux-Toolkit пропонує сучасний підхід до управління станом зі зменшенням шаблонного коду, тоді як Redux-Saga забезпечує потужний механізм для управління сайд-ефектами, такими як взаємодія з API.

Архітектура системи, побудована на цих технологіях, орієнтована на модульність та високий рівень абстракції. Було визначено, що кожен модуль у системі є самодостатнім і здатен виконувати певну функцію незалежно від інших модулів, що сприяє легкості тестування та відладки. Між модулями встановлені чіткі інтерфейси взаємодії, що забезпечують надійну інтеграцію та знижують залежності між компонентами системи.

Така архітектура сприяє створенню масштабованих та легко підтримуваних вебзастосунків. Вона готова до реалізації складних бізнес-процесів, при цьому забезпечуючи високу швидкість розвитку та можливість адаптації до змінних вимог користувачів і ринку. Враховуючи сучасні тенденції в розробці програмного забезпечення, вибір цих технологій є обгрунтованим та перспективним рішенням.

Поряд з визначенням архітектурної структури, було уточнено склад програмного засобу, детально описали функції кожного модуля і розбили їх на окремі компоненти. Також було вибрано та обґрунтували набір необхідних технологій для реалізації поставлених завдань, включаючи TypeScript, React, Redux, Redux-Toolkit, Redux-Saga Node.js, Express, NPM і Docker.

Висновок з цього розділу вказує на те, що інтеграція Redux-Toolkit і Redux-Saga є оптимальним рішенням для створення надійного, масштабованого та безпечного вебзастосунку. Цей підхід не тільки сприяє ефективній розробці та легкості підтримки коду, але й забезпечує високий рівень користувацького досвіду завдяки швидкій обробці запитів та реальному часу відгуку.

Наступним етапом є реалізація спроектованого ПЗ та його тестування, а також доведення ефективності удосконаленого методу поєднання технологій Redux-Toolkit та Redux-Saga для роботи з API вебресурсів.

4 РЕАЛІЗАЦІЯ ТА ТЕСТУВАННЯ ПРОГРАМНОГО ЗАСОБУ

4.1 Програмна реалізація

4.1.1 Розробка програмних модулів

Структура проекту представлена на рисунку 4.1:

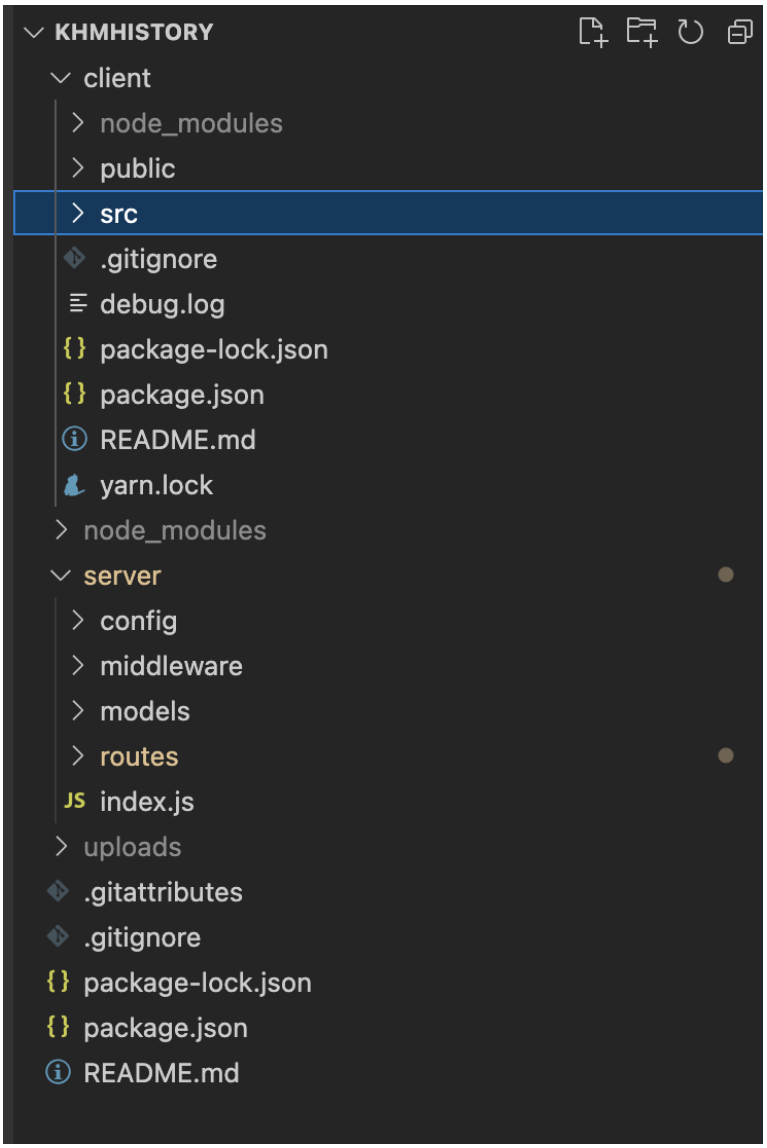


Рисунок 4.1 – Загальна структура проекту

Проект включає дві головні директорії:

– client – де розміщені файли, пов'язані з клієнтською частиною проекту;

– server – де зібрані файли, що стосуються серверної частини проекту;

Далі розглянемо вміст папки server, який демонструється на рисунку 4.2.

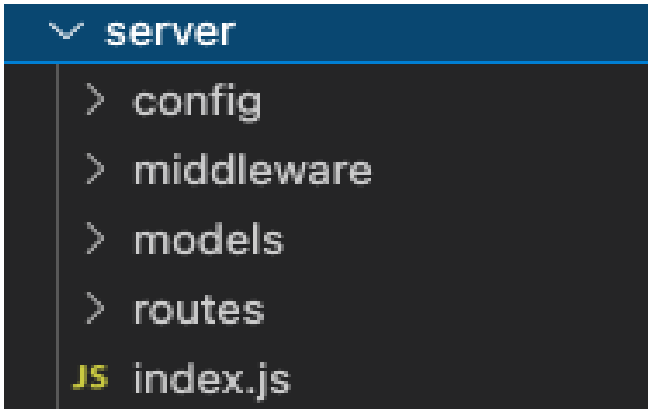


Рисунок 4.2 − Вміст папки server

У цій директорії розташований основний файл index.js, який служить точкою входу; він відповідає за ініціалізацію підключення до бази даних і запускає сервер на порті 5001. Директорія config зберігає важливі конфігураційні дані, включаючи рядки з'єднання з базою даних та .env файли (environment files). Директорія middleware включає файл auth.js, який містить вспоміжну функцію для ідентифікації користувача через JWT (JSON Web Token). Відсутність токена свідчить про те, що користувач не має авторизації. Директорія models визначає моделі, які формують структуру даних об'єктів у базі даних. У проекті присутні дві такі моделі: Article та User, які представлені на рисунку 4.3.

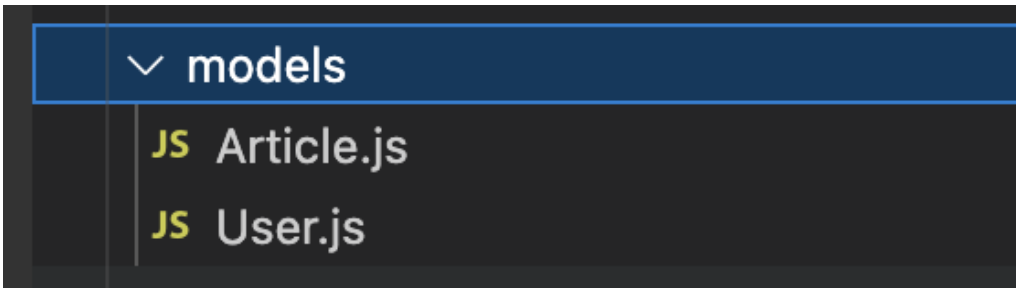


Рисунок 4.3 − Список моделей проекту

Приклад моделі:

```
const mongoose = require('mongoose');
const Schema = mongoose.Schema;

const articleSchema = mongoose.Schema({
    writer: {
```

```
        type: Schema.Types.ObjectId,
        ref: 'User'
    },
    title: {
        type: String,
        maxlength: 50
    },

    description: {
        type: String
    },

    images: {
        type: Array,
        default: []
    },

    likes: {
        type: Number,
        maxlength: 100,
        default: 0
    },

    views: {
        type: Number,
        default: 0
    }
}, { timestamps: true })


articleSchema.index({
    title:'text',
    description: 'text',
}, {
    weights: {
        name: 5,
        description: 1,
    }
})

const Article = mongoose.model('Article', articleSchema);

module.exports = { Article }
```

Далі перейдемо до директорії routes, деталі якої наведено на рисунку 4.4. Ця папка містить набір функцій, які використовуються для обробки запитів з певними URL-шляхами. Ці функції управляють запитами до бази даних, що надходять з клієнтської частини: вони можуть додавати нові записи до бази, оновлювати дані та відправляти відповіді до клієнта у відповідному форматі.

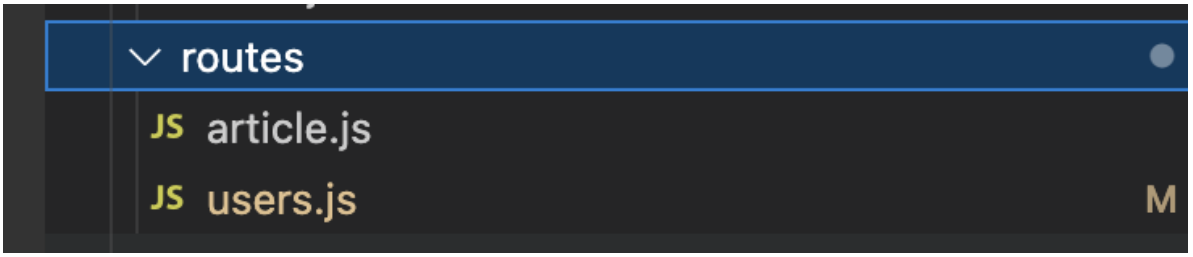


Рисунок 4.4 – Список роутів проекту

Ось приклад маршруту (роута), який виконує отримання списку всіх статей з бази даних:

```
router.post("/getArticles", (req, res) => {

    let order = req.body.order ? req.body.order : "desc";
    let sortBy = req.body.sortBy ? req.body.sortBy : "_id";
    let limit = req.body.limit ? parseInt(req.body.limit) : 100;
    let skip = parseInt(req.body.skip);

    let findArgs = {};
    let term = req.body.searchTerm;

    for (let key in req.body.filters) {

        if (req.body.filters[key].length > 0) {
            findArgs[key] = req.body.filters[key];
        }
    }

    console.log(findArgs)

    if (term) {
        Article.find(findArgs)
            .find({ $text: { $search: `\"${term}\"` } })
            .populate("writer")
            .sort([[sortBy, order]])
            .skip(skip)
            .limit(limit)
            .exec((err, articles) => {
                if (err) return res.status(400).json({ success: false, err })
                res.status(200).json({ success: true, articles, postSize: articles.length })
            })
    } else {
        Article.find(findArgs)
            .populate("writer")
            .sort([[sortBy, order]])
            .skip(skip)
            .limit(limit)
```

```
        .exec((err, articles) => {
          if (err) return res.status(400).json({ success: false, err })
          res.status(200).json({ success: true, articles, postSize: articles.length })
        })
    }

});
```

Тепер перейдемо до аналізу директорії client, зміст якої ілюструється на рисунку 4.5.

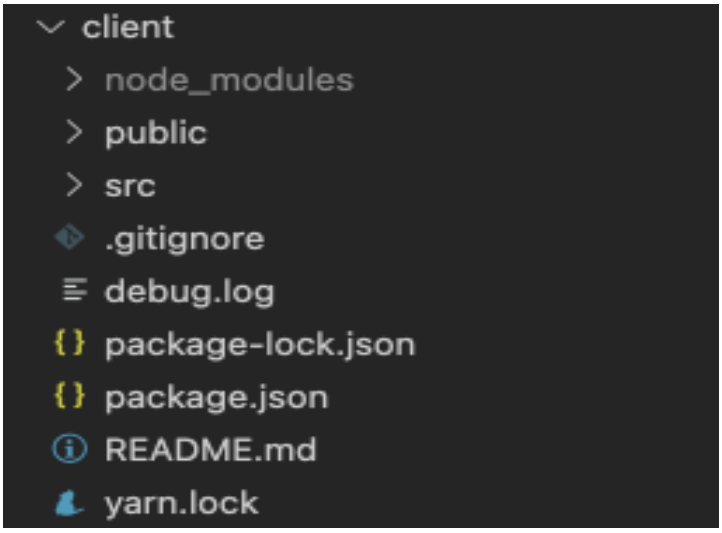


Рисунок 4.5 − Вміст папки client

У цій директорії більша частина файлів виконує утилітарні функції, тоді як основний обсяг вихідного коду знаходиться всередині папки src, вміст якої ви можете побачити на рисунку 4.6.

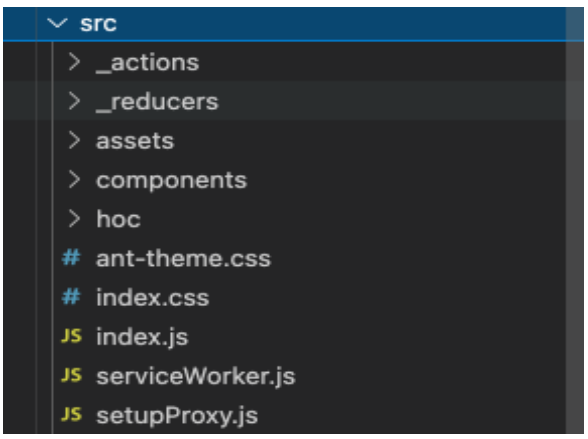


Рисунок 4.6 − Папка src

Файл index.js служить точкою входу та основним файлом для клієнтської частини проекту. Він містить кореневий компонент застосунку (root), який обгортає всі інші компоненти:

```
ReactDOM.render(
<Provider
    store={createStoreWithMiddleware(
      Reducer,
    )}
>
<Router>
<App />
</Router>
</Provider>,
  document.getElementById("app")
);
```

Файли, що мають розширення .css, зберігають стилі проекту, визначаючи візуальне оформлення користувацького інтерфейсу. Ось приклад таких стилів:

```
body {
  font-family: -apple-system, BlinkMacSystemFont, "Segoe UI", Helvetica, Arial, sans-serif, "Apple Color Emoji",
    "Segoe UI Emoji", "Segoe UI Symbol";
  font-size: 14px;
  line-height: 1.5;
  color: #91b3e5;
  background-color:#133f46 !important;

}
input-feedback {
  color: red;
  height: 5px;
  margin-top: -12px;
}

table {
  font-family: arial, sans-serif;
  border-collapse: collapse;
  width: 100%;
}

td,th {
  border: 1px solid #fafafa;
  text-align: left;
  padding: 8px;
}
```

У директорії actions знаходяться файли, які визначають дії, що ініціюються користувачем. Директорія reducers містить ред'юсери – це функції, які відповідають за управління станом даних у проекті відповідно до виконуваних користувачем дій (actions). Влаштування директорій reducers та actions відображено на рисунку 4.7.

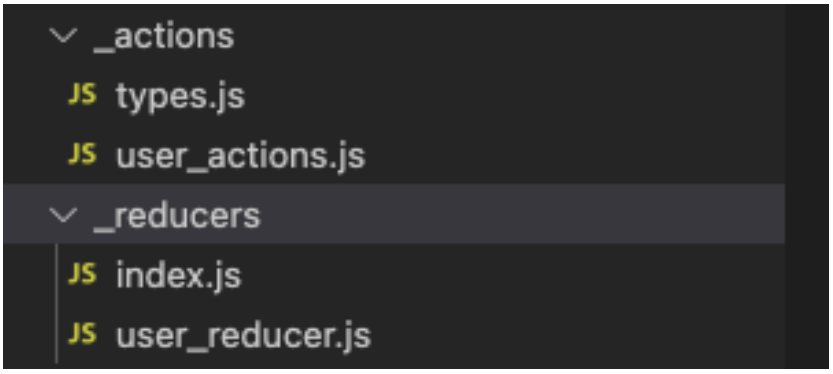


Рисунок 4.7 − Папки actions та reducers

Приклад action:

```
export function addToLikedPosts(_id) {
  const request = axios
    .get(`${USER_SERVER}/addToLikedPosts?articleId=${_id}`)
    .then((response) => response.data);

  return {
    type: ADD_TO_LIKED_POSTS_USER,
    payload: request,
  };
}
```

Приклад редюсера:

```
export default function (state = {}, action) {
  switch (action.type) {
    case ADD_TO_LIKED_POSTS_USER:
      return {
        ...state,
        userData: {
          ...state.userData,
          likedPosts: action.payload,
        },
      };
    case REMOVE_LIKED_POSTS_ITEM_USER:
      return {
        ...state,
```

```
        likedPostsDetail: action.payload.likedPostsDetail,
        userData: {
          ...state.userData,
          likedPosts: action.payload.likedPosts,
        },
      };
    default:
      return state;
  }
}
```

Директорія assets містить зображення, які застосовуються в компонентах вебаплікації, як можна побачити на рисунку 4.8.

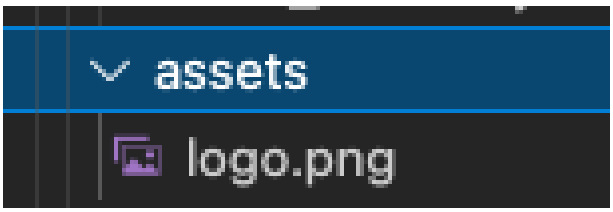


Рисунок 4.8 – Папка assets

Компоненти, які формують структуру застосунку, розміщені в директорії components. Організація цієї папки ілюстрована на малюнку 4.9. У даному проекті компоненти були створені за допомогою патерну React JSX, що дозволяє в межах одного файлу комбінувати HTML-розмітку та логіку обробки даних.

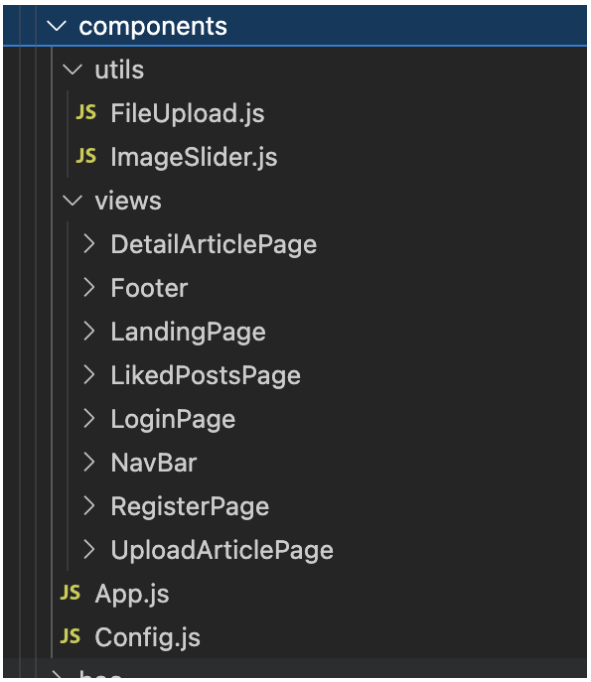


Рисунок 4.9 – Структура папки components

Приклад компоненту:

```
import React, { useEffect, useState } from "react";
import { useDispatch } from "react-redux";
import {
  getLikedPostsItems,
  removeLikedPostsItem,
} from "../../../_actions/user_actions";
import UserCardBlock from "./Sections/UserCardBlock";

function LikedPostsPage(props) {
  const dispatch = useDispatch();

  const [ShowTotal, setShowTotal] = useState(false);

  useEffect(() => {
    let likedPostsItems = [];
    if (props.user.userData && props.user.userData.likedPosts) {
      if (props.user.userData.likedPosts.length > 0) {
        props.user.userData.likedPosts.forEach((item) => {
          likedPostsItems.push(item.id);
        });
        dispatch(
          getLikedPostsItems(likedPostsItems, props.user.userData.likedPosts)
        );
      }
    }
  }, [props.user.userData]);

  const removeFromLikedPosts = (articleId) => {
    dispatch(removeLikedPostsItem(articleId)).then((response) => {
      if (response.payload.likedPostsDetail.length <= 0) {
        setShowTotal(false);
      }
    });
  };

  return (
    <div style={{ width: "85%", margin: "3rem auto" }}>
      <h1>Вподобані статті</h1>
      <div>
        <UserCardBlock
          articles={props.user.likedPostsDetail}
          removeItem={removeFromLikedPosts}
        />
      </div>
    </div>
  );
}

export default LikedPostsPage;
```

4.1.2 Реалізація методів поліпшення технічних характеристик системи

Традиційний підхід до роботи з API вебсервісів часто пов'язаний з прямими запитами до сервера та управлінням відповідями в контексті компонентів. Це може призвести до дублювання коду, зниження продуктивності та ускладнення відстеження стану запитів. У цьому розділі розглянуто, як застосування архітектури на базі Redux Toolkit і Redux-Saga може поліпшити технічні характеристики системи порівняно з традиційними методами.

Швидкодія: Використовуючи Redux-Saga, було реалізувано централізоване управління асинхронними операціями, знижуючи навантаження на основний потік виконання і підвищуючи реактивність інтерфейсу. Саги дозволяють ефективно обробляти паралельні запити до API, уникаючи блокування інтерфейсу користувача і, відповідно, підвищуючи загальну швидкість роботи застосунку.

Надійність: Redux Toolkit спрощує управління станом, забезпечуючи зменшення помилок завдяки зменшенню кількості бойлерплейту та використанню функцій на кшталт createReducer та createAction. Це додає додаткову надійність у відстеженні станів та подій у системі.

Зменшення кількості помилок: Завдяки типізації у TypeScript і чітко визначеним шаблонам управління станом і сайд-ефектами, було мінімізувано ризик виникнення помилок в часі розробки та експлуатації. Такий підхід також сприяє легшому налагодженню та тестуванню коду.

Посилення контролю над станами: Redux-Saga забезпечує детальний контроль над асинхронними операціями, дозволяючи розробникам легко реагувати на складні послідовності подій та впорядковувати асинхронну логіку відправки та обробки даних.

Приклади коду:

Управління асинхронними операціями з Redux-Saga. Як бачимо на цьому прикладі, розробник може контролювати порядок операції і бачити результат на кожному етапі:

```
function* fetchUserData(action) {
  try {
    const user = yield call(Api.fetchUser, action.payload.userId);
    yield put({type: 'FETCH_USER_SUCCESS', user});
  } catch (e) {
    yield put({type: 'FETCH_USER_FAILED', message: e.message});
  }
}
function* userSaga() {
  yield takeEvery('FETCH_USER_REQUESTED', fetchUserData);
}
```

Організація редюсерів з Redux Toolkit. Для кожної дії є окремий метод, який дозволяє зручно міняти будь-яке значення об’єкту стану:

```
const usersSlice = createSlice({
  name: 'users',
  initialState,
  reducers: {
    fetchUserRequested(state) {
      state.loading = true;
    },
    fetchUserSuccess(state, action) {
      state.loading = false;
      state.user = action.payload.user;
      state.error = null;
    },
    fetchUserFailed(state, action) {
      state.loading = false;
      state.error = action.payload.message;
    },
  },
});
```

Приклад селекторів, які використовуються для витягування та обчислення похідних даних зі стану, що дозволяє всім компонентам отримувати лише необхідну інформацію.

```
import { createSelector } from '@reduxjs/toolkit';
const selectUsersDomain = (state) => state.users;
```

```
const selectAllUsers = createSelector(
  [selectUsersDomain],
  (usersState) => usersState.entities
);

const selectUsersLoading = createSelector(
  [selectUsersDomain],
  (usersState) => usersState.loading
);

const selectUsersError = createSelector(
  [selectUsersDomain],
  (usersState) => usersState.error
);
```

Використання селекторів у компоненті React:

```
const UserComponent = () => {
  // Витягуємо дані зі стану за допомогою хука useSelector і вище визначених селекторів
  const allUsers = useSelector(selectAllUsers);
  const loading = useSelector(selectUsersLoading);
  const error = useSelector(selectUsersError);

  // Рендеримо UI в залежності від стану
  if (loading === 'pending') return <div>Loading...</div>;
  if (error) return <div>Error: {error}</div>;

  return (
    <ul>
      {allUsers.map((user) => (
        <li key={user.id}>{user.name}</li>
      ))}
    </ul>
  );
};
```

Підхід, що базується на Redux Toolkit і Redux-Saga, не тільки забезпечує більш ефективне управління станом та асинхронними процесами у порівнянні з

традиційними методами, але й значно підвищує швидкодію, надійність та контроль над станами в системі. Це дозволяє створювати більш якісні та легко підтримувані вебзастосунки.

Програмний код подано у додатку А.

4.2 Результати тестування програмного засобу та їх аналіз

4.2.1 Вибір методів тестування

В області розробки програмного забезпечення, основна мета тестування — це виявлення та усунення недоліків для покращення характеристик як продуктивності, так і безпеки програм, включаючи поліпшення враження користувачів від використання продукту. Це сприяє підвищенню задоволеності замовників і знижує ризик виникнення проблем у кінцевих користувачів, що може статися в разі запуску програми без глибокого тестування.

Тестування в інформаційних технологіях поділяється на дві головні категорії: тестування функціональності та тестування нефункціональних характеристик.

Функціональне тестування охоплює перевірку роботи різних функцій програмного застосунку, з метою забезпечення досягнення очікуваних результатів. Воно включає різноманітні підходи, як-от:

– тестування окремих модулів;
– перевірка інтеграції компонентів;
– комплексне тестування системи;
– регресивне тестування;
– тестування з використанням методів білої та чорної скриньок;
– тестування інтерфейсів.

Функціональне тестування може виконуватися як вручну, так і з використанням автоматизованих інструментів.

Нефункціональне тестування фокусується на аспектах, які не стосуються прямо функцій програми, але важливі для забезпечення якості, наприклад, ефективності, надійності, зручності використання, безпеки. Цей тип тестування зазвичай включає:

– перевірку продуктивності;

– тестування безпеки;

– випробування на витривалість;

– перевірка сумісності;

– оцінка зручності користування;

– тестування масштабованості;

– стрес-тести;

– перевірка відповідності стандартам;

– оцінка ефективності;

– тестування системи на стійкість до критичних помилок;

– перевірка локалізації.

Нефункціональне тестування, як правило, виконується з використанням спеціалізованих інструментів через його складність.

Спочатку розглянемо окремо методи тестування технологій Redux-Saga та Redux-Toolkit.

Тестування Redux Saga:

– Unit-тести генераторів: тести на окремі генератори для перевірки правильності їх логіки та поведінки.

– Integration тести для функцій-роботів (saga workers): тести, які перевіряють, чи працюють функції-робітники (workers) відповідно до очікувань при запуску певних дій або подій.

– End-to-End (E2E) тести: перевірка взаємодії між Redux Saga та іншими частинами програми, такими як інтерфейс користувача чи мережеві запити.

Тестування Redux Toolkit:

– Unit-тести для дій (actions) та редукторів: перевірка, чи коректно працюють функції дій та редуктори в Redux Toolkit.

– Integration тести для магазину (store): перевірка інтеграції між різними частинами магазину (store) Redux, такими як дії, редуктори та підписка на зміни стану.

– UI тести: тестування взаємодії між Redux Toolkit та інтерфейсом користувача.

Інструменти для тестування:

– Jest: популярний фреймворк для тестування JavaScript, який можна використовувати для писання unit-тестів та інтеграційних тестів.

– Redux Saga Test Plan: бібліотека для тестування Redux Saga, яка допомагає зручно тестувати побічні ефекти та генератори.

– Redux Toolkit вбудовані функції для тестування: Redux Toolkit надає інструменти для тестування, такі як configureStore(), які можна використовувати для підготовки магазину до тестування.

Для тестування взаємодії Redux-Toolkit та Redux-Saga у контексті роботи з API вебресурсів, можна використовувати кілька методів, з акцентом на модульному тестуванні. Ось декілька підходів: модульні тести для Reducers, тестування Saga, тестування інтеграції, тестування API Викликів, Snapshot Тестування та тестування UI Компонентів. Кожен з підходів до тестування має свої аспекти.

Модульні тести для Reducers (Redux-Toolkit):

– використання бібліотеки Jest для створення тестових сценаріїв (автоматизовані тести з Jest дозволяють легко інтегрувати тестування у процес розробки, забезпечуючи постійну перевірку якості коду);

– модульні тести дозволяють перевіряти кожен reducer окремо, забезпечуючи точність і впевненість у їх роботі;

– тестування чистоти reducers, перевірка, що вони повертають правильний стан для різних вхідних даних;

– тестування обробки різних action types;

Тестування Saga (Redux-Saga):

– використання redux-saga-test-plan для модульного тестування sagas;

– перевірка, що sagas правильно викликають API, обробляють різні відповіді та помилки;

– тестування ланцюжків ефектів (effects), таких як call, put, takeLatest;

Тестування інтеграції:

– використання redux-mock-store для створення мокованого store, що дозволяє тестувати взаємодію між reducers та sagas;

– симуляція різних сценаріїв, таких як успішні та невдалі запити до API;

Тестування API Викликів:

– використання nock або axios-mock-adapter для імітації запитів до API;

– перевірка, що sagas правильно обробляють відповіді та помилки API;

Snapshot Тестування:

– використання jest для створення snapshot тестів станів store після різних дій;

Тестування UI Компонентів:

– інтеграція з React компонентами, використання react-testing-library для тестування, як UI відповідає на зміни в стані Redux.

Провівши аналіз методів тестування, було прийнято рішення використовувати модульне та ручне тестування, а також для автоматизації модульного тестування використовувати бібліотеку Jest. Обидва ці підходи до тестування, модульне та ручне, забезпечують різні, але взаємодоповнюючі переваги. Модульне тестування ідеально підходить для перевірки внутрішньої логіки та ізольованих компонентів системи, в той час як ручне тестування відмінно підходить для визначення користувацького досвіду та взаємодії в реальних умовах. Комбінація обох підходів забезпечує глибоке розуміння як індивідуальних частин системи, так і її загальної функціональності. При модульному тестуванні важливо зосередитись на тестуванні індивідуальних частин системи (reducers, sagas, action creators) в ізоляції, щоб забезпечити, що кожен компонент працює належним чином.

При модульному тестуванні використовуються автоматизовані тести, які можуть бути виконані швидко та в повторюваних умовах, що забезпечує широке

покриття коду та виявлення помилок на ранніх стадіях розробки. Застосування Jest дозволяє виконувати швидке та ефективне тестування завдяки його функціям мокування та асинхронного тестування, що є особливо важливим при роботі з Redux-Saga та асинхронними операціями. З іншого боку, ручне тестування залишається незамінним для перевірки тих аспектів, де необхідно людське сприйняття, наприклад, для оцінки зручності інтерфейсу та загального користувацького досвіду.

Додатково, ручне тестування дозволяє виявити складні випадки використання, які можуть не бути очевидними при модульному тестуванні. Це включає сценарії, що вимагають комплексного розуміння бізнес-процесів та можуть включати взаємодію між різними частинами системи, які важко імітувати автоматично. Однак, модульне тестування є більш ефективним при частому перекодуванні та рефакторингу коду, оскільки воно дозволяє швидко перевірити, що зміни не порушили існуючу функціональність.

В результаті, інтеграція Jest у процес модульного тестування дає можливість для налаштування тестових сценаріїв, що відтворюють різноманітні випадки використання та забезпечення високого рівня кодової надійності. Вибір між модульним та ручним тестуванням не є взаємовиключним; натомість, вони слугують для підтримки одне одного, забезпечуючи більш повне тестування та вищу якість програмного продукту.

4.2.2 Розробка тестових сценаріїв

Мета модульного тестування полягає у перевірці програмного забезпечення шляхом проведення емпіричних досліджень його відповідності вимогам, зазначеним у специфікації. Таблиця 4.1 містить список тестових сценаріїв та опис очікуваного результату для кожної функції програми. Вона слугує також як контрольний список для розробників та тестувальників, забезпечуючи систематичний підхід до перевірки кожної функції та гарантуючи, що жодна частина програми не залишиться без уваги під час тестування.

Таблиця 4.1 – Тестові випадки модульного тестування

Ч. ч.	Steps (Кроки/дії)	Expected Results (Очікувані результати)	Actual results (Реальні результати)
1	2	3	4
1	Відкрити вебсайт	Повинна відкритись головна сторінка.	Головна сторінка успішно відкрилась.
2	Відкрити список статтей	Повиннен піти запит до API на отримання списку статей, пройти успішно за невеликий час, та повернути як результат масив об’єктів статтей.	Запит на отримання статтей пішов, в цей час на екрані з’явився лоадер – індикатор завантаження ресурсу. Запит пройшов менше ніж за секунду без помилок, лоадер зник, а статті з`явились на екрані.
3	Натиснути кнопку «Login»	Повинна з’явитися сторінка входу в свій обліковий запис.	На екрані відкрилась сторінка з формою для авторизації користувача.
4	Авторизація	Після заповнення форми авторизації некоректними даними, повинна спрацювати валідація.	Ввівши неправильні дані, з’являються повідомлення про помилки під кожним полем, яке є невалідним, ці поля підсвічуються червоним кольором.
5	Авторизація	Ввівши правильні дані, повинен піти запит на авторизацію, користувач повинен отримати авторизаційний токен щоб вважатись авторизованим.	Запит на логін був відправлений. На екрані з’явився лоадер – індикатор завантаження. Запит пройшов за пів секунди, користувач авторизувався
6	Реєстрація	Після заповнення форми реєстрації некоректними даними, повинна спрацювати валідація.	Ввівши неправильні дані, з’являються повідомлення про помилки під кожним полем, яке є невалідним, ці поля підсвічуються червоним кольором.

Продовження таблиці 4.1

7	Реєстрація	Ввівши правильні дані, повинен піти запит на реєстрацію, користувач повинен створитись профіль.	Запит на реєстрацію був відправлений. На екрані з’явився лоадер – індикатор створення профілю. Запит пройшов за секунду, користувач зайшов в новостворений профіль.
8	Скидання паролю	Ввівши електронну пошту, якої немає в базі даних, повинна спрацювати валідація.	Ввівши неправильний e-mail, з’явилось повідомлення про помилки під кожним полем, яке є невалідним, воно підсвітилось червоним кольором.
9	Скидання паролю	Ввівши правильні дані, повинен піти запит на скидання паролю, користувачу маю прийти лист-підтвердження на електронну пошту.	Запит на скидання паролю був відправлений. На екрані з’явився лоадер – індикатор обробки запиту. Потім було показано повідомлення про те, що запит успішний і через декілька хвилин на пошту був відправлений лист.
10	Зміна паролю	Перейшовши з електронної пошти за посиланням і ввівши некоректний новий пароль або його підтвердження, має спраювати валідація.	Ввівши неправильний пароль або його підтвердження, з’явилось повідомлення про помилку під відповідним полем, воно підсвітилось червоним кольором.
11	Зміна паролю	Ввівши правильні дані, повинен піти запит на зміну паролю користувача.	Запит на зміну паролю був відправлений. На екрані з’явився лоадер – індикатор обробки запиту. Запит пройшов за третину секунди, пароль користувача змінився.
12	Отримання даних профілю	Користувач зайшов на сторінку свого профілю. Очікується вивів інформації користувача.	Після входу на сторінку профілю користувача, на екрані з’явився лоадер – індикатор обробки запиту отримання даних. Через 0.4 секунди дані було отримано та відображено на екрані.

Кінець таблиці 4.1

13	Зміна даних профілю	Після введення некоректниих даних користувача, повинна спрацювати валідація.	Змінивши дані на некоректні, з’являються повідомлення про помилки під кожним полем, яке є невалідним, ці поля підсвічуються червоним кольором.
14	Зміна даних профілю	Ввівши правильні дані, повинен піти запит на зміну даних профілю користувача.	Запит на зміну даних профілю користувача був відправлений. На екрані з’явився лоадер – індикатор обробки запиту. Запит пройшов за третину секунди, дані користувача змінились.

В результаті перевірки кожного компоненту встановлено, що досягнуті результати відповідають очікуванням.

4.2.3 Аналіз результатів тестування

Модульне тестування було проведено за допомогою бібліотеки Jest.

Приклад коду тестування авторизації користувача:

```
import { runSaga } from 'redux-saga';
import { handleLogin } from './loginSaga';
import { loginSuccess, loginFailure } from './actions';
import { api } from './api'; // Це ваша абстракція API

// Мок API виклику
jest.mock('./login', () => ({
  loginUser: jest.fn(),
}));

describe('saga fdnjhbpfws]', () => {
  it('повинна викликати api і відправити дію успіху при успішному вході', async ()
=> {
    const dispatchedActions = [];
    const mockUser = { username: 'test', password: 'test' };
    api.loginUser.mockResolvedValue(mockUser);
    const store = {
      dispatch: (action) => dispatchedActions.push(action),
    };
```

```
    await runSaga(store, handleLogin, { username: 'test', password: 'test'
}).done;
    expect(api.loginUser).toHaveBeenCalledWith('test', 'test');
    expect(dispatchedActions).toContainEqual(loginSuccess(mockUser));
  });

  it('повинна викликати api і відправити дію невдачі при невдалому вході', async
() => {
    const dispatchedActions = [];

    const error = 'Невірні дані для входу';
    api.loginUser.mockRejectedValue(new Error(error));

    const store = {
      dispatch: (action) => dispatchedActions.push(action),
    };

    await runSaga(store, handleLogin, { username: 'test', password: 'wrong'
}).done;

    expect(api.loginUser).toHaveBeenCalledWith('test', 'wrong');
    expect(dispatchedActions).toContainEqual(loginFailure(error));
  });
});
```

Результати тестування програмних модулів на рисунках 4.1 – 4.13. На цих рисунках показано, що тестові випадки, наведені у таблиці 4.1, пройдені успішно.

Рисунок 4.1 – Результат тестування запуску головної сторінки

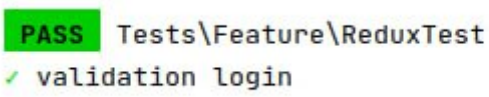


Рисунок 4.2 – Результат тестування валідації на формі авторизації

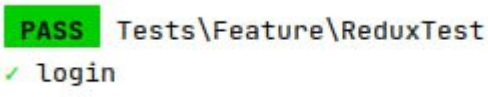


Рисунок 4.3 – Результат тестування запиту на авторизацію користувача

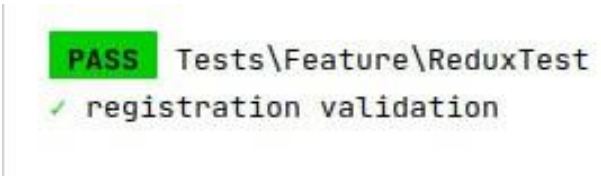


Рисунок 4.4 – Результат тестування валідації на формі реєстрації

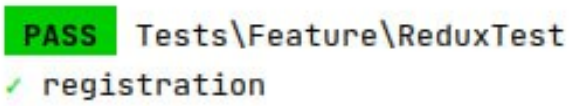


Рисунок 4.5 – Результат тестування запиту на реєстрацію користувача

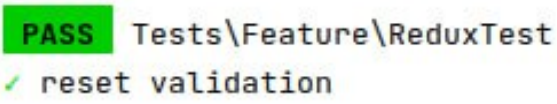


Рисунок 4.6 – Результат тестування валідації на формі скидання паролю

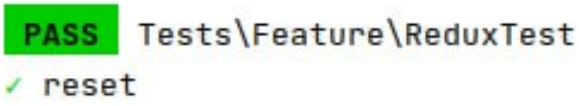


Рисунок 4.7 – Результат тестування запиту на скидання паролю

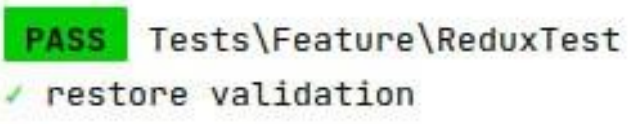


Рисунок 4.8 – Результат тестування валідації на формі введення нового паролю

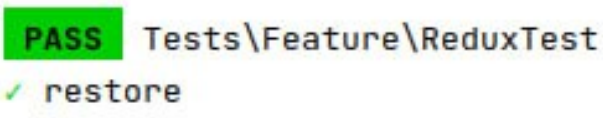


Рисунок 4.9 – Результат тестування запиту на створення нового паролю

Рисунок 4.10 − Результат тестування запиту на отримання даних користувача

Рисунок 4.11 − Результат тестування запиту на зміну зображення профілю користувача

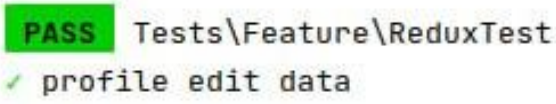


Рисунок 4.12 − Результат тестування запиту на зміну даних користувача

Рисунок 4.13 − Результат тестування запиту на отримання статтей

4.3 Оцінка ефективності удосконаленого методу вирішення задачі

Для оцінки ефективності удосконаленого методу поєднання технологій Redux-Toolkit та Redux-Saga у роботі з API вебресурсів, було використано дані, отримані з двох сценаріїв: з використанням удосконаленого методу та без його використання. Оцінка здійснюється на основі порівняння часу відгуку API при різній кількості запитів. Для наочності було складено графік порівняння часу відгуку API з використанням удосконаленого методу (Redux-Toolkit та Redux-

Saga) та без його використання та таблицю порівняння часу відгуку API при різній кількості запитів (рисунок 4.14).

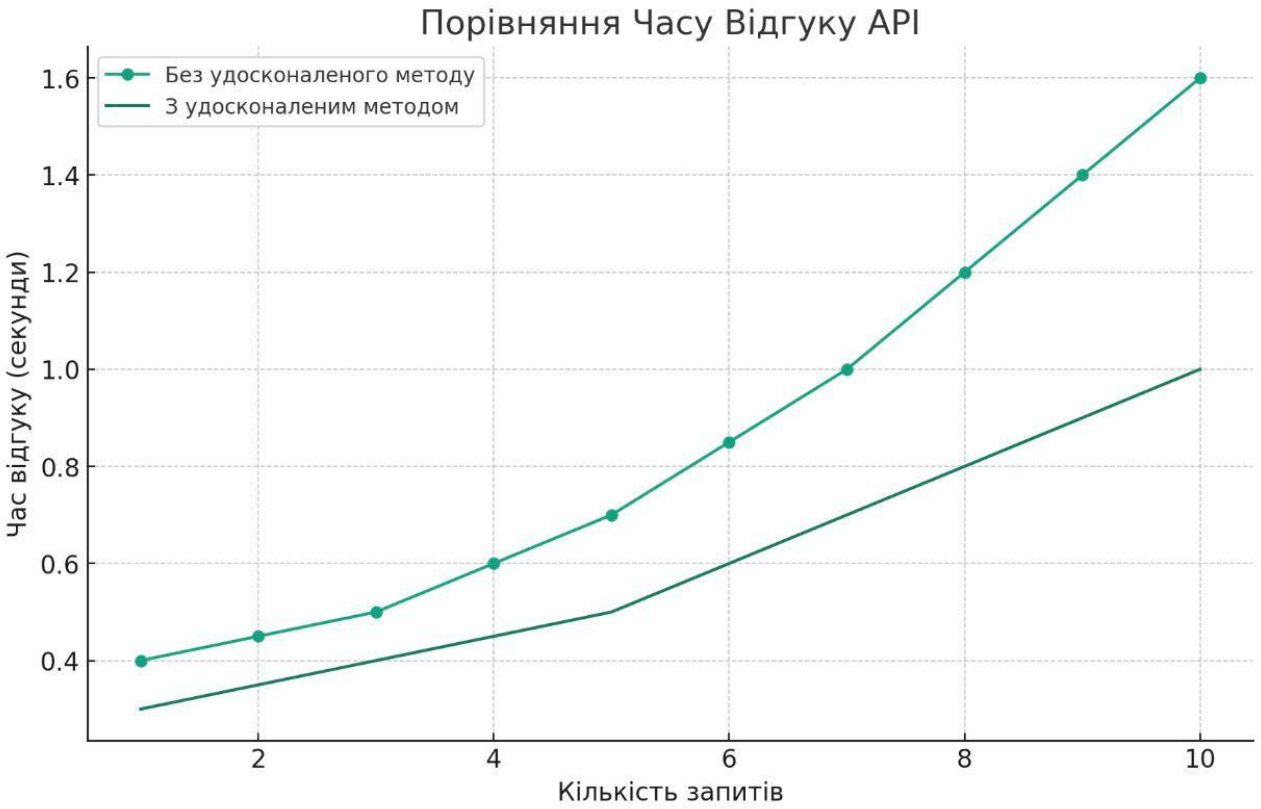


Рисунок 4.14 − Графік порівняння часу відгуку API з використанням удосконаленого методу (Redux-Toolkit та Redux-Saga) та без його використання

На рисунку 4.14 представлено порівняння часу отримання результатів від API з використанням удосконаленого методу (Redux-Toolkit та Redux-Saga) та без його використання. Як видно, з використанням удосконаленого методу час відгуку API знижується при збільшенні кількості запитів порівняно з традиційним підходом. Це може свідчити про ефективність удосконаленого методу у плані швидкості обробки запитів.

Графік показує, що з використанням удосконаленого методу, час відгуку API був нижчим у кожному з тестових випадків. Це вказує на покращену ефективність обробки запитів. Наприклад, при одному запиті час відгуку скоротився з 0.4 секунди до 0.3 секунди, і ця тенденція зберігалася при збільшенні кількості запитів.

Такий підхід може поліпшити масштабованість системи, дозволяючи легше адаптуватися до змін у вимогах до продуктивності та кількості користувачів без необхідності значних змін у архітектурі. Ці результати можуть бути особливо важливими для застосунків з високими вимогами до швидкодії, таких як фінансові платформи або ігри в реальному часі, де кожна мілісекунда затримки може вплинути на загальне сприйняття користувачем якості продукту.

Таблиця 4.2 – Порівняння часу відгуку API при різній кількості запитів з використанням удосконаленого методу (Redux-Toolkit та Redux-Saga) та без його використання

№ Запиту	Час відгуку без удосконаленого методу (секунди)	Час відгуку з удосконаленим методом (секунди)
1	0.4	0.3
2	0.45	0.35
3	0.5	0.4
4	0.6	0.45
5	0.7	0.5
6	0.85	0.6
7	1.0	0.7
8	1.2	0.8
9	1.4	0.9
10	1.6	1.0

Ця таблиця відображає порівняння часу відгуку API при різній кількості запитів, демонструючи, як впливає використання удосконаленого методу на продуктивність додатку.

Таблиця, що містить детальні дані про час відгуку для кожного кількості запитів, додатково підтверджує ефективність удосконаленого методу. Ми спостерігаємо послідовне зниження часу відгуку при використанні удосконаленого методу порівняно з традиційним підходом.

Зменшення часу відгуку є важливим чинником у підвищенні загальної продуктивності вебзастосунків. Це не тільки покращує користувацький досвід, але й забезпечує більшу стабільність і надійність при обробці великої кількості

запитів. У сучасних вебзастосунках, де велика увага приділяється часу відгуку та ефективності взаємодії з сервером, такі поліпшення можуть мати значний вплив на загальну якість та конкурентоспроможність продукту.

На підставі аналізу даних з графіка та таблиці, можна зробити висновок, що удосконалений метод поєднання Redux-Toolkit та Redux-Saga ефективно покращує час відгуку API, забезпечуючи більш швидку та ефективну взаємодію з вебресурсами. Це робить його цінним інструментом для розробників, які прагнуть оптимізувати продуктивність своїх вебзастосунків, особливо в умовах високої конкуренції та зростаючих вимог до швидкодії.

Застосування удосконаленого методу також сприяє кращій масштабованості системи, дозволяючи легко адаптуватися до зростаючого обсягу запитів без зниження продуктивності. Оптимізація взаємодії з API через такі методи може суттєво знизити час завантаження даних, що є критичним для застосунків, які обробляють великі об'єми інформації або вимагають негайної відповіді, як наприклад, в реальному часі аналітичні дашборди.

Крім того, імплементація цього методу може спростити архітектуру системи, зменшуючи потребу в складних операціях оптимізації на стороні сервера, що може призвести до зниження витрат на інфраструктуру та підтримку. У такий спосіб, використання Redux-Toolkit та Redux-Saga не лише покращує безпосередню продуктивність відгуку API, але й вносить стратегічний вклад у загальну економічну ефективність розробки та експлуатації вебзастосунків.

4.4 Інтеграція та налаштування програмного засобу

Безперебійна робота платформи забезпечується за умови, що апаратні засоби відповідають таким вимогам:

- 1024 Мб вбудованої пам'яті;
- 3 Гб оперативної пам'яті;
- процесор з двома ядрами;
- Інтернет-з'єднання зі швидкістю не менше ніж 3 Мбіт/с.

Для розгортання проекту на комп'ютері має бути встановлено Node.JS та базу даних MongoDB. Після відкриття директорії проекту слід запустити два командних рядка (термінали). Запуск клієнтської частини здійснюється через такі команди, введені у термінал:

- перехід у директорію client;
- виконання команди npm install;
- запуск команди npm start.

Для запуску серверної частини введіть наступні команди:

- перейдіть у директорію server;
- виконайте команду npm install;
- запустіть виконання команди node index.js.

4.5 Висновки

У даному розділі детально розглянуто на прикладі інтеграцію і взаємодію технологій Redux-Toolkit та Redux-Saga для ефективної роботи з API вебресурсів.

Значна увага приділяється деталям реалізації, включаючи створення ефективних саг, обробку помилок, та керування паралельними процесами. Проведено ретельне модульне тестування, що включає перевірку кожного аспекту інтеграції, а також різноманітні тестові сценарії для оцінки роботи системи в різних умовах, зокрема виконано тестування навантаження, щоб перевірити стабільність та продуктивність системи при різних умовах. Для забезпечення максимальної ефективності, розглянуто ключові патерни проектування, які можуть бути застосовані при використанні Redux-Toolkit та Redux-Saga, включно з такими підходами як debounce effects для оптимізації частоти запитів та selectors для мемоізації селективних запитів до стану. Також проаналізовано стратегії кешування, які можуть додатково зменшити навантаження на сервери API та прискорити відгук на кінцевому користувачеві. Ці стратегії, разом із ретельним тестуванням та аналізом, формують міцну основу для створення стійких і

масштабованих вебзастосунків, здатних витримувати високі навантаження та забезпечувати високий рівень доступності сервісу.

Удосконалення архітектури вебзастосунків за допомогою цих технологій також сприяє кращому розділенню відповідальності між різними частинами системи, що робить код більш читабельним та легшим для підтримки. Це дозволяє розробникам швидше реагувати на зміни вимог та ефективніше керувати станом застосунків, а також покращує можливості для співпраці в багатоукладних командних проектах. Завдяки цьому, вебзастосунки стають більш адаптивними до змін, що в кінцевому результаті веде до зниження загальних витрат на розробку та підтримку.

Враховуючи результати емпіричного дослідження, зазначено, що поєднання Redux-Toolkit і Redux-Saga сприяє збільшенню продуктивності та надійності вебзастосунків. Особливо підкреслено поліпшення в обробці паралельних запитів, що підвищило ефективність виконання запитів та управлінні складними станами, що оптимізувало обробку даних. Також відмічено, що цей підхід значно знижує ймовірність помилок у роботі з API та підвищує загальну стабільність системи.

На підставі аналізу висновки свідчать про значне поліпшення ефективності розробки та підтримки вебзастосунків, що використовують ці технології.

ВИСНОВКИ

У рамках цієї кваліфікаційної роботи було проведено ґрунтовне дослідження сучасних підходів до управління станами в вебзастосунках, зокрема, зосередившись на інтеграції технологій Redux-Toolkit та Redux-Saga для роботи з API вебресурсів. Встановлено, що ці технології мають значний потенціал у покращенні ефективності та зручності розробки, проте їхня комбінація до цього часу не мала чіткої методології. Таким чином, основною метою дослідження стала розробка методу, що дозволить оптимізувати використання цих технологій разом.

У першому розділі кваліфікаційної роботи було проаналізовано існуючі методи управління станами в вебзастосунках, з особливим акцентом на Redux-Toolkit та Redux-Saga. Досліджено їх основні переваги, недоліки та потенційні можливості інтеграції. Окрім цього, проведено аналіз використання API в сучасних вебзастосунках, що дало змогу виявити ключові проблеми та обмеження, з якими стикаються розробники.

У другому розділі була розроблена концепція поєднання Redux-Toolkit та Redux-Saga в єдину систему для оптимізації роботи з API. Досліджено потенційні синергетичні ефекти від їх комбінування, а також визначено найкращі практики інтеграції. Розглянуто можливі сценарії використання та розроблено набір рекомендацій щодо ефективної імплементації комбінованого підходу.

Третій розділ присвячено практичній реалізації запропонованого методу. Була розроблена програмна модель, що демонструє можливості комбінованого використання Redux-Toolkit та Redux-Saga для роботи з API. У цьому розділі також виконано аналіз вимог до програмного забезпечення, розроблено його структуру та архітектуру. Для реалізації було використано сучасні технології, такі як React, Redux, Node.js.

У четвертому та заключному розділі зосереджено увагу на тестуванні та аналізі результатів використання розробленого методу. Окрему увагу було приділено вибору технологій, які б найкращим чином підтримували реалізацію запропонованого методу, включаючи React, Node.js, а також інструменти для

тестування, такі як Jest. Тестування показало значне поліпшення в швидкості та надійності вебзастосунків, особливо при роботі з складними асинхронними запитами та великими обсягами даних. Результати демонструють, що комбінація Redux-Toolkit і Redux-Saga не тільки покращує продуктивність, але й сприяє більш організованій та структурованій розробці вебзастосунків.

Підсумовуючи вищезазначене, можна стверджувати, що розроблений метод поєднання технологій Redux-Toolkit та Redux-Saga для роботи з API вебресурсів має значну наукову новизну та практичну цінність.

Цей метод не тільки відкриває нові можливості для розробників вебзастосунків, але й сприяє підвищенню ефективності та якості розробки, забезпечуючи більшу гнучкість у вирішенні завдань, пов'язаних з управлінням станами та асинхронними операціями.

Результати дослідження показали, що інтеграція Redux-Toolkit та Redux-Saga дозволяє досягти високого рівня оптимізації при роботі з великими та складними даними, зменшуючи час на розробку та покращуючи загальну структуру взастосунків. Це, в свою чергу, відкриває шлях для створення більш масштабованих та надійних вебсистем, які можуть ефективно взаємодіяти з різноманітними вебAPI.

З огляду на широке застосування вебтехнологій в сучасному світі, важливість цього дослідження не може бути переоцінена. Воно надає нові інструменти та методології, які дозволять IT-компаніям розвивати більш ефективні, швидкі та надійні вебзастосунки, що відповідають сучасним вимогам ринку. Особливо це важливо для проектів, які вимагають високої швидкості обробки даних та взаємодії з різноманітними внутрішніми та зовнішніми API.

Крім того, результати дослідження сприятимуть подальшому розвитку та удосконаленню вебтехнологій. Інтеграція та оптимізація процесів розробки є ключовими аспектами у пошуку нових та ефективніших підходів до створення вебзастосунків.

Розроблений метод може бути використаний як основа для подальших досліджень у цій області, надаючи міцний фундамент для інновацій.

Таким чином, результати дослідження за темою кваліфікаційної роботи мають наукову новизну та практичну цінність.

Результати дослідження опубліковані у фаховому науковому виданні [31].

Копії наукових публікацій подані у додатку Б.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1. Кваліфікаційна робота магістра: методичні вказівки щодо її виконання для студентів спеціальності «Інженерія програмного забезпечення» / Г. І. Радельчук. – Хмельницький : ХНУ, 2017. – 80 с.

2. Alex Banks, Eve Porcello. «Learning React: Modern Patterns for Developing React Apps». Хмельницький : O'Reilly Media, 2020. – 310 с.

3. Mark Erikson, Tereza Sokol. «Redux Toolkit Quick Start Guide: Create Complex and Scalable Applications with Modern Redux». Хмельницький : Packt Publishing, 2020. – 188 с.

4. David Nance. «Redux Quick Start Guide: A beginner's guide to managing app state with Redux». Хмельницький : Packt Publishing, 2018. – 156 с.

5. Marc Garreau, Will Faurot. «Redux in Action». Хмельницький : Manning Publications, 2018. – 320 с.

6. Redux Toolkit Documentation. Redux. URL: https://redux-toolkit.js.org/ (дата звернення: 03.09.2023).

7. Redux-Saga Documentation. Redux-Saga. URL: https://redux-saga.js.org/ (дата звернення: 05.09.2023).

8. Dan Abramov. «You Might Not Need Redux". Medium. URL: https://medium.com/@dan_abramov/you-might-not-need-redux-be46360cf367 (дата звернення: 06.09.2023).

9. Erik Rasmussen. «The Evolution of Redux». Medium. URL: https://medium.com/@erikras/the-evolution-of-redux-6c16e806b9e1 (дата звернення: 06.09.2023).

10. Mark Erikson. «Redux - Not Dead Yet!». Blog.isquaredsoftware.com. URL: https://blog.isquaredsoftware.com/2021/01/context-redux-differences/ (дата звернення: 09.09.2023).

11. Tania Rascia. «Understanding Redux: The World’s Easiest Guide to Beginning Redux». Taniarascia.com. URL: https://www.taniarascia.com/redux-react-guide/ (дата звернення: 11.09.2023).

12. Dmitriy Shekhovtsov. «Redux-Saga vs Redux-Thunk». Hacker Noon. URL: https://hackernoon.com/redux-saga-vs-redux-thunk-6a6c9f0f59b2 (дата звернення: 12.09.2023).

13. «Asynchronous Redux: Redux-Thunk vs. Redux-Saga». Fullstack Academy. URL: https://www.fullstackacademy.com/blog/asynchronous-redux-redux-thunk-vs-redux-saga (дата звернення: 16.09.2023).

14. «API Integration with Redux». Smashing Magazine. URL: https://www.smashingmagazine.com/api-integration-with-redux/ (дата звернення: 19.09.2023).

15. «Managing State in Modern React Apps with Redux Toolkit and TypeScript». LogRocket. URL: https://blog.logrocket.com/managing-state-in-modern-react-apps-with-redux-toolkit-typescript/ (дата звернення: 19.09.2023).

16. Redux-Saga Beginner Tutorial. Redux-Saga. URL: https://redux-saga.js.org/docs/introduction/BeginnerTutorial.html (дата звернення: 20.09.2023).

17. «React Redux Tutorial for Beginners: Learning Redux in 2021». Dave Ceddia's Blog. URL: https://daveceddia.com/redux-tutorial/ (дата звернення: 21.09.2023).

18. «Redux Toolkit with TypeScript: a Practical Tutorial». TypeScript Programming. URL: https://www.typescriptlang.org/docs/handbook/react-&-redux.html (дата звернення: 22.09.2023).

19. «Advanced Redux Patterns: Normalizing API Responses with Redux Toolkit». Valentinog.com. URL: https://www.valentinog.com/blog/redux/ (дата звернення: 26.09.2023).

20. «Redux-Saga for Simpler Side Effects in Redux Apps». The Net Ninja. URL: https://netninja.dev/p/redux-saga-for-simpler-side-effects-in-redux-apps (дата звернення: 30.09.2023).

21. «React, Redux, and Saga with TypeScript». Infoworld. URL: https://www.infoworld.com/article/3581765/react-redux-and-saga-with-typescript.html (дата звернення: 30.09.2023).

22. «Understanding Generators in Redux-Saga». Alligator.io. URL: https://alligator.io/redux/redux-saga/ (дата звернення: 02.10.2023).

23. «Testing Redux-Saga: A Beginner's Guide». Testim.io. URL: https://www.testim.io/blog/testing-redux-saga-a-beginners-guide/ (дата звернення: 05. 10.2023).

24. «How to Use Redux in Your React TypeScript App». Freecodecamp.org. URL: https://www.freecodecamp.org/news/how-to-use-redux-in-your-react-typescript-app/ (дата звернення: 10.10.2023).

25. «React Redux: An In-Depth Tutorial». Edureka. URL: https://www.edureka.co/blog/react-redux-tutorial/ (дата звернення: 14.10.2023).

26. «Redux Toolkit: The Standard Way to Write Redux». React Redux Firebase. URL: https://www.react-redux-firebase.com/docs/recipes/redux-toolkit.html (дата звернення: 20.10.2023).

27. «Building Modern Applications with Redux and Redux-Saga». Medium. URL: https://itnext.io/building-modern-applications-with-redux-and-redux-saga-368e5c2dbeb2 (дата звернення: 23.10.2023).

28. «Redux vs. MobX vs. Redux-Saga: Which Should You Choose?». JavaScript Scene. URL: https://medium.com/javascript-scene/redux-vs-mobx-vs-redux-saga-df0af5ae54d0 (дата звернення: 26.09.2023).

29. «The Definitive Guide to Redux Saga». Thoughtbot. URL: https://thoughtbot.com/blog/the-definitive-guide-to-redux-saga (дата звернення: 30.10.2023).

30. «Redux-Saga Tutorial: How Redux Saga Makes Handling Side Effects Easier». Upmostly. URL: https://upmostly.com/tutorials/redux-saga-tutorial (дата звернення: 03.11.2023).

31. Собко В.В. Метод поєднання технологій Redux-Toolkit та Redux-Saga для роботи з API вебресурсів. URL: https://kn.khmnu.edu.ua/wp-content/uploads/sites/18/apkn-2023-corpuspaper.pdf (дата звернення: 04.11.2023).

ДОДАТОК А

(обов’язковий)

ПРОГРАМНИЙ КОД

А.1 Програмний код файлу hooks/httpClient.ts

```
import { useState, useEffect } from 'react';
import { useDispatch } from 'react-redux';
import snakeCaseKeys from 'snakecase-keys';
import camelCaseKeys from 'camelcase-keys';

import {
  AxiosInstance,
  AxiosRequestConfig,
  AxiosResponse,
  AxiosError,
} from 'axios';

// actions
import { actions as logoutActions } from 'store/app/slice';

const useHttpClient = (
  httpClientInstance: AxiosInstance,
): [string | null, () => void] => {
  // local error state
  const [error, setError] = useState<string | null>(null);

  // actions
  const dispatch = useDispatch();
  const onLogout = () => dispatch(logoutActions.logoutTrigger());

  // const setPermissionsError = (data: IDataPayload) =>
  //   dispatch(globalActions.permisionsError(data));

  // request interceptor
  const reqInterceptor = httpClientInstance.interceptors.request.use(
    async (config: AxiosRequestConfig) => {
      setError(null);

      // formatting data
      if (config.data) {
        config.data =
          config.data instanceof FormData
            ? config.data
            : snakeCaseKeys(config.data, { deep: true });
      }

      // formatting headers
```

```
    const accessToken = await localStorage.getItem('access_token');
    if (accessToken) {
      config.headers.Authorization = `Bearer ${accessToken}`;
    }

    return config;
  },
  (err: any) => {
    setError(err);
    return Promise.reject(err);
  },
);

// response interceptor
const resInterceptor = httpClientInstance.interceptors.response.use(
  (res: AxiosResponse) => {
    return camelCaseKeys(res.data, { deep: true });
  },
  (err: AxiosError) => {
    let formattedMessage: string | null = null;

    if (err && err.response) {
      // errors handling
      switch (err.response.status) {
        case 401:
          onLogout();
          break;

        case 403:
          break;

        case 500:
          formattedMessage = 'Unknown error';
          break;

        default:
          break;
      }

      if (err.response.data.message) {
        formattedMessage = err.response.data.message;
      }

      if (err.response.data.errors) {
        Object.keys(err.response.data.errors).forEach(errorKey => {
          // @ts-ignore
          err.response.data.errors[errorKey].forEach((line: string) => {
            formattedMessage += `\r\n${line}`;
          });
        });
      }

      if (err.response.status !== 401) {
```

```
          setError(formattedMessage);
        }
      }

      return Promise.reject(err);
    },
  );

  // watch reqInterceptor & resInterceptor - eject request & response interceptors
  useEffect(
    () => () => {
      httpClientInstance.interceptors.request.eject(reqInterceptor);
      httpClientInstance.interceptors.response.eject(resInterceptor);
    },
    [
      reqInterceptor,
      resInterceptor,
      httpClientInstance.interceptors.request,
      httpClientInstance.interceptors.response,
    ],
  );

  const errorClearedHandler = () => {
    setError(null);
  };

  return [error, errorClearedHandler];
};

export default useHttpClient;
```

A.2 Програмний код файлу store/configureStore.ts

```
import { configureStore, getDefaultMiddleware } from '@reduxjs/toolkit';
import { createInjectorsEnhancer, forceReducerReload } from 'redux-injectors';
import createSagaMiddleware from 'redux-saga';

import { createReducer } from './reducers';

export function configureAppStore() {
  const reduxSagaMonitorOptions = {};
  const sagaMiddleware = createSagaMiddleware(reduxSagaMonitorOptions);
  const { run: runSaga } = sagaMiddleware;

  // Create the store with saga middleware
  const middlewares = [sagaMiddleware];

  const enhancers = [
    createInjectorsEnhancer({
      createReducer,
      runSaga,
    }),
```

```
  ];

  const store = configureStore({
    reducer: createReducer(),
    middleware: [
      ...getDefaultMiddleware({
        serializableCheck: false,
      }),
      ...middlewares,
    ],
    devTools:
      /* istanbul ignore next line */
      process.env.NODE_ENV !== 'production' ||
      process.env.PUBLIC_URL.length > 0,
    enhancers,
  });

  // Make reducers hot reloadable, see http://mxs.is/googmo
  /* istanbul ignore next */
  if (module.hot) {
    module.hot.accept('./reducers', () => {
      forceReducerReload(store);
    });
  }

  return store;
}
```

A.3 Програмний код файлу store/reducers.ts

```
/**
 * Combine all reducers in this file and export the combined reducers.
 */

import { combineReducers } from '@reduxjs/toolkit';

import { InjectedReducersType } from 'utils/types/injector-typings';

/**
 * Merges the main reducer with the router state and dynamically injected reducers
 */
export function createReducer(injectedReducers: InjectedReducersType = {}) {
  // Initially we don't have any injectedReducers, so returning identity function
to avoid the error
  if (Object.keys(injectedReducers).length === 0) {
    return state => state;
  } else {
    return combineReducers({
      ...injectedReducers,
    });
  }
}
```

А.4 Програмний код файлу store/modules/login/saga.ts

```
import { call, put, takeLatest } from 'redux-saga/effects';

import { actions as loginActions } from './slice';
import { actions as appActions } from 'store/modules/app/slice';
import * as services from './services';
import {
  setAuthAccessToken,
  setHasBeenAlreadyAuthBefore,
} from '../app/services';

/**
 * Login request/response handler
 */
export function* fetchLoginSaga({
  payload,
}: ReturnType<typeof loginActions.fetchDataTrigger>) {
  try {
    const { accessToken } = yield call(services.loginService, payload);

    yield put(loginActions.fetchDataSuccess());
    yield put(appActions.autoLoginSetToken({ token: accessToken }));
    yield call(setAuthAccessToken, accessToken);
    yield call(setHasBeenAlreadyAuthBefore);
  } catch (err) {
    yield put(loginActions.fetchDataFailed(err));
  } finally {
    yield put(loginActions.fetchDataFulfilled());
  }
}

/**
 * Root saga manages watcher lifecycle
 */
export function* loginSaga() {
  yield takeLatest(loginActions.fetchDataTrigger.type, fetchLoginSaga);
}
```

А.5 Програмний код файлу store/modules/login/selectors.ts

```
import { call, put, takeLatest } from 'redux-saga/effects';

import { actions as loginActions } from './slice';
import { actions as appActions } from 'store/modules/app/slice';
import * as services from './services';
import {
  setAuthAccessToken,
  setHasBeenAlreadyAuthBefore,
} from '../app/services';

/**
 * Login request/response handler
 */
export function* fetchLoginSaga({
  payload,
}: ReturnType<typeof loginActions.fetchDataTrigger>) {
  try {
    const { accessToken } = yield call(services.loginService, payload);

    yield put(loginActions.fetchDataSuccess());
```

```
    yield put(appActions.autoLoginSetToken({ token: accessToken }));
    yield call(setAuthAccessToken, accessToken);
    yield call(setHasBeenAlreadyAuthBefore);
  } catch (err) {
    yield put(loginActions.fetchDataFailed(err));
  } finally {
    yield put(loginActions.fetchDataFulfilled());
  }
}

/**
 * Root saga manages watcher lifecycle
 */
export function* loginSaga() {
  yield takeLatest(loginActions.fetchDataTrigger.type, fetchLoginSaga);
}
```

A.6 Програмний код файлу store/modules/login/services.ts

```
import { $apiAuthClient } from 'utils/request';
import { FetchTriggerPayload } from './types';

export function loginService(payload: FetchTriggerPayload) {
  const url = 'api/v1/login';
  return $apiAuthClient.post(url, payload);
}
```

A.7 Програмний код файлу store/modules/login/slice.ts

```
/*
 * Login Slice
 *
 */

import { PayloadAction } from '@reduxjs/toolkit';
import { createSlice } from 'utils/@reduxjs/toolkit';

import { initialState } from './state';
import { FetchPayloadErrorType, FetchTriggerPayload } from './types';

const slice = createSlice({
  name: 'login',
  initialState,
  reducers: {
    fetchDataTrigger(state, action: PayloadAction<FetchTriggerPayload>) {
      state.loading = true;
      state.done = false;
      state.error = null;
    },
    fetchDataSuccess(state) {
      state.done = true;
    },
    fetchDataFailed(state, action: PayloadAction<FetchPayloadErrorType>) {
      state.error = action.payload;
    },
    fetchDataFulfilled(state) {
      state.loading = false;
      state.done = false;
    },
  },
```

```
});

export const { actions, reducer, name } = slice;
```

A.8 Програмний код файлу store/modules/login/hook.ts

```
import { useInjectReducer, useInjectSaga } from 'utils/redux-injectors';
import { loginSaga } from './saga';
import { name, reducer, actions } from './slice';

export const useLoginSlice = () => {
  useInjectReducer({ key: name, reducer });
  useInjectSaga({ key: name, saga: loginSaga });

  return { actions };
};
```

A.9 Програмний код файлу store/modules/login/state.ts

```
import { LoginState } from './types';

export const initialState: LoginState = {
  loading: false,
  done: false,
  error: null,
};
```

A.10 Програмний код файлу store/modules/login/types.ts

```
/**
 * Login State
 */
export interface LoginState {
  loading: boolean;
  done: boolean;
  error?: unknown | null;
}

/**
 * Login Fetch
 */
export interface FetchTriggerPayload {
  email: string;
  password: string;
  reffererUrl: string | null;
}

export type FetchPayloadErrorType = unknown | null;
```

A.11 Програмний код файлу containers/auth/login/index.tsx

```
// npm
import React, { useEffect, useState } from 'react';
import { useDispatch, useSelector } from 'react-redux';
import { useTranslation } from 'react-i18next';
import { Redirect, useLocation } from 'react-router-dom';
```

```
// utils
import checkValidity from 'utils/check-validity';

// store
import { useLoginSlice } from 'store/modules/login/hook';
import { useProviderLoginSlice } from 'store/modules/providerLogin/hook';
import { selectLoading } from 'store/modules/login/selectors';
import { selectIsAuthenticated } from 'store/modules/app/selectors';

// data
import { initFormData } from './data';
import { LoginSEO } from './SEO';

// locales
import { translations } from 'locales/translations';

// types
import { FormFieldType } from 'types/units/form-field';
import {
  ButtonDimensionsType,
  ButtonGroupsType,
} from 'components/atoms/Button/types';
import { ProviderTypes } from 'store/modules/providerLogin/types';

// components
import GoogleLoginButton from 'components/features/GoogleLoginButton';
import FacebookLoginButton from 'components/features/FacebookLoginButton';

// styled
import {
  Wrapper,
  Header,
  HeaderLabel,
  HeaderLogo,
  Form,
  Input,
  ForgotPasswordLink,
  SignUpLink,
  LinksButtons,
  ActionsButtons,
  ProvidersButtons,
  ProvidersButtonsCaption,
  ProvidersButtonsSocials,
} from './styles';
import Button from 'components/atoms/Button';
import useQuery from '../../../hooks/useQuery';

// root
export function Login() {
  // slice hook
  const { actions: loginActions } = useLoginSlice();
  const { actions: providerLoginActions } = useProviderLoginSlice();

  // locales hook
  const { t } = useTranslation();

  // router hook
  const location = useLocation();
  const query = useQuery();

  // selectors
  const isAuthenticated = useSelector(selectIsAuthenticated);
  const loading = useSelector(selectLoading);
```

```
// local data
const [formData, setFormData] = useState(initFormData);

// form validation state
const [formIsValid, setFormIsValid] = useState(false);

// dispatch
const dispatch = useDispatch();
const onLogin = payload => dispatch(loginActions.fetchDataTrigger(payload));
const onProviderLogin = payload =>
  dispatch(providerLoginActions.fetchTrigger(payload));

// init form validity
useEffect(() => {
  let tempFormIsValid = Object.keys(formData).every(
    controlKey =>
      !formData[controlKey].invalid && formData[controlKey].touched,
  );

  setFormIsValid(tempFormIsValid);
}, [formData]);

// form inputs change handler
const inputChangedHandler = (controlName, value) => {
  const { isValid, errorText } = checkValidity({
    value,
    rules: formData[controlName].rules,
  });

  const updatedFields = {
    ...formData,
    [controlName]: {
      ...formData[controlName],
      value,
      invalid: !isValid,
      errorMessage: isValid ? '' : errorText,
      touched: true,
    },
  };

  setFormData(updatedFields);
};

// provider failure handler
const providerFailureHandler = err => {
  console.log(err);
};

// form submit handler
const submitHandler = event => {
  event.preventDefault();

  let reffererUrl: string | null = null;

  if (query.has('action') && query.get('action') === 'sign-to-product') {
    reffererUrl = sessionStorage.getItem('utm_referrer_url');
  }

  const updatedPayload = {
    email: formData.email.value,
    password: formData.password.value,
    reffererUrl: reffererUrl,
  };
```

```
    onLogin(updatedPayload);
  };

  // form formatting
  const formElementsArray: Array<FormFieldType> = [];
  Object.keys(formData).forEach(key => {
    formElementsArray.push({
      id: key,
      ...formData[key],
    });
  });

  // if isAuthenticated and redirect is absent in query, then try to replace to
from-location
  if (isAuthenticated) {
    if (query.has('redirect')) {
      const redirectUrl = query.get('redirect');
      query.delete('redirect');

      return (
        <Redirect
          to={{
            pathname: `/${redirectUrl}`,
            search: `?${query.toString()}`,
          }}
        />
      );
    } else {
      const { from } = (location.state as any) || {
        from: { pathname: '/courses' },
      };
      // history.replace(from);
      return <Redirect to={from} />;
    }
  }

  // render
  return (
    <>
      <LoginSEO />

      <Wrapper>
        <Header>
          <HeaderLabel>{t(translations.containers.login.title)}</HeaderLabel>

          <HeaderLogo isStretched={true} />
        </Header>

        <Form onSubmit={submitHandler}>
          {formElementsArray.map(formElement => (
            <Input
              {...formElement}
              key={formElement.id}
              changed={({ value }) =>
                inputChangedHandler(formElement.id, value)
              }
            />
          ))}

          <LinksButtons>
            <ForgotPasswordLink to="/password/restore">
              {t(translations.containers.login.actions.forgotPasswordLabel)}
            </ForgotPasswordLink>
```

```
            <SignUpLink
              to={{
                pathname: '/auth/sign-up',
                state: { from: location },
                search: `?${query.toString()}`,
              }}
            >
              {t(translations.containers.login.actions.signUpLabel)}
            </SignUpLink>
          </LinksButtons>

          <ActionsButtons>
            <Button
              type="submit"
              group={ButtonGroupsType.FillBlack}
              dimension={ButtonDimensionsType.Regular}
              isLoading={loading}
              disabled={!formIsValid}
            >
              {t(translations.containers.login.actions.submitFormLabel)}
            </Button>
          </ActionsButtons>

          <ProvidersButtons>
            <ProvidersButtonsCaption>
              {t(translations.containers.login.providers.caption)}
            </ProvidersButtonsCaption>

            <ProvidersButtonsSocials>
              <GoogleLoginButton
                onSuccess={accessToken =>
                  onProviderLogin({
                    provider: ProviderTypes.Google,
                    accessToken,
                  })
                }
                onFailure={providerFailureHandler}
              />
              <FacebookLoginButton
                onSuccess={accessToken =>
                  onProviderLogin({
                    provider: ProviderTypes.Facebook,
                    accessToken,
                  })
                }
                onFailure={providerFailureHandler}
              />
            </ProvidersButtonsSocials>
          </ProvidersButtons>
        </Form>
      </Wrapper>
    </>
  );
}
```

A.12 Програмний код файлу hoc/withErrorHandler.ts

```
import React, { useEffect } from 'react';
import { AxiosInstance } from 'axios';
import { Theme } from 'styles/theme/themes';
import { toast } from 'react-toastify';
import 'react-toastify/dist/ReactToastify.css';
```

```
import useHttpClient from 'hooks/httpClient';

interface IProps {
  theme: Theme;
}

const withErrorHandler =
  (
    WrappedComponent: React.FC<IProps>,
    $apiAuthClient: AxiosInstance,
    $apiResourcesClient: AxiosInstance,
    $apiCourseClient: AxiosInstance,
  ) =>
  props => {
    const [authError, clearAuthError] = useHttpClient($apiAuthClient);
    const [resourcesError, clearResourcesError] =
      useHttpClient($apiResourcesClient);

    const [courseError, clearCourseError] = useHttpClient($apiCourseClient);

    useEffect(() => {
      if (authError) {
        emitToast(authError);
        clearAuthError();
      }
    }, [authError, clearAuthError]);

    useEffect(() => {
      if (resourcesError) {
        emitToast(resourcesError);
        clearResourcesError();
      }
    }, [resourcesError, clearResourcesError]);

    useEffect(() => {
      if (courseError) {
        emitToast(courseError);
        clearCourseError();
      }
    }, [courseError, clearCourseError]);

    const emitToast = message => {
      if (message) {
        toast.error(message);
      }
    };

    return <WrappedComponent {...props} />;
  };

export default withErrorHandler;
```

A.13 Програмний код файлу utils/request.ts

```
// core
import axios, { AxiosInstance, AxiosRequestConfig } from 'axios';

const configAuth: AxiosRequestConfig = {
  baseURL: `${process.env.REACT_APP_API_AUTH_URL}/`,
  headers: {
    Accept: 'application/json',
    'Content-Type':
      'application/json;multipart/form-data;application/x-www-form-urlencoded',
```

```
  },
  withCredentials: false,
};

const $apiAuthClient: AxiosInstance = axios.create(configAuth);

const configResource: AxiosRequestConfig = {
  baseURL: `${process.env.REACT_APP_API_RESOURCES_URL}/`,
  headers: {
    Accept: 'application/json',
    'Content-Type':
      'application/json;multipart/form-data;application/x-www-form-urlencoded',
  },
  withCredentials: false,
};

const $apiResourcesClient: AxiosInstance = axios.create(configResource);

const configCourse: AxiosRequestConfig = {
  baseURL: `${process.env.REACT_APP_API_COURSE_URL}/`,
  headers: {
    Accept: 'application/json',
    'Content-Type':
      'application/json;multipart/form-data;application/x-www-form-urlencoded',
  },
  withCredentials: false,
};

const $apiCourseClient: AxiosInstance = axios.create(configCourse);

export { $apiAuthClient, $apiResourcesClient, $apiCourseClient };
```

A.14 Програмний код файлу utils/redux-injectors.ts

```
import {
  useInjectReducer as useReducer,
  useInjectSaga as useSaga,
} from 'redux-injectors';
import {
  InjectReducerParams,
  InjectSagaParams,
  RootStateKeyType,
} from './types/injector-typings';

/* Wrap redux-injectors with stricter types */

export function useInjectReducer<Key extends RootStateKeyType>(
  params: InjectReducerParams<Key>,
) {
  return useReducer(params);
}

export function useInjectSaga(params: InjectSagaParams) {
  return useSaga(params);
}
```

A.15 Програмний код файлу containers/users/profile/index.tsx

```
// core
import React, { useEffect } from 'react';
```

```
import { useDispatch, useSelector } from 'react-redux';
import { useHistory } from 'react-router-dom';

// types
import { ProfileIncludesTypes } from 'store/modules/profile/types';
import { RoleTypes } from 'types/models/role';
import {
  ButtonDimensionsType,
  ButtonGroupsType,
} from 'components/atoms/Button/types';

// store
import { useProfileSlice } from 'store/modules/profile/hook';

import {
  selectProfileData,
  selectProfileFetchLoading,
  selectSocialsList,
} from 'store/modules/profile/selectors';

// locales
import { useTranslation } from 'react-i18next';
import { translations } from 'locales/translations';

// components
import { ProfileSEO } from './SEO';

// styles
import {
  Wrapper,
  Caption,
  EditButton,
  Header,
  Inner,
  Content,
  Aside,
  Avatar,
  DefaultAvatar,
  DefaultAvatarIcon,
  Info,
  ChangePassword,
} from './styles';
import InfoTable from './InfoTable';
import EditIcon from 'components/icons/Edit';
import { ColorType, TypesType } from 'components/atoms/LoadingIndicator/types';
import LoadingIndicator from 'components/atoms/LoadingIndicator';

export function Profile() {
  // slice hook
  const { actions } = useProfileSlice();

  // locales hook
  const { t } = useTranslation();

  // react router history
  const history = useHistory();

  // selectors
  const profileData = useSelector(selectProfileData);
  const profileFetchLoading = useSelector(selectProfileFetchLoading);
  const socialsList = useSelector(selectSocialsList);

  // dispatch
  const dispatch = useDispatch();
```

```
const requestProfileData = payload =>
  dispatch(actions.fetchProfileDataTrigger(payload));

// on mount
useEffect(() => {
  const role = RoleTypes.Client;
  const profileQuery = {
    include: [ProfileIncludesTypes.Roles, ProfileIncludesTypes.Purchase].join(
      ',',
    ),
  };

  requestProfileData({ role, query: profileQuery });

  // eslint-disable-next-line react-hooks/exhaustive-deps
}, []);

// render
return (
  <>
    <ProfileSEO />

    <Wrapper>
      <Header>
        <Caption>{t(translations.containers.profile.caption)}</Caption>

        <EditButton
          type="button"
          group={ButtonGroupsType.OutlinedBlack}
          dimension={ButtonDimensionsType.Regular}
          isLoading={false}
          disabled={false}
          appendIcon={EditIcon}
          onClick={() => history.push('profile/edit')}
        >
          {t(translations.containers.profile.actions.editProfile)}
        </EditButton>
      </Header>

      <Inner>
        {profileData && (
          <Content>
            <Aside>
              {profileData.avatar && profileData.avatar.length > 0 ? (
                <Avatar src={profileData.avatar} alt="avatar" />
              ) : (
                <DefaultAvatar>
                  <DefaultAvatarIcon />
                </DefaultAvatar>
              )}
            </Aside>

            <Info>
              <InfoTable
                email={profileData.email}
                firstName={profileData.firstName}
                lastName={profileData.lastName}
                country={profileData.country}
                city={profileData.city}
                phone={profileData.phone}
                aboutMe={profileData.aboutMe}
                socials={socialsList}
              />
```

```
                <ChangePassword
                  type="button"
                  group={ButtonGroupsType.OutlinedBlack}
                  dimension={ButtonDimensionsType.Regular}
                  isLoading={false}
                  disabled={false}
                  appendIcon={EditIcon}
                  onClick={() => history.push('profile/edit#password')}
                >
                  {t(translations.containers.profile.actions.editPassword)}
                </ChangePassword>
              </Info>
            </Content>
          )}

          {profileFetchLoading && (
            <LoadingIndicator
              type={TypesType.Local}
              color={ColorType.Dark}
              size={7}
            />
          )}
        </Inner>
      </Wrapper>
    </>
  );
}
}
```

A.16 Програмний код файлу store/modules/profile/types.ts

```
import Profile from 'types/models/profile';
import { RoleTypes } from 'types/models/role';

/**
 * Profile State
 */
export interface ProfileState {
  profile: {
    data: Profile | null;
    fetch: {
      loading: boolean;
      done: boolean;
      error: unknown | null;
    };
    update: {
      loading: boolean;
      done: boolean;
      error: unknown | null;
    };
  };
  password: {
    update: {
      loading: boolean;
      done: boolean;
      error: unknown | null;
    };
  };
}

/**
 * Profile Includes
```

```
 */
export enum ProfileIncludesTypes {
  Roles = 'roles',
  Purchase = 'purchase',
}

/**
 * Payloads
 */
export interface SocialsPayload {
  facebook?: string;
  linkedin?: string;
  google?: string;
  instagram?: string;
}

export interface UserPayload {
  avatar?: string | null;
  firstName?: string;
  lastName?: string;
  phone?: string;
  aboutMe?: string;
}

/**
 * Reducers
 */

// fetch profile
export interface FetchDataTriggerPayload {
  role: RoleTypes;
  query: { [key: string]: number | string };
}

export type FetchDataFailurePayload = unknown | null;

// update profile
export type UpdateDataTriggerPayload = SocialsPayload & UserPayload;

export type UpdateDataFailurePayload = unknown | null;

// update password
export interface UpdatePasswordTriggerPayload {
  role: RoleTypes;
  data: {
    password: string;
    passwordConfirmation: string;
  };
}

export type UpdatePasswordFailurePayload = unknown | null;

/**
 * Services
 */

// get profile
export interface GetDataParams {
  role: RoleTypes;
  query: { [key: string]: number | string };
}

// put profile
export type PostDataParams = SocialsPayload & UserPayload;
```

```
// put password
export interface PostPasswordParams {
  role: RoleTypes;
  data: {
    password: string;
    passwordConfirmation: string;
  };
}
```

А.17 Програмний код файлу store/modules/profile/state.ts

```
import { ProfileState } from './types';

// The initial state of the Profile
export const initialState: ProfileState = {
  profile: {
    data: null,
    fetch: {
      loading: false,
      done: false,
      error: null,
    },
    update: {
      loading: false,
      done: false,
      error: null,
    },
  },
  password: {
    update: {
      loading: false,
      done: false,
      error: null,
    },
  },
};
```

А.18 Програмний код файлу store/modules/profile/slice.ts

```
import { PayloadAction } from '@reduxjs/toolkit';
import { createSlice } from 'utils/@reduxjs/toolkit';

import { initialState } from './state';

// types
import {
  FetchDataFailurePayload,
  FetchDataTriggerPayload,
  UpdateDataFailurePayload,
  UpdateDataTriggerPayload,
  UpdatePasswordTriggerPayload,
  UpdatePasswordFailurePayload,
} from './types';
import Profile from 'types/models/profile';

const slice = createSlice({
  name: 'profile',
  initialState,
  reducers: {
    // fetch
    fetchProfileDataTrigger(
```

```
  state,
  action: PayloadAction<FetchDataTriggerPayload>,
) {
  state.profile.fetch.loading = true;
  state.profile.fetch.done = false;
  state.profile.fetch.error = null;
},
fetchProfileDataSuccess(state, action: PayloadAction<Profile>) {
  state.profile.data = action.payload;
  state.profile.fetch.done = true;
},
fetchProfileDataFailed(
  state,
  action: PayloadAction<FetchDataFailurePayload>,
) {
  state.profile.fetch.error = action.payload;
},
fetchProfileDataFulfilled(state) {
  state.profile.fetch.loading = false;
},

// update
updateProfileDataTrigger(
  state,
  action: PayloadAction<UpdateDataTriggerPayload>,
) {
  state.profile.update.loading = true;
  state.profile.update.done = false;
  state.profile.update.error = null;
},
updateProfileDataSuccess(state, action: PayloadAction<Profile>) {
  state.profile.data = action.payload;
  state.profile.update.done = true;
},
updateProfileDataFailed(
  state,
  action: PayloadAction<UpdateDataFailurePayload>,
) {
  state.profile.update.error = action.payload;
},
updateProfileDataFulfilled(state) {
  state.profile.update.loading = false;
  state.profile.update.done = false;
},

// update password
updatePasswordTrigger(
  state,
  action: PayloadAction<UpdatePasswordTriggerPayload>,
) {
  state.password.update.loading = true;
  state.password.update.done = false;
  state.password.update.error = null;
},
updatePasswordSuccess(state) {
  state.password.update.done = true;
},
updatePasswordFailed(
  state,
  action: PayloadAction<UpdatePasswordFailurePayload>,
) {
  state.password.update.error = action.payload;
},
updatePasswordFulfilled(state) {
```

```
      state.password.update.loading = false;
      state.password.update.done = false;
    },

    // clean profile data
    cleanProfileData(state) {
      state.profile.data = null;
    },
  },
});

export const { actions, reducer, name } = slice;
```

A.19 Програмний код файлу store/modules/profile/services.ts

```
import { $apiAuthClient } from 'utils/request';

// types
import { GetDataParams, PostDataParams, PostPasswordParams } from './types';

// utils
import requestQueryFormatting from 'utils/request-query-formatting';

export function getProfileData({ role, query = {} }: GetDataParams) {
  // let url = `api/v1/${role}/me`;
  let url = `api/v1/me`;
  url = requestQueryFormatting(url, query);

  return $apiAuthClient.get(url);
}

export function postProfileData(payload: PostDataParams) {
  let url = `api/v1/me/info`;

  return $apiAuthClient.post(url, payload);
}

export function postPassword({ role, ...payload }: PostPasswordParams) {
  // let url = `api/v1/${role}/change_password`;
  let url = `api/v1/me/change_password`;

  return $apiAuthClient.post(url, payload);
}
```

A.20 Програмний код файлу store/modules/profile/selectors.ts

```
import { createSelector } from '@reduxjs/toolkit';

import { RootState } from 'types';
import { initialState } from './state';

// First select the relevant part from the state
const selectDomain = (state: RootState) => state.profile || initialState;

export const selectProfileData = createSelector(
  [selectDomain],
  profileState => {
    if (profileState.profile.data) {
      const { purchase, socials, ...data } = profileState.profile.data;

      return data;
    } else {
```

```
      return null;
    }
  },
);

export const selectPurchasesList = createSelector(
  [selectDomain],
  profileState =>
    profileState.profile.data?.purchase?.data &&
    profileState.profile.data.purchase.data.length > 0
      ? profileState.profile.data?.purchase?.data
      : [],
);

export const selectSocialsList = createSelector([selectDomain], profileState =>
  profileState.profile.data?.socials?.data &&
  profileState.profile.data?.socials?.data.length > 0
    ? profileState.profile.data?.socials?.data
    : [],
);

export const selectProfileFetchLoading = createSelector(
  [selectDomain],
  profileState => profileState.profile.fetch.loading,
);

export const selectProfileFetchDone = createSelector(
  [selectDomain],
  profileState => profileState.profile.fetch.done,
);

export const selectProfileFetchError = createSelector(
  [selectDomain],
  profileState => profileState.profile.fetch.error,
);

export const selectProfileUpdateLoading = createSelector(
  [selectDomain],
  profileState => profileState.profile.update.loading,
);

export const selectProfileUpdateDone = createSelector(
  [selectDomain],
  profileState => profileState.profile.update.done,
);

export const selectProfileUpdateError = createSelector(
  [selectDomain],
  profileState => profileState.profile.update.error,
);

export const selectPasswordUpdateLoading = createSelector(
  [selectDomain],
  profileState => profileState.password.update.loading,
);

export const selectPasswordUpdateDone = createSelector(
  [selectDomain],
  profileState => profileState.password.update.done,
);

export const selectPasswordUpdateError = createSelector(
  [selectDomain],
  profileState => profileState.password.update.error,
```

```
);
```

A.21 Програмний код файлу store/modules/profile/saga.ts

```
import { call, put, takeLatest } from 'redux-saga/effects';
import { actions as profileActions } from './slice';
import { getProfileData, postProfileData, postPassword } from './services';

/**
 * Fetch Profile request/response handler
 */
export function* fetchProfileDataSaga({
  payload,
}: ReturnType<typeof profileActions.fetchProfileDataTrigger>) {
  try {
    const { data } = yield call(getProfileData, payload);

    yield put(profileActions.fetchProfileDataSuccess(data));
  } catch (err) {
    yield put(profileActions.fetchProfileDataFailed(err));
  } finally {
    yield put(profileActions.fetchProfileDataFulfilled());
  }
}

/**
 * Update Profile request/response handler
 */
export function* updateProfileDataSaga({
  payload,
}: ReturnType<typeof profileActions.updateProfileDataTrigger>) {
  try {
    const { data } = yield call(postProfileData, payload);

    yield put(profileActions.updateProfileDataSuccess(data));
  } catch (err) {
    yield put(profileActions.updateProfileDataFailed(err));
  } finally {
    yield put(profileActions.updateProfileDataFulfilled());
  }
}

/**
 * Update Password request/response handler
 */
export function* updatePasswordDataSaga({
  payload,
}: ReturnType<typeof profileActions.updatePasswordTrigger>) {
  try {
    yield call(postPassword, payload);

    yield put(profileActions.updatePasswordSuccess());
  } catch (err) {
    yield put(profileActions.updatePasswordFailed(err));
  } finally {
    yield put(profileActions.updatePasswordFulfilled());
  }
}

/**
 * Root saga manages watcher lifecycle
 */
export function* profileSaga() {
```

```
  yield takeLatest(
    profileActions.fetchProfileDataTrigger.type,
    fetchProfileDataSaga,
  );
  yield takeLatest(
    profileActions.updateProfileDataTrigger.type,
    updateProfileDataSaga,
  );
  yield takeLatest(
    profileActions.updatePasswordTrigger.type,
    updatePasswordDataSaga,
  );
}
```

A.22 Програмний код файлу store/modules/profile/hook.ts

```
import { useInjectReducer, useInjectSaga } from 'utils/redux-injectors';
import { profileSaga } from './saga';
import { name, reducer, actions } from './slice';

export const useProfileSlice = () => {
  useInjectReducer({ key: name, reducer });
  useInjectSaga({ key: name, saga: profileSaga });

  return { actions };
};
```

A.23 Програмний код файлу types/models/profile.ts

```
import Role, { RoleTypes } from './role';
import Progress from './progress';
import Social from './social';

export default interface Profile {
  uuid: string;
  email: string;
  firstName: string;
  lastName: string;
  avatar: string | null;
  country: string;
  city: string;
  phone?: string;
  aboutMe: string;
  roleId: number;
  roleName: RoleTypes;
  purchase: {
    data: Array<Progress>;
  };
  socials: {
    data: Array<Social>;
  };
}
```

ДОДАТОК Б

(обов’язковий)

КОПІЯ НАУКОВОЇ ПУБЛІКАЦІЇ

Міністерство освіти і науки України
Хмельницький національний університет

ЗБІРНИК НАУКОВИХ ПРАЦЬ

за матеріалами XV Всеукраїнської науково-практичної конференції
«Актуальні проблеми комп'ютерних наук АПКН-2023»

17-18 листопада 2023

Хмельницький 2023

УДК 004:37:001:62

Збірник наукових праць за матеріалами XV Всеукраїнської науково-практичної конференції «Актуальні проблеми комп'ютерних наук АПКН-2023». Хмельницький. 2023. 345с.

У збірнику наукових праць подані перспективні практичні розробки аспірантів, студентів та здобувачів в області сучасних інформаційних технологій. Розглянуто актуальні проблеми комп'ютерних наук, комп'ютерної інженерії, прикладної математики й інженерії програмного забезпечення, приведено ряд робіт по впровадженню інформаційних технологій у виробництво та управління. Висвітлено перспективні розробки сучасних систем пошуку, обробки й захисту інформації, медійних та комунікаційних системи.

УДК 004:37:001:62

Матеріали конференції відтворені з авторських оригіналів. При макетуванні можливі незначні зміни компоновки контенту авторських оригіналів.

Участь у конференції та складові всіх її етапів (розгляд праць, макетування, публікація збірника наукових праць та видача сертифікатів) є безкоштовними для всіх учасників. Оргкомітет конференції висловлює подяку учасникам конференції та сподівається на подальшу співпрацю.

З питань проведення конференції та подальшого обміну інформацією звертатись на e-mail конференції: *apkt.khnu@gmail.com*

ЗМІСТ

УДК 004.4

Собко В.В.

Хмельницький національний університет

МЕТОД ПОЄДНАННЯ ТЕХНОЛОГІЙ REDUX-TOOLKIT ТА REDUX-SAGA ДЛЯ РОБОТИ З API ВЕБ-РЕСУРСІВ

Розглянуто метод поєднання технологій Redux-Toolkit та Redux-Saga для роботи з API веб-ресурсів для подальшого опрацювання даних з метою виводу їх на екран, збереження в локальне сховище пристрою, тощо. Запропонований метод є швидким, надійним, зрозумілим, дає змогу легко керувати запитами, моніторити їх стан, результат, наявність помилок та можливість їх обробляти.

The method of combining Redux-Toolkit and Redux-Saga technologies was considered for working with the API of web resources for further data processing with the aim of displaying them on the screen, saving them to the device's local storage, etc. The proposed method is fast, reliable, clear, allows you to easily manage requests, monitor their status, result, presence of errors, and the ability to process them.

З розвитком нових технологій та постійним підвищенням рівня інформатизації суспільства, проблема ефективної обробки мережевих запитів в додатках стає дедалі актуальнішою. У цьому контексті поєднання технологій Redux-Toolkit та Redux-Saga для роботи з API веб-ресурсів виявляється надзвичайно актуальним і важливим підходом.

Метою дослідження даного методу є декілька чинників:

1. Оцінка Ефективності: дослідження може спрямовуватися на оцінку ефективності даного методу. Розробники можуть аналізувати час відгуку API, кількість запитів, споживану пам'ять і інші метрики для визначення, наскільки добре цей підхід впливає на продуктивність додатку.
2. Порівняння З Іншими Підходами: дослідження може проводитися з метою порівняння методу поєднання Redux-Toolkit і Redux-Saga з іншими підходами до управління станом та роботи з API, такими як Redux-Thunk, Apollo Client (GraphQL), або Fetch API. Це допоможе визначити переваги та недоліки кожного підходу.
3. Оптимізація Логіки Додатку: дослідження може відбуватися з метою оптимізації логіки додатку, зокрема роботи з API. Розробники можуть досліджувати, як краще організувати та структурувати запити до API, обробку помилок, кешування даних і інші аспекти логіки.
4. Покращення Досвіду Користувача: основною метою може бути покращення досвіду користувача. Дослідження може допомогти виявити шляхи покращення швидкодії та надійності додатку при взаємодії з API, що відобразиться на задоволенні користувачів.

5. Підготовка Розробників: дослідження може служити для підготовки розробників до використання цього методу в практиці. Відповідні тренінги, вебінари або навчальні матеріали можуть бути розроблені на основі дослідження, щоб забезпечити зростання компетентності в цій галузі.

6. Розробка Нових Рекомендацій: на основі дослідження можуть бути розроблені рекомендації або кращі практики щодо використання Redux-Toolkit і Redux-Saga для роботи з API, які можуть бути корисними для розробників у подальших проектах.

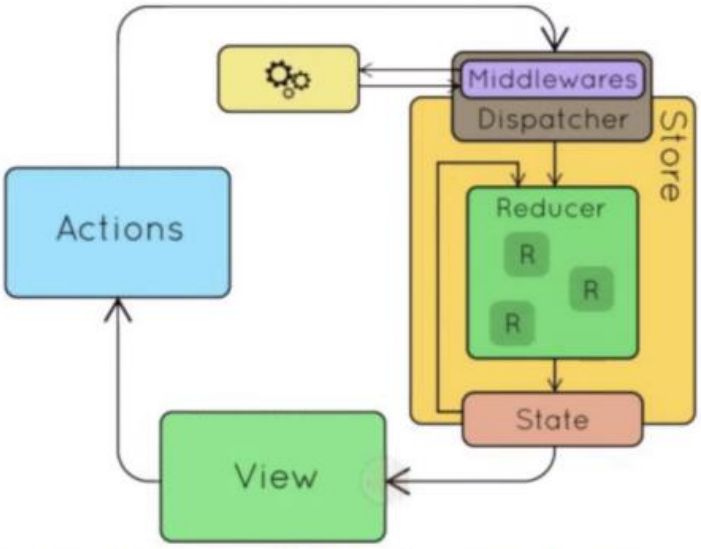


Рисунок 1 – Схема роботи досліджувального методу

Отже, запропонований метод для роботи з API веб-ресурсів забезпечує надійну архітектуру для клієнтських частин проектів. Використання такого підходу суттєво зменшує кількість помилок при виконанні мережевих запитів, обробляє їх, дає можливість отримати всю інформацію про запит як в момент виконання, так і після завершення, а також легко опрацьовувати отримані дані. Подальші дослідження спрямовані на ще більшу автоматизацію процесу, що забезпечить можливість інтеграції даної програмної архітектури в справжні веб-додатки та системи, а також зробить легшим поріг входження для розробників, які з нею будуть працювати.

Перелік посилань

1. Redux Fundamentals, Part 1: Redux Overview [Електронний ресурс] – Режим доступу до ресурсу: https://redux.js.org/tutorials/fundamentals/part-1-overview/ .
2. Redux-Saga: Beginner Tutorial [Електронний ресурс] – Режим доступу до ресурсу: https://redux-saga.js.org/docs/introduction/BeginnerTutorial/ .
3. Wieruch, Robin. Road to React: Your journey to master plain yet pragmatic React.js. – New York, 2020. – 226с.

ДОДАТОК В

(обов’язковий)

ПРЕЗЕНТАЦІЙНІ МАТЕРІАЛИ

Хмельницький Національний університет
Факультет інформаційних технологій
Кафедра інженерії програмного забезпечення

МЕТОД ПОЄДНАННЯ ТЕХНОЛОГІЙ REDUX-TOOLKIT ТА REDUX-SAGA ДЛЯ РОБОТИ З API ВЕБРЕСУРСІВ

Виконав
студент II курсу, групи ІПЗм-22-1
Собко Владислав Вадимович

Керівник:
Бедратюк Леонід Петрович
доктор фіз.-мат. наук, професор

2023 р.

ОБ'ЄКТ, ПРЕДМЕТ І МЕТА ДОСЛІДЖЕННЯ

- Об'єкт дослідження - процеси і методики розробки клієнтської частини веб-додатків, а саме управління станом та асинхронними операціями, що взаємодіють з веб-API.
- Предмет дослідження - конкретні механізми та підходи до інтеграції Redux-Toolkit і Redux-Saga як інструментів для створення ефективної архітектури управління станом веб-додатків, оптимізації процесу роботи з серверними даними через API та обробки побічних ефектів
- Мета дослідження - аналіз та розробка оптимальної методології використання цих двох технологій разом, що дозволить розробникам не тільки підвищити ефективність розробки та легкість підтримки коду, але й забезпечить більшу масштабованість та гнучкість у роботі з API вебресурсів

АКТУАЛЬНІСТЬ ТЕМИ

- Поєднання Redux-Toolkit та Redux-Saga для роботи з API вебресурсів є актуальним через їх здатність значно покращувати процес розробки та управління складними веб-додатками. Redux-Toolkit спрощує управління станом додатку, зменшуючи кількість шаблонного коду та підвищуючи продуктивність розробки. Redux-Saga ефективно управляє асинхронними операціями та побічними ефектами, роблячи код більш читабельним та легшим для тестування. Це поєднання дозволяє розробникам зосередитися на бізнес-логіці, забезпечуючи при цьому масштабованість та легкість підтримки великих додатків.

ЗАВДАННЯ ДОСЛІДЖЕННЯ

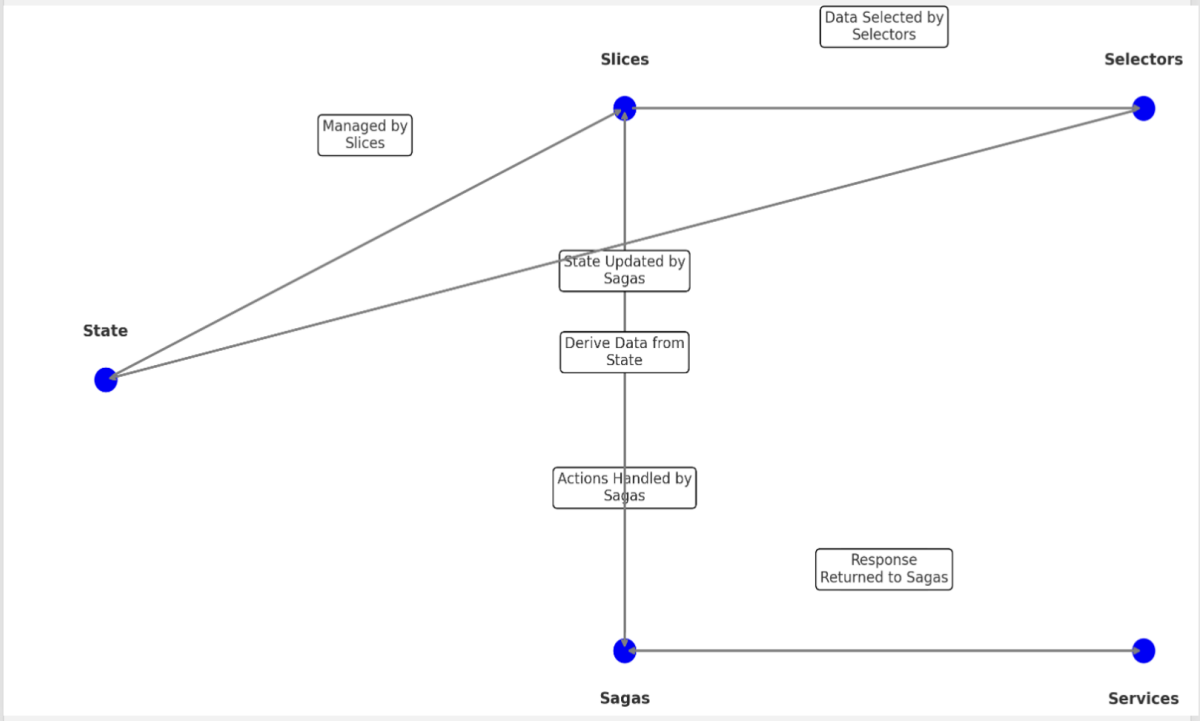
Відповідно до мети, необхідно вирішити наступні завдання дослідження:

- Проаналізувати (в загальному) сферу роботи з API вебресурсів, дослідити сучасні концієпції та підходи.
- Розробити концептуальну модель взаємодії Redux-Toolkit та Redux-Saga в контексті управління станом веб-застосунків та асинхронними операціями.
- Проаналізувати та вибрати найбільш ефективні практики використання цих технологій.
- Спроектувати архітектуру веб-застосунку, який інтегрує в себе Redux-Toolkit та Redux-Saga для роботи з API.
- Виконати програмну реалізацію розробленого методу, протестувати та оцінити його ефективність на практиці та оформити документацію щодо використання даного методу.

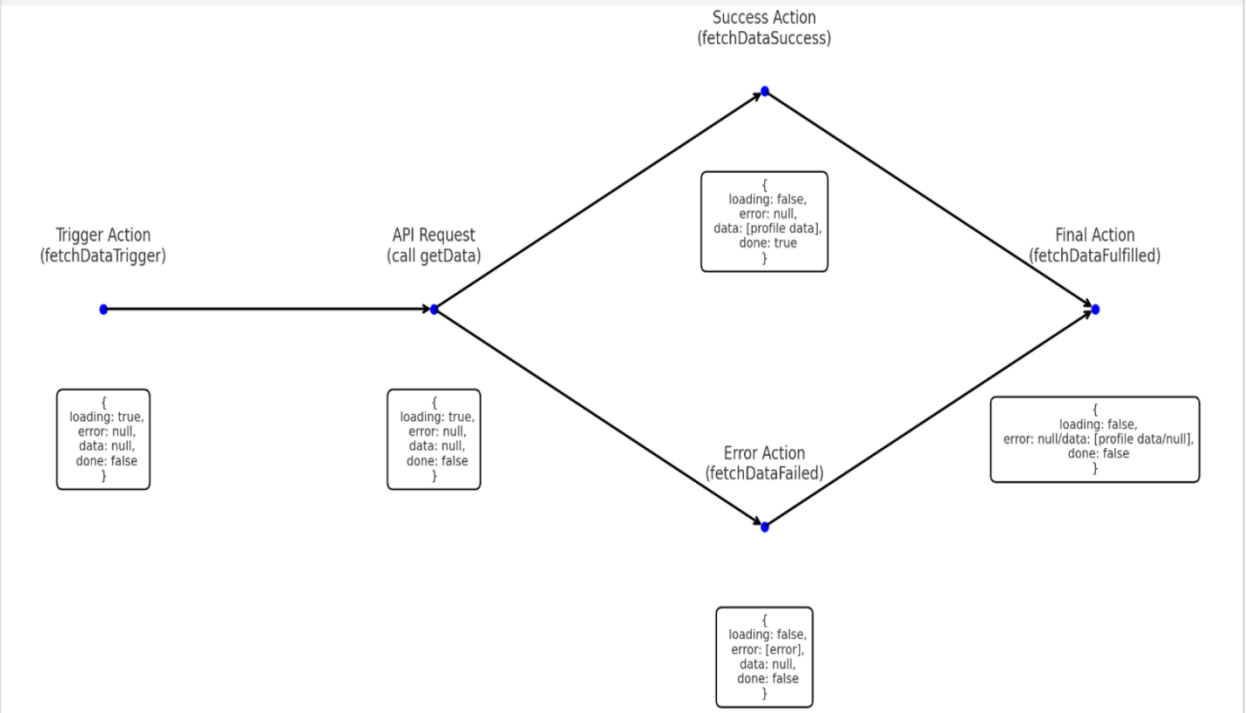
АНАЛІЗ ГАЛУЗІ ТА ВИДІЛЕННЯ ПРОБЛЕМ

- Комплексне управління станом: При роботі з API, стан додатку часто стає складним через необхідність відстежувати завантаження даних, помилки, кешування відповідей тощо.
- Управління побічними ефектами: Запити до API є асинхронними операціями, які можуть бути складними для управління.
- Розділення логіки: Великі додатки часто мають складну логіку, яка може стати заплутаною, якщо вона не правильно організована.
- Масштабування додатку: Як додатки ростуть, управління їхнім станом та побічними ефектами стає все складнішим.

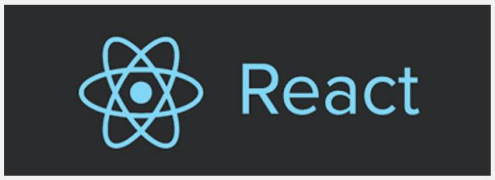
ТИПОВА СХЕМА РОБОТИ МОДУЛЯ У РОЗРОБЛЕНОМУ МЕТОДІ



ДЕТАЛІЗОВАНА СТРУКТУРА РОБОТИ МОДУЛЯ У РОЗРОБЛЕНОМУ МЕТОДІ



ВИКОРИСТАНІ ТЕХНОЛОГІЇ



ОТРИМАНІ РЕЗУЛЬТАТИ

- Покращена читабельність та підтримуваність коду: Код став чистішим, лешгим для підтримки. Це особливо корисно в великих проектах, де збереження коду чистим і зрозумілим є ключовим.
- Спрощення процесу розробки складних функціональностей: простіше та ефективніше керування станом та асинхронними операціями.
- Збільшено надійність: зведено до мінімуму кількість помилок, підвищено загальну надійність та відмовостійкість додатку.
- Прискорено швидкодію: оптимізований час модифікації даних та підвищено загальну продуктивність додатку

НАУКОВА НОВИЗНА

- Удосконалено підхід до управління станом у веб-додатках, що полягає в використанні Redux-Toolkit, який спрощує конфігурацію та зменшує обсяг шаблонного коду порівняно з традиційним Redux, що сприяє більш ефективному та гнучкому управлінню станом.
- Дістало подальшого розвитку управління побічними ефектами в JavaScript, завдяки використанню Redux-Saga, який застосовує генератори ES6 для керування асинхронними процесами, що відрізняється підвищеною читабельністю коду та полегшує його тестування.
- Вперше визначено підхід до покращення архітектури веб-додатків, який включає поєднання Redux-Toolkit та Redux-Saga, що сприяє створенню більш модульної, масштабованої та легко підтримуваної структури, розвиваючи раніше відомі методики розробки веб-технологій.

ПРАКТИЧНЕ ЗНАЧЕННЯ

Практична цінність отриманих результатів полягає в успішній розробці методу поєднання технологій Redux-Toolkit та Redux-Saga для роботи з API вебресурсів.

Завдяки розробленій архітектурі поліпшеним критеріям, які визнають якість програмного продукту (швидкодія, простота розуміння, відмовостійкість та інші), у порівнянні з класичними рішеннями, розроблений метод має шанси стати основою великих проектів у різних галузях .

НАУКОВІ ПУБЛІКАЦІЇ

- Собко В.В. Метод поєднання технологій Redux-Toolkit та Redux-Saga для роботи з API веб-ресурсів. Збірник наукових праць праць конференції АПКН-2023

ВИСНОВКИ

- Під час виконання кваліфікаційної роботи проведено аналіз та розроблено оптимальну методологію використання технологій Redux-Toolkit та Redux-Saga разом, що дозволить розробникам не тільки підвищити ефективність розробки та легкість підтримки коду, але й забезпечить більшу масштабованість та гнучкість у роботі з API вебресурсів

- Результати використання методу на практиці показують його ефективність та працездатність, а отже, його можна брати як основний архітектурний паттерн для створення комерційних проектів.

Дякую за увагу!

Завідувачу кафедри інженерії програмного
забезпечення проф. Леоніду БЕДРАТЮКУ
здобувача вищої освіти
Владислава СОБКА
факультет ІТ, 2 курс, група ПІЗм-22-1

ЗАЯВА

З правилами чинного Положення про систему забезпечення академічної доброчесності в Хмельницькому національному університеті, згідно з яким виявлення плагіату є підставою для відмови в допуску кваліфікаційної роботи до захисту і застосування заходів дисциплінарної та академічної відповідальності, ознайомлений. Про використання програмно-технічних засобів для перевірки кваліфікаційних робіт здобувачів вищої освіти на плагіат оповіщений та надаю свою згоду на обробку й збереження університетом моєї роботи в інституційному репозитарії університету.

Також надаю університету право на передачу моєї роботи для обробки та збереження в базах даних програмно-технічних засобів (Unicheck та Anti-Plagiarism) та використання роботи для виявлення плагіату в інших роботах, які перевіряються програмно-технічними засобами та користувачами, що мають доступ до цих програмно-технічних засобів, виключно в обмежених цілях для виявлення плагіату в текстах робіт.

Робота для перевірки університетом надається в друкованому та електронному варіанті. Електронна версія моєї роботи збігається (ідентична) з друкованою.

07.12.2023р.
дата

підпис

UNICHECK
by Turnitin

Ім'я користувача:
ІПЗ

ID перевірки:
1015974585

Дата перевірки:
06.12.2023 00:52:50 EET

Тип перевірки:
Doc vs Internet + Library

Дата звіту:
06.12.2023 00:53:48 EET

ID користувача:
100012953

Назва документа: КвР_Собко

Кількість сторінок: 82 Кількість слів: 13021 Кількість символів: 104688 Розмір файлу: 2.26 MB ID файлу: 1015653947

Виявлено модифікації тексту (можуть впливати на відсоток схожості)

6.42%
Схожість

Найбільша схожість: 1.84% з джерелом з Бібліотеки (ID файлу: 1009383373)

5.99% Джерела з Інтернету | 619 ..... Сторінка 84

3.14% Джерела з Бібліотеки | 134 ..... Сторінка 86

0.43% Цитат

Цитати | 3 ..... Сторінка 87

Не знайдено жодних посилань

0%
Вилучень

Немає вилучених джерел

Модифікації

Виявлено модифікації тексту. Детальна інформація доступна в онлайн-звіті.

Підозріле форматування | 18 сторінок

Tue Dec 05 23:49:22 EET 2023, Форкун Юрій Вікторович, Хмельницький національний університет, ХНУ

Anti-Plagiarism v-15.257

Максимальне співпадіння з одним документом 2.0%

Словники перевірки: en_US, ru_RU, ua_UA. **Помилок в документах: 14%**

ID: 121876 Назва: Метод поєднання технологій Redux-Toolkit та Redux-Saga для роботи з API вебресурсів Додано в БД: 2023-12-05 Автора: В.В. Собко Керівники: д-р фіз.-мат. наук, професор Л. П. Бедратюк Консультанти: Опоненти:	Документ		Сумарний збіг по Базі Даних	
	Символи	Лексеми	Символи	Лексеми
	85733	800	2988 (3%)	43 (5%)

Джерело плагіату

ID	Опис	Наявність плагіату в документі	
		Символи	Лексеми

РІШЕННЯ ЕКСПЕРНОЇ КОМІСІЇ
КАФЕДРИ ІНЖЕНЕРІЇ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ
ПРО ДОПУСК КВАЛІФІКАЦІЙНОЇ РОБОТИ ДО ЗАХИСТУ

Підтверджуємо ознайомлення з результатами звіту/звітів подібності щодо роботи, продукованими програмно-технічним засобом(ами) перевірки текстів на плагіат.
Назва: «Метод поєднання технологій Redux-Toolkit та Redux-Saga для роботи з API вебресурсів»
Автор: Собко Владислав Вадимович
Спеціальність: 121 – Інженерія програмного забезпечення
Освітня програма: Освітньо-професійна програма «Інженерія програмного забезпечення»
Науковий керівник: Бедратюк Леонід Петрович, доктор фіз.-мат. наук, професор
Після аналізу звіту подібності зроблено такий висновок:

№	Висновок	Позначка про відповідність
1	Запозичення, виявлені в роботі, є законними і не є плагіатом. Робота приймається до захисту.	**відповідає**
2	Виявлені запозичення не є плагіатом, розміщені у розділах, які не описують безпосередньо авторське дослідження, але кількість цитат перевищує обсяг, виправданий поставленою метою роботи. Робота приймається до захисту, але має бути відкоригована. Відкоригований варіант має бути поданий на кафедру за два дні до захисту, разом із заявою щодо самостійності виконання письмової роботи та ідентичності друкованої й електронної версії роботи.	
3	Виявлені запозичення не є плагіатом, але частково розміщені у розділах, які описують безпосередньо авторське дослідження, а кількість цитат перевищує обсяг, виправданий поставленою метою роботи. В зв'язку з цим мета роботи та поставлені завдання не були досягнені. Робота може бути допущена до захисту (наступного року) після того, як буде відкоригована та доопрацьована і успішно пройде повторну перевірку на академічний плагіат.	
4	Робота містить навмисні текстові спотворення, передбачувані спроби укриття запозичень або інші прояви академічного плагіату. Робота містить фабрикацію або фальсифікацію даних. Робота не допускається до захисту.	
5	Інше:	

Підтвердження:

Запозичення, виявлені у роботі, є законними і не є плагіатом, оскільки: у тексті кваліфікаційної роботи системою перевірки на плагіат Unicheck виявлено схожість з деякими документами у частині загальновживаних обов'язкових словосполучень у стандартних бланках (титулка, бланк завдання), у структурі змісту, у назвах розділів/підрозділів, у назвах публікацій переліку джерел посилання тощо;

Максимальний обсяг запозичень, визначений системою Anti-Plagiarism, складає 2.0%. Обсяг запозичень, визначений системою Unicheck виявлення збігів ідентичності/схожості, складає 6.42% і адресується до 619 джерел з інтернету і 134 джерела з бібліотеки, що, з урахуванням наведених обгрунтувань, відповідає характеру теми і свідчить на користь кваліфікаційної роботи.

Дата 6.12.2023

Завідувач кафедри ІПЗ _____ Леонід БЕДРАТЮК

Гарант освітньої програми _____ Оксана ЯШИНА

Керівник кваліфікаційної роботи _____ Леонід БЕДРАТЮК

ХМЕЛЬНИЦЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ

РЕЦЕНЗІЯ НА КВАЛІФІКАЦІЙНУ РОБОТУ

Здобувач Собко Владислав Вадимович

Тема Метод посднання технологій Redux-Toolkit та Redux-Saga для роботи з API вебресурсів

Спеціальність 121 «Інженерія програмного забезпечення»

Обсяг кваліфікаційної роботи:

Кількість листів креслень _____; кількість сторінок записки 130

1. Короткий зміст роботи та прийнятих рішень У кваліфікаційній роботі проведено детальний аналіз використання Redux-Toolkit та Redux-Saga для ефективної роботи з API вебресурсів. В ході дослідження виявлено ключові виклики та невирішені питання у цій області. Завдяки аналізу розроблено та програмно втілено методику, яка оптимізує взаємодію з API за допомогою цих технологій. Розроблений підхід сприяє підвищенню продуктивності та надійності веб-додатків, які використовують мікросервісну або клієнт-серверну архітектуру, інтегруючи інноваційні стратегії та комбінуючи вже існуючі рішення для оптимізації взаємодії з API. Це дозволило підвищити загальну продуктивність систем, збільшити пропускну спроможність та зменшити кількість помилок при роботі з прикладним програмним інтерфейсом.

2. Висновок про відповідність роботи поставленому завданню Кваліфікаційна робота освітнього ступеня «магістр» у повній мірі відповідає поставленому завданню як у теоретичній, так і в практичній її частині.

3. Характеристика виконання кожного розділу роботи, ступінь використання останніх досягнень науки і техніки і передових методів роботи У вступній частині дослідження наголошується на актуальності інтеграції Redux-Toolkit та Redux-Saga для роботи з API вебресурсів, формулюються цілі та завдання дослідження. Також описуються наукова новизна та практична значущість отриманих результатів. Перший розділ присвячений аналізу сфери використання Redux-Toolkit та Redux-Saga, а також існуючих методів взаємодії з API. Визначені методологічні підходи для рішення поставлених задач і викладена детальна постановка задачі. У другому розділі розглядаються методи і способи вирішення цих задач, включаючи вдосконалення традиційних підходів та впровадження нових стратегій. Третій розділ містить обґрунтування проектних рішень для забезпечення технічних вимог, сумісності та взаємодії компонентів системи. Четвертий розділ включає аналіз реалізації програмного засобу, його технічні та технологічні характеристики, а також емпіричне дослідження, що підтверджує ефективність запропонованого методу. Робота завершується рекомендаціями щодо застосування удосконаленого методу для підвищення ефективності взаємодії з API вебресурсів.

4. Позитивні сторони роботи Кваліфікаційна робота дає змогу глибоко зрозуміти сучасні підходи у розробці веб-додатків. Вона висвітлює ефективність інтеграції Redux-Toolkit і Redux-Saga для створення масштабованих та високоефективних API-взаємодій, підвищуючи продуктивність та якість веб-додатків. Також, ця робота демонструє практичне застосування цих технологій, надаючи розробникам цінні знання про оптимізацію процесу розробки та підвищення стабільності веб-додатків.

5. Негативні сторони роботи Кваліфікаційна робота на цю тему може зіткнутися зі складнощами у забезпеченні глибокого розуміння складних концепцій, які лежать в основі Redux-Toolkit та Redux-Saga, що може ускладнити засвоєння матеріалу для новачків. Також, фокусування на специфічному поєднанні технологій може обмежити гнучкість вибору інших підходів та інструментів для роботи з API вебресурсів.

6. Оцінка графічного оформлення та пояснювальної записки роботи Графічне оформлення виконане відповідно до теми кваліфікаційної роботи з дотриманням вимог стандартів. Пояснювальна записка відповідає вимогам стандартів до її оформлення.

7. Відгук про роботу в цілому В цілому кваліфікаційна робота заслуговує позитивної оцінки. Весь матеріал роботи структурований, чіткий та послідовний. Усі розділи роботи є послідовними та логічними, що дозволяє чітко розуміти викладений матеріал у рамках тематики кваліфікаційної роботи. Графічний матеріал дозволяє наочно побачити доцільність та ефективність рішень, які були прийняті для вирішення поставленої
задачі.

8.Інші зауваження

9. Оцінка кваліфікаційної роботи Розглянувши позитивні та негативні сторони представленої кваліфікаційної роботи, можна зробити висновок, що вона заслуговує оцінки «відмінно».

РЕЦЕНЗЕНТ (прізвище, ім'я, по батькові, посада, місце роботи)
Ніченорук Андрій Олександрович, канд. техн. наук, доцент

07.12.2023
Дата

(Підпис)