РЕФЕРАТ

Розробка односторінкового веб-застосунку (SPA) з використанням Vue.js і React: порівняльний аналіз продуктивності, користувацького досвіду та доцільності // Кваліфікаційна робота // Пасіка Дмитро Романович // Тернопільський національний технічний університет імені Івана Пулюя, факультет комп’ютерно-інформаційних систем та програмної інженерії, група СПм-62 // Тернопіль, 2023 // с. - 88, рис. - 33, додат. - 1, бібліогр. - 18.

Кваліфікаційна робота, 121 – Інженерія програмного забезпечення. – Тернопільський національний технічний університет імені Івана Пулюя, Тернопіль, 2023.

Ключові слова: ОДНОСТОРІНКОВИЙ ВЕБ-ЗАСТОСУНОК, SPA, VUE.JS, REACT, АРХІТЕКТУРА, ФРЕЙМВОРК, БІБЛІОТЕКА, ВЕБ-КОМПОНЕНТИ.

Мета роботи полягає у порівнянні React та Vue, розробці односторінкового веб застосунку (SPA) з використанням бібліотеки React та односторінкового веб-застосунку (SPA) з використанням фреймворку Vue.js, подальшого їх порівняння.

В кваліфікаційній роботі проаналізовані ключові аспекти та відмінності бібліотек та фреймворків, а саме React та Vue.js. Проведено порівняльний аналіз та досліджено переваги і недоліки даних підходів реалізації програмних застосунків типу SPA. Розроблено веб-застосунок.

Дослідження односторінкових додатків є важливим у контексті сучасної веб-розробки. Оскільки відіграє ключову роль у розумінні та впровадженні передових практик для створення ефективних, швидких та візуально привабливих веб-додатків.

ABSTRACT

Development of a Single-Page Web Application (SPA) using Vue.js and React: A Comparative Analysis of Performance, User Experience, and Appropriateness // Qualification Work // Pasika Dmytro Romanovych // Ternopil National Technical University named after Ivan Pul'uj, Faculty of Computer Information Systems and Software Engineering, group SPm-62 // Ternopil, 2023 // pp. - 88, fig. - 33, appendices - 1, bibliography - 18.

Qualification work, 121 – Software Engineering. – Ternopil National Technical University named after Ivan Pul'uj, Ternopil, 2023.

Key words: SINGLE-PAGE WEB APPLICATION, SPA, VUE.JS, REACT, ARCHITECTURE, FRAMEWORK, LIBRARY, WEB COMPONENTS.

The objective of this work is to compare React and Vue by developing a Single Page Application (SPA) using the React library and another SPA using the Vue.js framework, followed by their comparative analysis.

In this qualification work, key aspects and differences of the libraries and frameworks, specifically React and Vue.js, are analyzed. A comparative analysis is conducted, exploring the advantages and disadvantages of these approaches in implementing SPA-type software applications. A web application has been developed.

The study of single-page applications is important in the context of modern web development, as it plays a key role in understanding and implementing best practices for creating efficient, fast, and visually appealing web applications.

Зміст

ПЕРЕЛІК СКОРОЧЕНЬ....... 8

ВСТУП....... 9

1 ОГЛЯД ПРЕДМЕТНОЇ ОБЛАСТІ РОЗРОБКИ ВЕБ-ЗАСТОСУНКІВ....... 12

1.1 Веб застосунки....... 12

1.2 Застосування односторінкових веб-застосунків....... 16

1.3 Основні проблеми односторінкових веб-застосунків....... 18

1.4 Фреймворки та бібліотеки, різниця та значення в розробці веб застосунків....... 20

1.5 Проблеми використання фреймворків та бібліотек в односторінкових веб-застосунках....... 20

1.6 Апаратне та програмне забезпечення веб-систем та інструменти розробки SPA....... 21

1.7 Висновки огляду предметної області розробки веб застосунків....... 23

2 МЕТОДИ ТА АЛГОРИТМИ РОЗРОБКИ SPA З ВИКОРИСТАННЯМ REACT ТА VUE.JS АРХІТЕКТУРА ТА ПРИНЦИПИ REACT І VUE.JS....... 24

2.1 Архітектура та принципи React і Vue.js....... 24

2.2 Компоненти React і Vue.js....... 27

2.3 Стан в React і Vue.js....... 30

2.4 Менеджери стану додатку в React та Vue.js....... 34

2.5 Реактивність React та Vue.js....... 39

2.6 Методи життєвого циклу React та Vue....... 46

2.7 Тестування....... 54

2.8 Масштабованість та продуктивність....... 56

2.9 Висновки дослідження методів і алгоритмів роботи SPA з використанням Reavt та Vue.js 57

3 ОСОБЛИВОСТІ РОЗРОБКИ СИСТЕМИ ОДНОСТОРІНКОВОГО ВЕБ-ЗАСТОСУНКУ З ВИКОРИСТАННЯМ REACT І VUE.JS 58

3.1 Вибір інструментів та налаштування робочого середовища 58

3.2 Ініціалізація проекту 60

3.3 Розробка функціональності 65

3.4 Тестування 73

3.5 Висновки дослідження особливостей розробки систем односторінкових веб-додатків 77

4 ОХОРОНА ПРАЦІ ТА БЕЗПЕКА В НАДЗВИЧАЙНИХ СИТУАЦІЯХ 78

4.1 Охорона праці 78

4.2 Шкідливі та небезпечні фактори при використанні комп’ютерних систем та захист від них користувачів. 83

4.3 Оцінка стійкості системи управління підприємством. 86

ВИСНОВОК 91

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ 92

ДОДАТОК А АПРОБАЦІЯ РЕЗУЛЬТАТІВ РОБОТИ 95

ДОДАТОК Б ЛІСТИНГ КОДУ 97

ПЕРЕЛІК СКОРОЧЕНЬ

SPA – Single Page Application

MPA – Multi Page Application

PWA - Progressive Web Application

UI - User Interface

API - Application Programming Interface

DOM - Document Object Model

URL - Uniform Resource Locator

HTTP - Hypertext Transfer Protocol

HTTPS - Hypertext Transfer Protocol Secure

SEO - Search Engine Optimization

JSX - JavaScript XML

XML - Extensible Markup Language

E2E - End-to-End

TS - TypeScript

JS - JavaScript

ES6 – ECMA Script 2015

MUI - MATERIAL-UI

ВСТУП

Актуальність розробки. В сьогоденні інтернет-технології розвиваються надзвичайно швидкими темпами, всі сфер нашого життя стають на шлях цифрової трансформації, зокрема бізнес часто переходить з аналогових ресурсів до цифрових, вирішення проблем що постають в цьому світлі – веб-застосунки.

Веб-технології рушійна сила сучасного бізнесу. У сучасному світі веб-технології стали невід'ємною частиною бізнесу. Вони дозволяють компаніям будь-якого розміру та галузі досягти успіху, використовуючи такі можливості:

1. Залучення нових клієнтів. Інтернет дозволяє компаніям налагодити контакт з потенційними клієнтами в будь-якому куточку світу. Це можна зробити за допомогою веб-сайтів.
2. Продаж більшої кількості товарів та послуг. Веб-застосунки, електронна комерція та інші веб-технології дозволяють компаніям продавати свої товари та послуги більш ефективно. Вони забезпечують зручний доступ до інформації про продукти та послуги, а також можливість здійснювати покупки 24/7.
3. Покращення досвіду клієнтів. Веб-технології дозволяють компаніям надавати клієнтам підтримку 24/7. Це можна зробити за допомогою онлайн-чату, веб-форм, бази знань та інших веб-технологій.
4. Оптимізації бізнес-процесів. Веб-технології можуть автоматизувати багато завдань, що дозволяє бізнесу працювати більш ефективно. Це включає такі завдання, як управління запасами, бухгалтерський облік та управління персоналом.

Веб-технології вже активно застосовуються в бізнесі, серед успішних прикладів такого застосування можна виділити декілька основних «гігантів»:

1. Amazon - найбільший у світі інтернет-магазин, який використовує веб-технології для продажу товарів і послуг клієнтам у всьому світі. Amazon має веб-сайт, який забезпечує зручний доступ до інформації про мільйони продуктів, а також можливість здійснювати покупки 24/7. Компанія також використовує веб-технології для автоматизації своїх складських операцій та управління логістикою.
2. Facebook - соціальна мережа, яка використовує веб-технології для з'єднання людей з усього світу. Facebook має веб-сайт, який дозволяє користувачам спілкуватися один з одним, ділитися інформацією та створювати спільноти. Компанія також використовує веб-технології для таргетування реклами.
3. Google - компанія, яка використовує веб-технології для надання різноманітних послуг, включаючи пошук, електронну пошту та хмарне сховище. Google має веб-сайт, який забезпечує доступ до цих послуг. Компанія також використовує веб-технології для аналізу даних та надання персоналізованих рекомендацій.

Доцільність розробки та дослідження односторінкових застосунків є безперечно важливою в ері цифрових технологій. Це прямий шлях до: підвищення ефективності бізнесу, покращення досвіду користувачів та розширення до цього важкодоступних в реалізації розширень бізнесу.

В контексті розробки веб-застосунків важливу роль грає підбір відповідних технологій до задач котрі ставить той чи інший проєкт. Серед лідерів технологій фронтенд розробки знаходяться React та Vue.js, вони мають доволі суттєві відмінності, тому слід зважати на архітектуру і задачі проекту на етапі вибору технології.

Вибір правильних інструментів розробки може мати значний вплив на успіх веб застосунку. Правильні інструменти можуть допомогти розробникам працювати швидше та ефективніше, створювати більш інтуїтивні та зручні для користувачів застосунки та знизити витрати на розробку та підтримку.

Метою є розробка односторінкового веб застосунку та дослідження основних інструментів, методів та технічних деталей розробки веб аплікацій з використанням фреймворку Vue.js та бібліотеки React.

Об'єктами є фреймворк Vue.js, бібліотека React.

Предметом досліджень є односторінкові веб-застосунки.

Наукова новизна: вивчення архітектурних підходів Vue.js і React, включаючи способи управління даними, інтеграцію з іншими бібліотеками та підходи до створення компонентів.

Апробація результатів кваліфікаційної роботи. Результати дипломної роботи магістра апробовано на науково-технічній конференції „Інформаційні моделі, системи та технології“.

1 ОГЛЯД ПРЕДМЕТНОЇ ОБЛАСТІ РОЗРОБКИ ВЕБ-ЗАСТОСУНКІВ

1.1 Веб застосунки

Веб-застосунки — це програми, які працюють через веб-браузер користувача. Вони можуть виконувати різноманітні функції, від простих інтерактивних інтерфейсів до складних систем управління даними.

Веб-браузер - це програмне забезпечення, яке дозволяє користувачам взаємодіяти з Інтернетом. Він відображає веб-сторінки, дозволяє переходити між ними, відтворювати мультимедійний контент і виконувати скрипти. Браузери, такі як Google Chrome, Firefox, Safari та Edge, є інструментами для доступу до вебу та використання його можливостей, включаючи пошук, спілкування, навчання та розваги.

За типами веб-додатки поділяються на:

1. Односторінкові додатки (SPA) - це веб-додатки, які працюють на одній сторінці. Вони не перевантажують сторінку при зміні вмісту, а оновлюють його динамічно за допомогою JavaScript.
2. Багатосторінкові додатки (MPA) - це веб-додатки, які складаються з кількох окремих сторінок. При переході між сторінками браузер повністю завантажує нову сторінку з сервера.
3. Прогресивні веб-додатки (PWA) - це веб-додатки, які пропонують користувачам досвід, подібний до досвіду використання мобільних додатків. Вони можуть працювати без доступу до Інтернету, отримувати сповіщення та мати такий же вигляд, як мобільні додатки.

Кожен з типів додатків має свої переваги і недоліки.

Серед переваг односторінкових додатків варто виділити:

1. Швидкодію: миттєва відповіді на дії користувача завдяки асинхронному завантаженню контенту та оновленню тільки потрібних частин сторінки без перезавантаження. Що в свою чергу зменшує вплив мережі на продуктивність застосунку.

2. Кращий користувацький досвід: SPA надають відчуття неперервності, оскільки користувач може працювати з додатком, не переходячи між багатьма сторінками.

3. Зменшення навантаження на сервер: оскільки додаток завантажується лише раз, потім взаємодія з сервером відбувається через API без повного перезавантаження.

4. Ефективне управління даними: використання спеціальних бібліотек та фреймворків (наприклад, React або Vue.js) дозволяє ефективно організовувати та оновлювати дані на сторінці.

До мінусів зазвичай відносять:

1. Погану оптимізацію для пошукових систем (SEO): через асинхронне завантаження контенту та використання JavaScript, побудова ефективних SEO-стратегій для SPA може вимагати додаткових зусиль.

2. Загальний обсяг завантаження: у деяких випадках, особливо при створенні великих додатків, великий обсяг JavaScript-коду може впливати на час завантаження сторінки, зокрема на початкове завантаження.

До переваг багатосторінкових додатків належать:

1. Краща оптимізація пошукових систем (SEO): кожна окрема сторінка може бути оптимізована для пошукових систем, оскільки кожна має свій унікальний URL та метадані.

2. Швидкість першого завантаження: зазвичай, перші сторінки завантажуються швидше, оскільки не потрібно завантажувати весь код додатку разом з першим запитом.

3. Сумісність зі стандартами браузера: MPA використовують стандартну навігацію за посиланнями, що забезпечує відмінну сумісність з різними браузерами та пристроями.

4. Більш простий розвиток: розробка MPA може бути простішою, оскільки кожна сторінка може мати свою власну структуру та незалежний код.

До мінусів варто віднести:

1. Повільна відповідь на користувацькі дії: перехід між сторінками вимагає перезавантаження, що може призвести до затримок у відповіді на дії користувача.

2. Важкість у відслідковуванні користувацьких дій: для додатків, де потрібно відслідковувати користувацькі дії або стан додатку, це може становити виклик через перезавантаження сторінок.

3. Споживання більшого обсягу ресурсів: кожне перезавантаження сторінки вимагає повторного завантаження ресурсів, що може призвести до збільшення витрат по трафіку та часу завантаження.

4. Менш зручний користувацький досвід: перехід між сторінками може призвести до втрати стану або контексту, особливо при заповненні форм або перегляді складних процесів.

5. Складність у розробці масштабних додатків: при розвитку складних додатків, де потрібно багато взаємозв'язаного функціоналу, управління багатьма сторінками може бути складним.

Серед переваг прогресивних веб-додатків:

1. Офлайн доступ: PWA можуть працювати в офлайні або при обмеженому з'єднанні, завдяки кешуванню ресурсів під час першого завантаження. Це дозволяє користувачам залишатися продуктивними без Інтернет-підключення.

2. Нативні можливості: використовують функціональні можливості пристрою, такі як сповіщення, доступ до камери, геолокація, щоб надати більш природний користувацький досвід, схожий на мобільні додатки.

3. Швидкодія: PWA мають тенденцію завантажуватися швидше, забезпечуючи користувачам швидкий доступ до контенту через кешування та оптимізацію завантаження ресурсів.

4. Висока мобільність: прогресивні веб-додатки можна легко інсталювати на пристрої користувача, не потрібно завантажувати через магазин додатків, що полегшує їх поширення та використання.

5. Безпека та оновлення: інформація завантажується через HTTPS, що забезпечує захист даних користувачів. Крім того, оновлення відбуваються автоматично, що забезпечує користувачам оновлення до останньої версії без необхідності їх вручного встановлення.

З мінусів використання PWA виділяють:

1. Обмеженість функцій: хоча PWA можуть мати доступ до деяких нативних функцій пристрою, вони все ще обмежені в порівнянні з повноцінними мобільними додатками, особливо в доступі до певних апаратних можливостей.

2. Підтримка браузерами: не всі браузери та версії підтримують усі функції PWA, що може призвести до неоднакового досвіду користувачів на різних платформах.

3. Очікування від користувачів: імітації функцій мобільного додатку може призвести до очікувань, що деякі функції будуть працювати так само як у нативних додатках.

4. Обмежена підтримка на старих пристроях: деякі старі пристрої або браузери можуть не підтримувати повністю функціонал PWA через обмежену технічну можливість.

5. Збільшений обсяг пам'яті: постіне кешування та зберігання ресурсів може вплинути на обсяг використаної пам'яті на пристрої користувача.

Звичайні програми та веб-додатки мають свої унікальні переваги.

Веб-додатки, наприклад, доступні через веб-браузер і працюють на різних пристроях без необхідності встановлення. Це зручно, оскільки не потрібно скачувати чи оновлювати програми, достатньо мати зв'язок з Інтернетом. Розробка веб-додатків може бути простішою, використовуючи вже відомі веб-технології, що полегшує процес для розробників. Але для деяких завдань, особливо тих, що потребують специфічних можливостей пристрою чи більшої інтеграції з операційною системою, нативні програми можуть бути більш ефективними. Вибір

між ними залежить від конкретних потреб користувача та функціональних вимог проекту.

1.2 Застосування односторінкових веб-застосунків

Односторінкові веб-додатки - це сучасний спосіб створення веб-додатків, який забезпечує кращий досвід для користувачів. SPA робить веб-застосунки більш ефективними для бізнесу, владних структур та соціальних проектів, які потребують швидкої та плавної взаємодії з користувачем.

У сучасному світі, де швидкість і зручність – ключові показники, SPA стали популярними серед бізнесу. Вони дозволяють створювати враження безперервності для користувачів, забезпечуючи більш гладку інтеракцію та швидше завантаження контенту. Додатки такого типу чудово вирішують завдання підвищення залучення користувачів, оптимізації роботи та покращення користувацького досвіду.

Україна активно використовує сучасні технології, щоб полегшити доступ до державних послуг громадянам. Нове покоління державних структур впроваджує SPA-додатки, які спрощують процедури та забезпечують швидкий доступ до різноманітних адміністративних послуг. Наприклад, Державна міграційна служба запровадила додаток "Дія", який дозволяє громадянам отримувати реєстрацію місця проживання, отримувати закордонні паспорти та посвідчення водія. Міністерство цифрової трансформації пропонує додаток "Дія.Підтримка", що надає фінансову допомогу тим, хто постраждав від військової агресії росії. Державна Фіскальна Служба України впровадила веб-додаток "Електронний кабінет платника податків" з метою спрощення взаємодії між платниками податків та службою. Цей інтерфейс дозволяє платникам податків здійснювати електронні операції, такі як подання звітності, сплата податків, перевірка статусу своєї податкової справи тощо. Використання цього додатку дозволяє забезпечити більш

зручну та ефективну взаємодію між платниками податків та податковим органом. Міністерство охорони здоров'я України розробило веб-додаток "Helsi.me", який створено з метою полегшення доступу пацієнтів до медичних послуг. Цей додаток дозволяє пацієнтам реєструватися на прийом до лікаря онлайн, отримувати електронні рецепти та медичні довідки. Крім цього, "Helsi.me" сприяє зручності спілкування між лікарем та пацієнтом, полегшуючи доступ до медичної інформації та послуг за допомогою електронних ресурсів. Це сприяє оптимізації медичної допомоги та забезпечує більш ефективне використання ресурсів системи охорони здоров'я.

Використання SPA-додатків має свої переваги для державних структур в Україні. Вони забезпечують покращений досвід користувача, дозволяючи безперервну та плавну взаємодію, цілодобову підтримку. Все це дозволяє державі надавати громадянам сучасні та якісні послуги.

SPA стали популярним рішенням для бізнесу завдяки низці переваг, які вони пропонують. Спроектовані з урахуванням користувацької зручності, вони забезпечують безперервну взаємодію без перезавантаження сторінок. Це робить взаємодію з додатком інтуїтивно зрозумілою та комфортною. Односторінкові додатки дозволяють зберігати продуктивність користувача, оскільки не потребують постійного перезавантаження. Це особливо важливо для мобільних користувачів, які цінують швидкість та зручність у використанні. І, крім того, вони забезпечують безпечне зберігання даних, маючи можливість зберігати їх в браузері, покращуючи безпеку та швидкість доступу.

Ці додатки широко використовуються у бізнесі. Наприклад, Amazon.com використовує SPA для швидкого та зручного пошуку товарів. Facebook та Twitter використовують цю технологію для спрощення взаємодії користувачів у соціальних мережах. А Gmail забезпечує швидкий доступ до пошти та легку навігацію між повідомленнями.

Загалом, односторінкові додатки стають потужним інструментом для бізнесу, сприяючи покращенню досвіду користувача, підвищенню продуктивності та забезпеченню безпеки даних.

1.3 Основні проблеми односторінкових веб-застосунків

SPA має декілька відомих проблем.

SEO: SPA складно індексуються пошуковими системами через їх динамічний вміст, що ускладнює виявлення цих додатків в результатах пошуку.

SEO (Search Engine Optimization) - це комплекс заходів, спрямованих на підвищення видимості веб-сайту в результатах пошукових систем, таких як Google, Bing чи Yahoo. Оптимізація SEO включає в себе стратегії, які роблять сайт більш привабливим та доступним для пошукових систем, серед найпопулярніших стратегій є зокрема такі: ключові слова та контент, оптимізація метатегів, технічна оптимізація, будівництво посилань, соціальні медіа та публікації, аналітика та відстеження, локальний SEO.

Коли мова йде про односторінковий додаток, важливо зазначити, що вони часто відрізняються від традиційних веб-сайтів за своєю структурою. SPA використовують динамічне оновлення контенту без перезавантаження сторінки, що ускладнює їхню індексацію для пошукових систем. Це означає, що пошукові системи можуть мати проблеми з адекватним індексуванням контенту SPA, а це в свою чергу може призвести до поганої видимості цих сайтів у результатах пошуку. Тобто, якщо веб-сайт не індексується правильно, його може бути складно знайти під час пошуку. Це важливий аспект для власників сайтів, оскільки хороша видимість у пошукових системах може значно вплинути на кількість відвідувачів та трафік на сайті, а отже, і на успіх бізнесу чи проекту.

Тому розробники SPA часто вивчають методи оптимізації, щоб забезпечити кращу індексацію своїх додатків пошуковими системами. До основних методів оптимізації відносять:

1. Структура URL-адрес: забезпечення дружніх для SEO URL-адрес. Це допомагає пошуковим системам краще індексувати сторінки додатку.

2. Метатеги та метаописи: використання ключових слів у метатегах сторінок для підвищення їх ранжування у пошуку.

3. Контент та ключові слова: розміщення унікального, релевантного контенту з ключовими словами, які відповідають запитам користувачів.

4. Оптимізація завантаження сторінок: покращення швидкості завантаження сторінок SPA, що важливо для користувачів та ранжування в пошуку.

5. Створення карти сайту та файлу robots.txt: використання цих інструментів для полегшення індексації сторінок пошуковими системами.

6. Аналітика та відстеження: використання інструментів аналітики для відстеження ефективності стратегій SEO та їх подальшого вдосконалення.

7. Внутрішнє посилання та навігація: створення логічної та зручної системи внутрішніх посилань для полегшення навігації користувачів і пошукових систем по вашому додатку.

Наступна типова проблема SPA – складність розробки. Розробка односторінкових додатків вимагає глибшого розуміння JavaScript та веб-технологій, ніж у випадку з традиційними багатосторінковими додатками. Це означає, що розробники повинні мати більше експертизи для створення складних SPA.

Питання сумісності також важливе. SPA можуть працювати не в усіх версіях браузерів, особливо в застарілих або несумісних. Це може створити проблеми для користувачів, які використовують такі браузери, оскільки додаток може працювати неправильно або навіть не запускатися.

Щодо безпеки, через те, що SPA зберігають дані в пам'яті браузера, вони можуть стати вразливими до зовнішніх атак. Це означає, що вони можуть бути більш доступні для хакерських атак, якщо не вжиті відповідні заходи безпеки.

1.4 Фреймворки та бібліотеки, різниця та значення в розробці веб застосунків

Фреймворки та бібліотеки - це два ключові інструменти для розробки веб-додатків. Вони забезпечують розробникам готові набори функцій та інструментів, які спрощують процес створення складних веб-додатків.

Фреймворки - це великі конструкції, що надають базову основу для розробки веб-додатків. Вони включають різні функції, такі як маршрутизація, управління станом, шаблони та компоненти. Фреймворки спрощують життя розробників, дозволяючи використовувати готові інструменти та методи.

Бібліотеки - це менші набори функцій, які можна використовувати для конкретних завдань у веб-додатках. Вони включають в себе функції для візуальних ефектів, обробки даних та зв'язку з базами даних. Використання бібліотек дозволяє розробникам додавати функціональність без написання коду з нуля.

Фреймворки зазвичай стабільніші та масштабніші, проте передають більше відповідальності розробникам. Бібліотеки, незважаючи на менший обсяг функцій, можуть бути менш складними у використанні [11].

Обидва ці інструменти мають значний вплив на розробку веб-додатків, допомагаючи скоротити час розробки, покращити якість та зробити додатки більш масштабованими. Обираючи між фреймворком і бібліотекою, розробники повинні врахувати вимоги проекту та їхні власні потреби.

1.5 Проблеми використання фреймворків та бібліотек в односторінкових веб-застосунках

Фреймворки і бібліотеки – потужні інструменти у розробці високоякісних односторінкових веб-застосунків (SPA). Проте, разом з їх використанням, можуть

виникати деякі труднощі. Основні проблеми, пов'язані з фреймворками та бібліотеками в SPA:

1. Складність: фреймворки і бібліотеки можуть бути важкими для освоєння та використання, особливо для новачків.
2. Залежність: фреймворки та бібліотеки часто підпорядковані іншим інструментам, що ускладнює управління додатком.
3. Тестування: може ускладнити процес тестування SPA.
4. Збільшення обсягу пам'яті, що використовується: використання може збільшити розмір додатка, утруднюючи завантаження та продуктивність.
5. Затримки у розробці: необхідність вивчення перед використанням може сповільнити процес розробки.
6. Втрата контролю: розробники можуть втратити контроль над кодом через використання готових рішень.

Потрібно добре розуміти ці проблеми, врахувавши їх перед вибором фреймворка чи бібліотеки. Вибираючи інструмент, важливо врахувати потреби проекту. Також, слід навчитися користуватися ними та регулярно оновлювати, дотримуючись спеціальних порад для мінімізації проблем у роботі з ними.

1.6 Апаратне та програмне забезпечення веб-систем та інструменти розробки SPA

Апаратне забезпечення веб-систем - це фізичні компоненти, необхідні для роботи веб-сайтів та додатків. Апаратне забезпечення включає в себе:

1. Комп'ютери: Комп'ютери є основним компонентом будь-якої веб-системи. Вони призначені для зберігання веб-сторінок, обробки запитів користувачів і відображення веб-вмісту.

2. Сервери: Сервери - це комп'ютери, які призначені для обслуговування веб-сайтів та додатків. Вони, як правило, мають більшу потужність, ніж звичайні комп'ютери, і можуть обробляти більше запитів одночасно.

3. Мережеве обладнання: Мережеве обладнання використовується для з'єднання комп'ютерів та серверів у мережі. Він включає в себе такі пристрої, як маршрутизатори, комутатори та модеми.

4. Веб-браузери: Веб-браузери - це програми, які використовують для перегляду веб-сторінок та додатків. Вони не дозволяють користувачам взаємодіяти з веб-вмістом.

Програмне забезпечення веб-систем - це програми, які необхідні для роботи веб-сайтів та додатків. Програмне забезпечення включає в себе:

1. Веб-сервери: Веб-сервери - це програми, які обробляють запити користувачів до веб-сайтів. Вони відповідають за передачу веб-сторінок та додатків користувачам.

2. Веб-фреймворки: Веб-фреймворки - це набір інструментів та бібліотек, які використовують для розробки веб-сайтів та додатків. Вони можуть допомогти розробникам спростити процес розробки та покращити якість веб-додатків.

3. Веб-бібліотеки: Веб-бібліотеки - це набір функцій, які можна використовувати для виконання конкретних завдань у веб-додатках. Вони часто включають такі функції, як візуальні ефекти, обробка даних та зв'язок із базами даних.

Інструменти розробки односторінкових веб-застосунків включають в себе різноманітні програми, спрямовані на полегшення роботи розробників та покращення функціональності їхніх проектів. Тут є кілька примірників таких інструментів та їх функціональне значення:

1. Едитори коду: Це програми, які дозволяють розробникам створювати та редагувати код веб-додатків. Наприклад: WebStorm, Visual

Studio Code, Sublime Text, або Atom. Вони надають зручність та можливості для швидкої роботи з кодом, підсвічуючи синтаксис та автодоповнення.

2. Статичні аналізатори: Ці програми використовуються для автоматичної перевірки коду на наявність помилок або підозрілих конструкцій. Наприклад, ESLint або prettier визначають потенційні проблеми перед виконанням коду.

3. Тестувальні бібліотеки: Програми, що використовуються для автоматизованого тестування функціональності веб-додатків. Наприклад, Cypress або Jest допомагають розробникам написати тести, щоб перевірити, чи працює додаток правильно.

Ці інструменти спільно з апаратним і програмним забезпеченням веб-систем виступають як складові для створення SPA. Апаратне забезпечення надає фізичну основу, а програмне - функціональність. Використання цих інструментів робить процес розробки більш ефективним, допомагаючи створювати SPA, які не лише працюють належним чином, але й мають інтуїтивний інтерфейс для користувачів.

1.7 Висновки огляду предметної області розробки веб застосунків

Було проведено аналіз веб-застосунків та їх особливостей. Проведено огляд сфер застосування односторінкових веб-додатків, розглянуто актуальність SPA, зважаючи на досвід України і іноземних компаній в області державних, соціальних та бізнес інтересів. Проаналізовано проблематику використання односторінкових веб-додатків. Проведено аналіз допоміжних інструментів розробки – фреймворків та бібліотек, проаналізовано їх переваги та недоліки.

2 МЕТОДИ ТА АЛГОРИТМИ РОЗРОБКИ SPA З ВИКОРИСТАННЯМ REACT ТА VUE.JS АРХІТЕКТУРА ТА ПРИНЦИПИ REACT I VUE.JS

2.1 Архітектура та принципи React і Vue.js

React – це один із найбільш популярних інструментів для розробки веб-додатків, який вивів розробку на новий рівень завдяки своїм особливостям та можливостям. Він почав свій шлях від команди Facebook і вибудував велику спільноту користувачів завдяки відкритості та доступності. Головна ідея Реакту полягає у розбитті користувацького інтерфейсу на окремі компоненти, що дозволяє розробникам створювати програми швидше та ефективніше. Цей підхід сприяє перевикористанню коду й уникненню зайвого повторення завдань.

Однією з ключових переваг Реакту є використання Virtual DOM – це своєрідний "віртуальний" варіант реальної моделі об'єктів у програмі, що дозволяє оптимізувати роботу з реальним DOM, модель роботи віртуально дому можна побачити на рисунку 2.1. Це робить оновлення та відображення змін у інтерфейсі набагато ефективнішими [1].

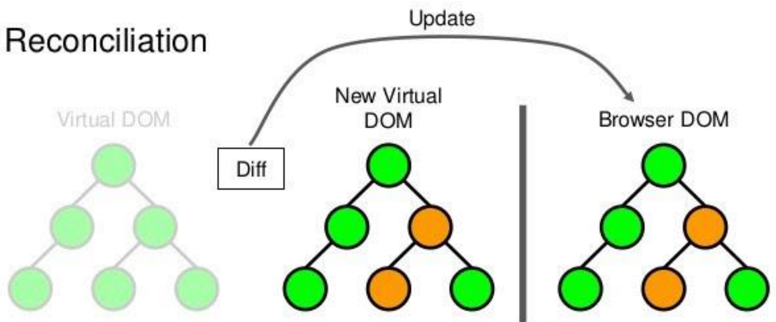

Рисунок 2.1 - Віртуальний DOM

Реакт приваблює розробників завдяки великій спільноті й різноманітним інструментам, які полегшують роботу. Його гнучкість дозволяє використовувати його в проектах будь-якої складності, від невеликих до великих, що забезпечує йому стабільну популярність серед розробників.

Компоненті підходи в React. В React існують два основні підходи до створення компонентів більш новий функційний стиль та більш старий класовий стиль.

Функціональний стиль програмування базується на використанні функцій для виконання завдань. У цьому підході функції використовуються як окремі блоки коду, які приймають вхідні дані й повертають результати. Його переваги полягають у простоті зрозуміння коду, ефективності та зручності тестування.

Класовий стиль, натомість, базується на використанні класів для створення об'єктів, які містять стан та поведінку. Його переваги включають стабільність, можливість розширення та ізоляцію стану й поведінки, що полегшує тестування та обслуговування коду.

Різниця між ними полягає в тому, як вони описують поведінку програмного забезпечення: функціональний стиль використовує функції для цього, тоді як класовий стиль працює з об'єктами, приклад на рисунку 2.2. У React обидва підходи використовуються для написання компонентів: функціональний стиль став більш популярним у зв'язку з його відповідністю принципам React, але класовий стиль може бути корисним для складних компонентів чи випадків розширення.

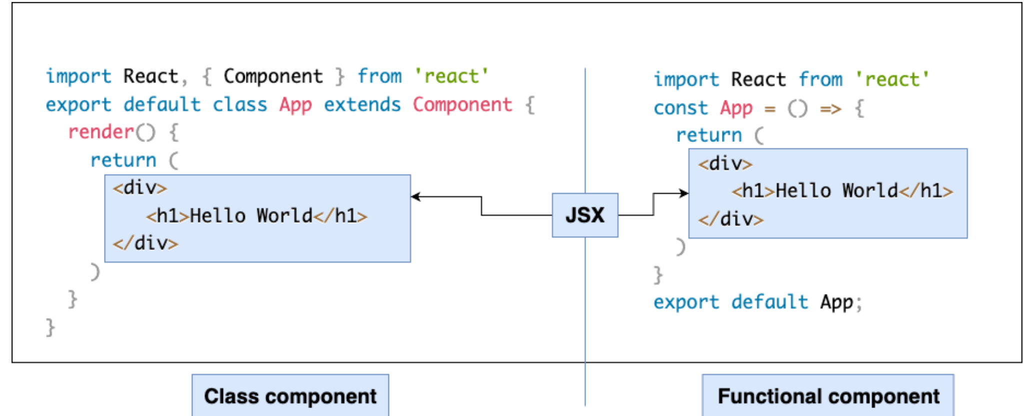

Рисунок 2.2 - Різниця класового та функційного підходів

Vue.js - це прогресивний JavaScript фреймворк для створення користувацьких інтерфейсів та односторінкових додатків. Він дозволяє розробникам створювати веб-додатки шляхом організації коду у вигляді реактивних компонентів.

Однією з його ключових особливостей є легкість вивчення завдяки простому API та доступній документації, що робить його привабливим для новачків. Vue пропонує реактивну систему, яка автоматично оновлює інтерфейс під час зміни даних, спрощуючи процес відображення інформації. Цей фреймворк побудований на компонентах, що полегшує розробку та підтримку коду, сприяючи повторному використанню компонентів у проекті. Крім того, Vue є досить модульним і може легко поєднуватися з іншими інструментами та бібліотеками, що дає розробникам гнучкість у виборі та розширенні можливостей при розробці.

У Vue, так само як і в React, існують різні підходи до написання компонентів - в Vue це Options API і Composition API. Ці підходи можна розглядати як різні підходи до структуризації та організації коду в компонентах. Початково в Vue використовувалося Options API, але з випуском Vue 3 була представлена Composition API. Options API, як відомо, є основним підходом у Vue 2. Він ґрунтується на визначенні об'єкту опцій у компоненті, де кожен параметр цього об'єкту (такий як data, methods, computed, watch тощо) відповідає певній функціональності компонента [2]. Цей підхід дозволяє легко розуміти структуру компонента, але може виникнути проблема, коли компонент стає великим і складним. Composition API, впроваджена у Vue 3, пропонує альтернативу Options API. Цей підхід дозволяє розбивати логіку компонента на логічні блоки за допомогою функцій (так званих "композицій"), які можна використовувати у різних компонентах. Composition API забезпечує більшу гнучкість, оскільки дозволяє робити перевикористання логіки, робить код більш організованим та простим для тестування. Також він спрощує управління станом та сприяє зменшенню дублювання коду.

Хоча обидва підходи можуть бути використані для розробки, Composition API відкриває нові можливості для розробників, дозволяючи створювати більш складні та швидкі у виконанні компоненти. Обидва API доступні у Vue 3, проте

Composition API виявляється більш перспективним і може бути більш популярним у майбутньому завдяки своїй гнучкості та чистоті організації коду.

2.2 Компоненти React і Vue.js

Компоненти представляють собою ключову складову як у React, так і у Vue.js. Вони є незалежними модулями коду, які можна використовувати повторно та відповідають за відображення конкретного фрагмента контенту в програмі.

Існують два основних підходи до створення компонентів у React: функціональний і класовий [1]. Функціональні компоненти виявляються більш популярними, оскільки вони краще відповідають основним принципам React, зокрема, простоті та можливості повторного використання коду.

Функціональний компонент React має синтаксис описаний на рисунку 2.3, а класовий можна побачити на рисунку 2.4.

```
function Welcome(props) {
    return <h1>Hello, {props.name}</h1>;
  }
```

Рисунок 2.3 - Ситаксис функційного компонента

```
class Welcome extends React.Component {
    render() {
      return <h1>Hello, {this.props.name}</h1>;
    }
  }
```

Рисунок 2.4 - Синтаксис класового компонента

Приклад компонента Vue в стилі Composition API та Optional API можна побачити на рисунку 2.5 та 2.6.

```
<script setup>
import { ref } from 'vue'

const message = ref('Hello World!')
</script>

<template>
  <h1>{{ message }}</h1>
</template>
```

Рисунок 2.5 - Синтаксис компонента Composition API

```
<script>
export default {
  data() {
    return {
      message: 'Hello World!'
    }
  }
}
</script>

<template>
  <h1>{{ message }}</h1>
</template>
```

Рисунок 2.6 - Синтаксис компонента Optional API

React, зазвичай, використовує JavaScript (JS), а саме ECMAScript 6 (або ES6), для написання компонентів. Однак, багато розробників також використовують TypeScript (TS) разом з React. TypeScript - це суперсет JavaScript, який додає статичну типізацію та деякі інші функції до JavaScript [10]. Використання TypeScript з React допомагає уникнути помилок під час розробки завдяки строгій перевірці типів, що сприяє покращенню безпеки та підтримує більшу стабільність коду.

Щодо Vue, цей фреймворк також підтримує JavaScript для розробки, але він також має додаткову підтримку для TypeScript. Vue.js 2 та Vue.js 3 мають певну ступінь інтеграції з TypeScript, що означає, що можна писати компоненти Vue за допомогою TypeScript, щоб отримати переваги статичної типізації та інших функцій, які надає TypeScript. Використання TypeScript у Vue дозволяє зробити код більш стабільним та легким для розуміння завдяки строгій типізації, що допомагає виявляти помилки на етапі розробки.

У світі React домінує декларативний підхід до роботи з компонентами, але можливе існування елементів, які використовують імперативний стиль. Декларативний підхід в React передбачає опис бажаного стану інтерфейсу, а не напряму керує процесом оновлення. Це означає, що ви описуєте, що хочете отримати, а React сам розбирається з тим, як це відобразити. Цей підхід зазвичай забезпечує зручність, чистоту та прозорість коду.

Щодо імперативного підходу, він орієнтований на прямне керування кожним кроком оновлення інтерфейсу. Це може включати прямі маніпуляції з DOM, які зазвичай вважаються менш чистими та менш зручними для супроводу.

У React, в основному, пропагується декларативний підхід через використання властивостей та стану для відображення компонентів, але час від часу може виникати потреба у використанні імперативних підходів, наприклад, у випадках оптимізації продуктивності або взаємодії з низькорівневими DOM операціями.

У Vue, як і в інших фреймворках, є можливість використовувати як декларативний, так і імперативний підхід до створення компонентів. Декларативний підхід у Vue полягає в описі бажаного стану або вигляду інтерфейсу, не вказуючи напряму, як це потрібно зробити. Використовуючи декларативний стиль, ви описуєте, що ви хочете побачити в інтерфейсі, та залишаєте Vue відповідальним за створення відповідного вигляду на основі цих описів. Наприклад, використання директив Vue, таких як v-for або v-if, є прикладом декларативного підходу. Ви описуєте умови, за яких потрібно відображати або рендерити елементи, а Vue здійснює це за вас. Імперативний підхід, навпаки, орієнтований на пряме керування кожним кроком в процесі оновлення інтерфейсу.

Це може включати прямі маніпуляції DOM або виклики методів Vue для зміни стану компонентів. Наприклад, пряме звертання до DOM API або безпосередні зміни стану у методах компонентів є прикладом імперативного підходу.

У Vue в основному пропагується декларативний підхід, оскільки він спрощує роботу з компонентами, робить код більш зрозумілим та легким для супроводу [2]. Однак, імперативний підхід може використовуватись у випадках, коли потрібно здійснити певний контроль над процесом оновлення інтерфейсу чи взаємодіяти з DOM операціями на низькому рівні.

2.3 Стан в React і Vue.js

У React, робота зі станом в класових і функціональних компонентах відрізняється через використання різних підходів.

У класових компонентах стан (state) визначається через конструктор класу. Він є об'єктом, який містить дані, важливі для відображення та функціонування компонента. Зміни стану в класових компонентах здійснюються через метод setState, що автоматично спричиняє перерендеринг компонентів і оновлення інтерфейсу на основі нового стану. Побачити приклад такого стану можна на рисунку 2.7.

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

    increaseCount = () => {
      this.setState({ count: this.state.count + 1 });
    };

    render() {
      return (
        <div>
          <p>Count: {this.state.count}</p>
          <button onClick={this.increaseCount}>Increase Count</button>
        </div>
      );
    }
  }
```

Рисунок 2.7 - Приклад використання стану в класовому компоненті React

У функціональних компонентах використовуються хуки (Hooks), такі як useState, для створення та оновлення стану. useState дозволяє визначити змінну стану та функцію для її оновлення. При зміні стану за допомогою функції оновлення, компонент перерендерюється, відображаючи новий стан. Побачити використання стану можна на рисунку 2.8.

Обидва підходи дозволяють компонентам React використовувати стан для зберігання та відображення даних у відповідності до їхнього стану. Однак, підходи до роботи зі станом різняться у використанні синтаксису та методів оновлення стану [4].

Щодо Vue є кілька ключових відмінностей у підході до роботи зі станом в Vue 2 і Vue 3. У Vue 2 для роботи зі станом компонентів використовувалася опція data. Однак у Vue 3 Composition API надає більшу гнучкість, дозволяючи використовувати setup для оголошення стану, методів, обробників подій та інших

логічних частин компонента. Приклади використання Vue 2 можна побачити на рисунку 2.9, а Vue 3 на рисунку 2.10.

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

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

  const increaseCount = () => {
    setCount(count + 1);
  };

  return (
    <div>
      <p>Count: {count}</p>
      <button onClick={increaseCount}>Increase Count</button>
    </div>
  );
}
```

Рис 2.8 Приклад використання стану в функційному компоненті React

```
Vue.component('button-counter', {
  template: `<button @click="incrementCounter">{{counter}}</button>`,

  data: function () {
    return {
      counter: 0
    }
  },

  methods: {
    incrementCounter: function() {
      this.counter += 1
    }
  }
})
```

Рисунок 2.9 - Приклад реалізації стану компоненти в Vue 2

```
<script setup lang="ts">
import { ref } from 'vue';
import PrimaryButton from './PrimaryButton.vue';
import DangerButton from './DangerButton.vue';

const count = ref(0);
</script>

<template>
  <div class="w-full mt-8 flex justify-center">
    <span ref="counter" class="text-3xl font-bold">{{ count }}</span>
  </div>
  <div class="w-full mt-4 flex flex-row space-x-4 justify-center">
    <primary-button ref="decrementButton" title="-" @click="count--" />
    <primary-button ref="incrementButton" title="+" @click="count++" />
  </div>
  <div class="w-full mt-4 flex flex-row space-x-4 justify-center">
    <danger-button ref="resetButton" title="Reset" @click="count = 0" />
  </div>
</template>
```

Рисунок 2.10 - Приклад реалізації стану компненти в Vue 3

У Vue 2 для управління станом компонентів використовувалася опція data, що дозволяла оголошувати змінні та дані. Методи також використовувалися для зміни цих даних [3]. Однак у Vue 3 з'явився Composition API, який дає більшу гнучкість та чіткість у роботі зі станом. Використання функції setup та ref дозволяє оголошувати та використовувати змінні у більш структурованому форматі. Реактивні змінні, оголошені через ref, автоматично оновлюють представлення, коли вони змінюються. Цей підхід дозволяє краще контролювати стан компонентів, полегшує їхнє тестування та створює більш читабельний код завдяки чіткій структурі та логіці компонентів. Vue 3 Composition API сприяє поліпшенню ефективності та зручності у роботі зі станом компонентів порівняно з попередніми версіями.

2.4 Менеджери стану додатку в React та Vue.js

Стейт менеджери відіграють ключову роль у розробці односторонніх додатків. Однією з їх основних переваг є централізоване зберігання та керування станом додатку, що позбавляє проблеми прокидання данних через компоненти, приклад можна побачити на рисунку 2.11. Це спрощує розуміння та підтримку коду, оскільки стан знаходиться в одному місці, що полегшує його масштабування, а зміни в ньому не вимагають модифікації усіх компонентів. Безперечно, важливою перевагою є покращена продуктивність за рахунок використання оптимізованих алгоритмів управління станом, особливо у додатках з великою кількістю даних. Захист стану є іншою суттєвою перевагою, оскільки стейт менеджери можуть застосовувати шифрування чи контроль доступу до нього, захищаючи дані від несанкціонованого доступу. У спільних додатках, де стан може змінюватися з різних частин, використання стейт менеджерів забезпечує синхронізацію стану між елементами додатку.

Приклади використання стейт менеджерів у SPA різноманітні: від зберігання даних користувача та прогресу гри до управління станом API. Це дозволяє створювати безпечні, масштабовані та ефективні додатки, наділені можливістю зручного зберігання та відновлення даних, що відображається в поліпшеному досвіді користувача та більшій продуктивності розробки.

Ось кілька популярних інструментів управління станом, які використовуються у односторонніх додатках (SPA):

1. Vuex: стейт-менеджер для фреймворку Vue.js. Він пропонує широкі можливості управління складним станом додатків та забезпечує централізований контроль.
2. Redux: це інструмент призначений для управління станом у React.js. Він пропонує простий та прозорий спосіб керування складними структурами стану.

3. MobX: це інструмент управління станом, який можна використовувати як для React.js, так і для Vue.js. Він відрізняється гнучкістю та простотою використання.

В контексті стейт менеджерів існує три сонoвних поняття:

1. State (стан) - внутрішній стан додатку, що може змінюватися з часом. Важливо тримати його якнайбільш прозорим та зрозумілим, оскільки це основний елемент, що керує всією логікою програми.
2. Dispatch (відправка) - механізм, який використовується для запуску або виклику дій (actions). Це спосіб зміни стану додатку. Коли потрібно оновити стан, викликають action через dispatch.
3. Action (дія) - просто об'єкт, який містить інформацію про те, який тип зміни потрібно виконати в стані додатку. Вони викликаються через dispatch і мають специфічний тип та можуть мати додаткові дані для зміни стану.

Існують загальні правила реалізації архітектури стейт менеджера в SPA, приклад можна побачити на рисунку 2.12. Збереження чистоти стану, де actions служать як єдиний спосіб внесення змін в стан. Крім того, важливо, щоб кожна дія була якнайбільш зрозумілою та однозначною, а стан додатку був узгодженим та несуперечливим після кожної дії. Це дозволяє підтримувати чистоту та стабільність вашого коду, а також полегшує відлагодження, оскільки ви знаєте точно, де і як відбуваються зміни в стані.

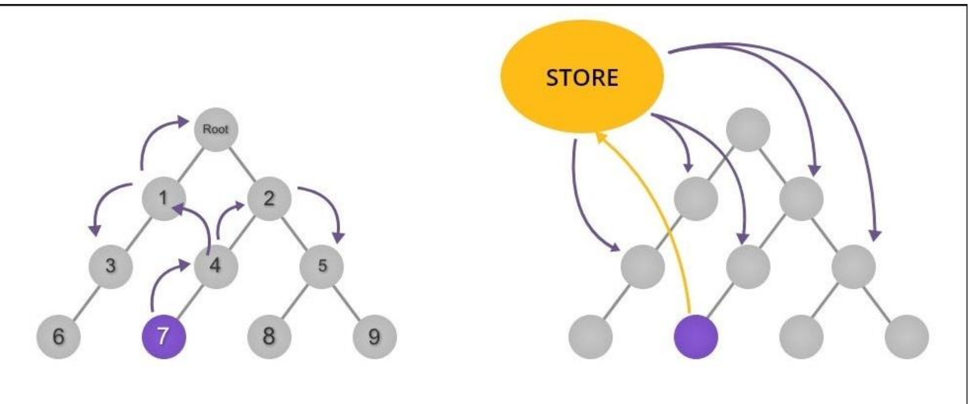

Рисунок 2.11 - Приклад прокидання данних без стейт менеджера і з ним

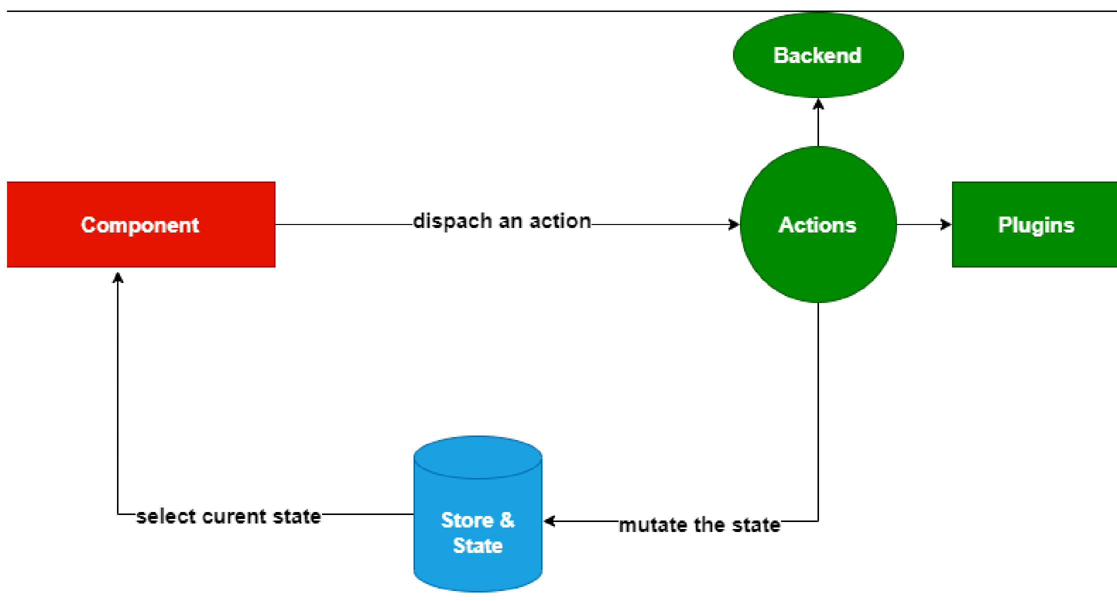

Рисунок 2.12 - Приклад архітектури стейт менеджера

Одним з найпопулярніших стейт менеджерів в React є Redux. Redux - це інструмент для керування станом у веб-додатках, спеціалізований на складних проектах з централізованим управлінням даними. Його основа - модель однорівневого стану, де усі дані додатка зберігаються в єдиному об'єкті. Це спрощує відслідковування та тестування змін у стані, сприяючи збереженню консистентності між компонентами. Redux оперує однорівневим потоком даних, в якому дії (actions) відправляються в редюсери, чисті функції, що приймають попередній стан та дію та повертають новий стан. Це допомагає уникнути плутанини та забезпечити передбачуваний потік даних. Схему роботи Redux можна побачити на рисунку 2.13.

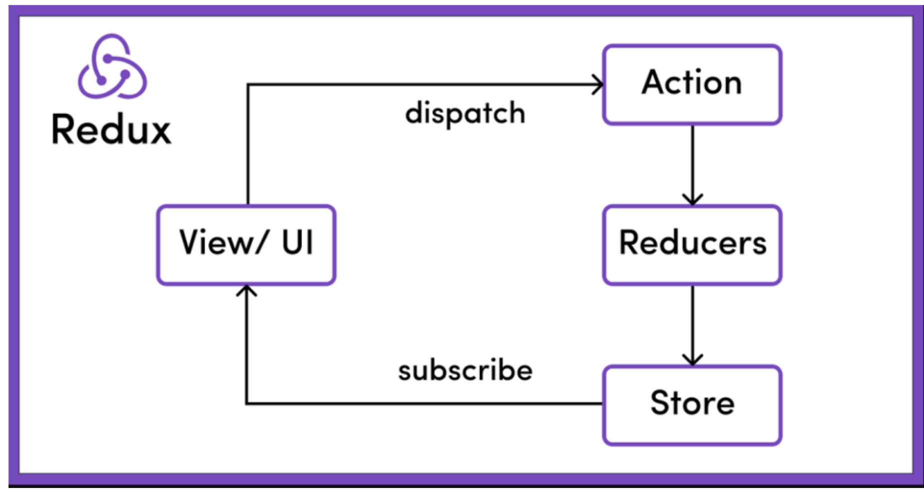

Рисунок 2.13 - Схема роботи Redux

Redux надає централізоване управління станом, що спрощує спільну роботу з даними між компонентами. Ця бібліотека забезпечує стабільну архітектуру для додатків будь-якої розмірності та допомагає уникнути плутанини в управлінні станом. Редюсери, дії та однорівневий стан - ключові елементи, що спрощують розробку та підтримку проектів у майбутньому.

Redux відмінно підходить для складних веб-додатків, які потребують централізованого управління станом та зберігання даних. Навіть у великих проектах, він допомагає зберігати консистентність між різними частинами програми та пропонує чіткі правила для управління даними.

Vuex - це інструмент для керування станом у веб-додатках, спеціально розроблений для фреймворка Vue.js. Його функціонал базується на концепціях, подібних до Redux, включаючи єдиний об'єкт стану, дії та редюсери. Vuex використовує модель однорівневого стану, де усі дані зберігаються в єдиному об'єкті, що полегшує відслідковування змін та тестування. Відмінності включають використання однорівневого потоку даних, що дозволяє уникнути плутанини та забезпечити консистентність взаємодії між компонентами додатка. В основі роботи Vuex лежить відправка дій компонентами, їх обробка за допомогою редюсерів, зміна стану додатка відповідно до цих дій та повідомлення компонентів про зміну стану, це продемонстровано на рисунку 2.14.

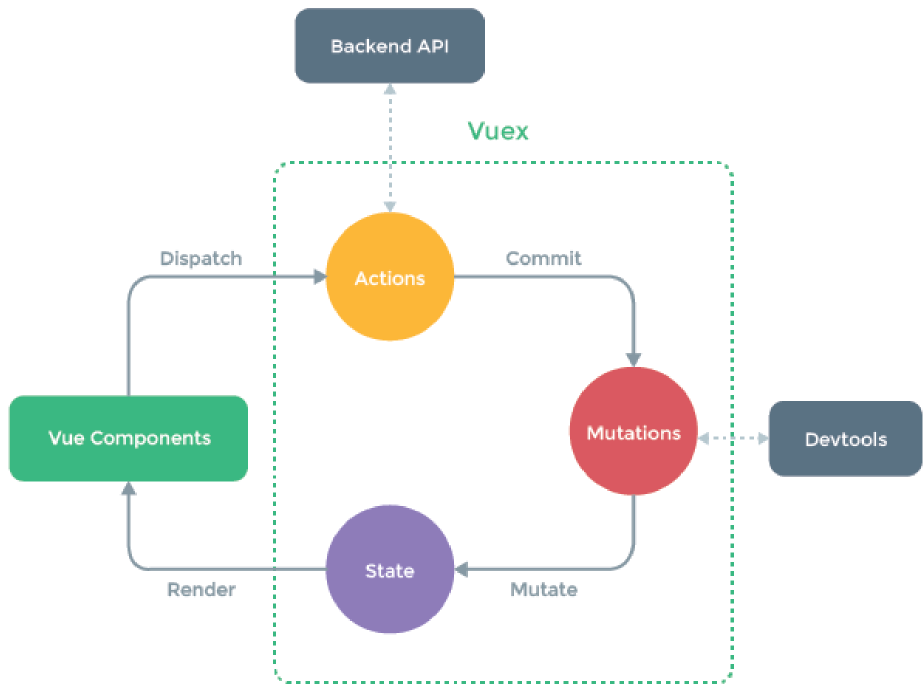

Рисунок 2.14 – Схема роботи Vuex

Vuex має ряд переваг, включаючи централізоване управління станом, прозорий та передбачуваний потік даних, стабільну та прогнозовану архітектуру, а також можливість масштабування для великих додатків. Ця бібліотека корисна для розробників, які створюють веб-додатки з централізованим управлінням станом за допомогою фреймворка Vue.js, допомагаючи уникнути плутанини та забезпечуючи консистентність взаємодії між компонентами. Vuex можна використовувати у багатьох випадках, включаючи додатки з багатим інтерфейсом користувача, які використовують багато даних, а також додатки, які потребують багаторазового використання даних чи масштабованості для великої кількості користувачів.

Вибір між Redux і Vuex залежить від конкретних потреб та переваг. Якщо мова йде про складний веб-додаток, який потребує централізованого управління станом, обидві бібліотеки можуть бути хорошим варіантом. Vuex зазвичай є більш зручним вибором для використання з фреймворком Vue.js, а Redux – типове рішення для додатків, що базуються на бібліотеці React.

2.5 Реактивність React та Vue.js

Зважаючи на те, що реактивність є ключовою концепцією у сучасному програмуванні, вона визначає здатність системи автоматично реагувати на зміни вхідних даних та відтворювати відповідний вихід. У вимірах веб-розробки та веб-додатків, реактивність покликана забезпечувати спритну відповідь системи на різноманітні зміни, що стаються в її вхідних даних. Вона виконує ключову роль у створенні інтерфейсів користувача, які завжди відображають актуальну інформацію та забезпечують високий рівень користувацького досвіду.

У парадигмі програмування реактивність виявляється через автоматичне оновлення компонентів програми при зміні даних, що лежать в основі їхньої логіки. У контексті веб-розробки це означає, що якщо дані, які відображаються на веб-сторінці, змінюються, відповідний інтерфейс автоматично відбиває ці зміни без необхідності вручному оновленні чи перезавантаженні сторінки. Наприклад, у випадку відображення списку товарів на веб-сайті, використання реактивних даних дозволяє автоматично оновлювати інформацію на сторінці, якщо дані про товари змінилися, новий товар був доданий або вже існуючий видалений. Такий механізм реактивності забезпечує користувачів актуальною інформацією без зусиль щодо оновлення сторінки.

Реактивність також приносить численні переваги у програмуванні. Вона допомагає уникнути великої кількості додаткового коду для оновлення інтерфейсу користувача та спрощує процес тестування, оскільки зміни в даних автоматично відображаються на веб-сторінці. Такий підхід сприяє покращенню користувацького досвіду та робить взаємодію з веб-додатком більш зручною і ефективною.

Реактивність у React несе в собі ідею механізму, який підтримує автоматичне оновлення компонентів в момент зміни вихідних даних, на основі яких ці компоненти будуються. Це концепція, що дозволяє створювати інтерфейси користувача, які завжди відображають актуальну та відповідну інформацію без

прямої участі розробника у процесі оновлення. Механізм реактивності в React має свої основні компоненти: віртуальний DOM та реактивні змінні.

Віртуальний DOM, іншими словами, це внутрішня модель DOM (Document Object Model), яку React використовує для представлення інтерфейсу користувача. Цей компонент визначає, які зміни потрібно внести у реальний DOM, забезпечуючи швидкі та ефективні оновлення елементів веб-сторінки, можна побачити на рисунку 2.15.

Реактивні змінні - це змінні, які автоматично оновлюються при зміні їхніх значень. Це ключовий механізм, який дозволяє відслідковувати та миттєво реагувати на зміни у вхідних даних. Коли змінюється значення такої змінної, React автоматично оновлює пов'язані з нею компоненти, гарантуючи їхню актуальність та коректне відображення даних.

Загальна схема роботи віртуального дому така:

1. Створення Компонентів: розробники створюють компоненти React, які повертають JSX, що перетворюється на JavaScript-об'єкти, описуючи структуру UI.
2. Ініціалізація Virtual DOM: при першому рендерингу React створює віртуальне дерево DOM, яке ефективно відображає UI компонентів у пам'яті і є легковаговою копією реального DOM.
3. Рендеринг у Реальний DOM: Ініційований віртуальний DOM вперше рендериться у реальний DOM, дозволяючи користувачам бачити UI на сторінці.
4. Оновлення Стану або Властивостей: Коли стан або властивості компонента змінюються, React створює нову версію віртуального DOM, яка відображає оновлений UI.
5. Порівняння Virtual DOM: React порівнює новий віртуальний DOM з попередньою версією, використовуючи алгоритм порівняння, щоб визначити, які частини реального DOM потребують оновлення.
6. Оптимізоване Оновлення Реального DOM: React оновлює тільки ті частини реального DOM, які зазнали змін, значно підвищуючи

продуктивність, оскільки взаємодія з реальним DOM є відносно дороговартісною операцією.

7. Повторення При Змінах: Цей процес повторюється кожен раз, коли стан або властивості компонента змінюються, автоматизуючи управління оновленнями DOM відповідно до змін у даних.

Важливо зазначити, що реактивність у React є ключовим інструментом, що допомагає створювати інтерфейси користувача, що поєднують в собі актуальність даних, ефективність та легкість у використанні. Цей механізм стає необхідною складовою для розробки високоякісних та надійних веб-додатків на платформі React.

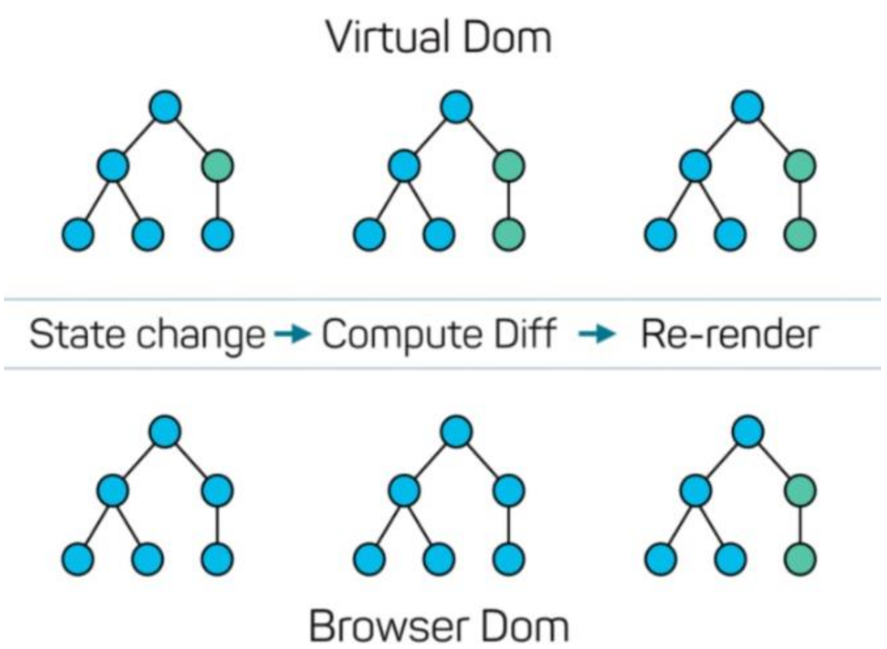

Рисунок 2.15 - Механізм роботи віртуального DOM

Реактивність у Vue.js є однією з ключових особливостей цього фреймворку. Вона дозволяє створювати інтерактивні веб-застосунки з ефективним управлінням даними.

Механізм реактивності в Vue.js, як у Composition API, так і в Options API, базується на кількох ключових принципах і технологіях. Хоча в обох API

використовуються схожі підходи для досягнення реактивності, існують відмінності у тому, як ці механізми реалізовані та використовуються.

Options API:

1. Реактивність через Getter/Setters: у Vue 2, який використовує Options API, реактивність досягається за допомогою геттерів та сеттерів JavaScript. Vue обгортає кожну властивість об'єкта data спеціальними геттерами та сеттерами, щоб відстежувати зміни.

2. Залежності та Перерахунок: коли властивості використовуються у шаблонах, вчислюваних властивостях (computed properties) або спостерігачах (watchers), Vue створює залежності. Коли змінюється властивість, Vue знає, які компоненти потрібно оновити.

Загальний принцип роботи Getter/Setters такий:

1. Ініціалізація: розробник оголошує реактивні дані у властивості data екземпляра Vue або компонента. Кожна властивість в об'єкті data є звичайною JavaScript-змінною.

2. Перетворення у Гетери/Сетери: Vue обходить кожну властивість у data та перетворює їх у гетери та сетери. Це перетворення дозволяє Vue відстежувати доступ до цих властивостей (читання та запис).

3. Відстеження Залежностей: коли компонент читає властивість з data (виклик гетера), Vue реєструє цей компонент як споживача властивості. Vue відстежує, які компоненти залежать від яких властивостей.

4. Виявлення Мутацій: коли властивість data змінюється (виклик сетера), Vue сповіщає систему реактивності про цю зміну. Це веде до запуску механізму оновлення для всіх компонентів, які залежать від цієї властивості.

5. Оновлення Компонентів: Vue ініціює перерендеринг компонентів, які залежать від змінених властивостей. Це оновлення забезпечує синхронізацію між реактивними даними та DOM.

6. Обмеження: гетери та сетери в Vue 2 не можуть відстежувати додавання або видалення властивостей після ініціалізації. Для реактивного

додавання нових властивостей після створення екземпляра використовуються методи Vue.set або vm.$set.

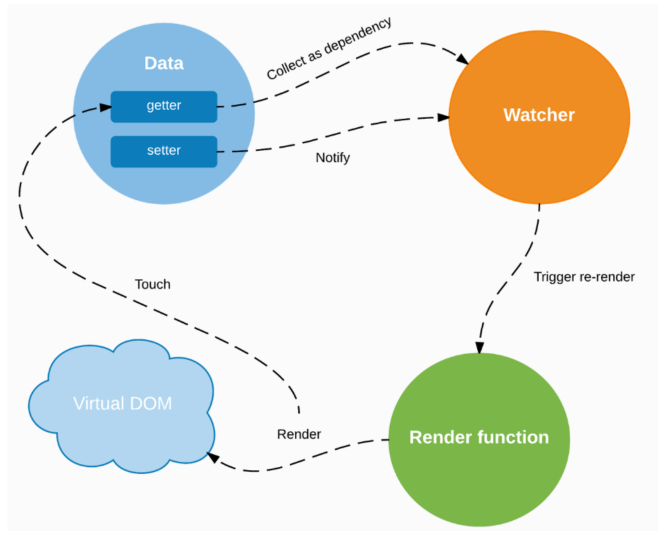

Рисунок 2.16 - Схема реактивності Options API

Composition API (Vue 3):

1. JavaScript Proxies для Реактивності: у Vue 3, який впровадив Composition API, основою реактивності є JavaScript Proxy. Proxy дозволяють Vue створювати реактивні об'єкти, які можуть відстежувати і перехоплювати зміни властивостей.

2. Функції reactive та ref: Vue 3 вводить функції reactive та ref для створення реактивних даних. reactive використовується для створення реактивних об'єктів, тоді як ref - для створення реактивних примітивних значень.

3. Краще Управління Залежностями: завдяки Proxy, Vue 3 здатний більш точно і ефективно управляти залежностями, що дозволяє уникнути непотрібних перерахунків та оновлень DOM.

JavaScript Proxy - це конструктор, що дозволяє створювати обгортку для об'єкта або функції і перехоплювати та визначати власну поведінку для основних операцій (наприклад, присвоєння властивостей, перебору і т.д.). У контексті

Composition API у Vue 3, Proxy використовується для створення реактивних референсів до даних за допомогою функцій reactive та ref.

Ось як можна схематично описати використання Proxy в Composition API для створення реактивних даних:

1. Створення Реактивного Об'єкта:
   - Виклик функції reactive() з об'єктом як аргументом.
   - Vue обгортає цей об'єкт в Proxy.
   - Proxy перехоплює звернення до властивостей і мутації об'єкта.
2. Використання Реактивних Даних:
   - Коли відбувається читання властивості з реактивного об'єкта, Proxy повідомляє Vue про залежності, які потрібно відстежувати.
   - Коли змінюється властивість реактивного об'єкта, Proxy повідомляє Vue про необхідність оновлення.
3. Відстеження Змін:
   - Vue відстежує, які компоненти залежать від цих даних.
   - Коли дані змінюються, Vue запускає перерендеринг залежних компонентів.
4. Оновлення DOM:
   - Vue використовує віртуальний DOM для визначення, які частини реального DOM потрібно оновити.
   - Тільки відповідні частини реального DOM оновлюються, що підвищує продуктивність.

Ця схема ілюструє основний принцип реактивності в Composition API Vue 3, де Proxy діє як основа для реактивної системи, що дозволяє фреймворку бути більш ефективним і гнучким.

Незалежно від того, чи використовується Options API чи Composition API, основні принципи реактивності у Vue залишаються однаковими:

1. Відстежування Змін: Vue відстежує, коли реактивні дані використовуються та змінюються, для забезпечення актуального стану DOM.
2. Автоматичне Оновлення DOM: коли реактивні дані змінюються, Vue автоматично оновлює DOM без необхідності ручного втручання з боку

Обидва API пропонують потужні та гнучкі способи для реалізації реактивності у Vue-застосунках, дозволяюч розробника.

Порівняння React і Vue в контексті реактивності та продуктивності є предметом інтенсивних дискусій серед розробників. Обидва фреймворки мають свої особливості, які впливають на швидкість і ефективність, але вибір між ними часто залежить від конкретного випадку використання та особистих переваг.

React: Virtual DOM та Компонентний підхід. React використовує Virtual DOM, який є ефективним способом управління змінами в DOM. Ключова перевага полягає в тому, що React порівнює попередній стан і новий стан віртуального DOM, щоб визначити найменші необхідні зміни в реальному DOM. Цей підхід зменшує кількість взаємодій з DOM, що є дороговартісною операцією з точки зору продуктивності. Однак, процес порівняння та оновлення може бути ресурсоємним, особливо для великих та складних UI.

Vue: Реактивність через Proxies та гетери/сетери. Vue використовує гетери/сетери в Vue 2 та Proxies в Vue 3 для реактивності. У Vue 2, гетери/сетери відстежують зміни в реактивних даних та автоматично спричиняють оновлення DOM. Vue 3 покращує цей процес, використовуючи JavaScript Proxies, які дозволяють більш гнучке управління реактивністю, включаючи відстеження динамічних властивостей. Це робить Vue ефективнішим у певних сценаріях, особливо коли мова йде про динамічне відстеження змін у даних.

Порівняння продуктивності. Загалом, React та Vue.js оптимізовані для високої продуктивності та швидких рендерів.

Різниця в продуктивності часто зводиться до специфіки застосування та архітектури конкретного додатку:

1. Великі та Складні Інтерфейси: React може мати переваги у великих додатках зі складними інтерфейсами завдяки ефективному управлінню станом та поділу на компоненти.
2. Динамічність та Гнучкість: Vue може бути більш ефективним у додатках, де часто відбуваються динамічні зміни даних, завдяки своєму гнучкому реактивному системі.

Вибір між React та Vue залишається залежним від конкретних вимог проекту, архітектурних рішень та особистих переваг розробників. Обидва фреймворки надають потужні інструменти для реалізації реактивних веб-додатків з високою продуктивністю. Важливо також зазначити, що продуктивність може бути оптимізована на багатьох рівнях розробки, і вибір фреймворку - лише один з елементів у цьому процесі.

2.6 Методи життєвого циклу React та Vue

Методи життєвого циклу — це основоположна концепція в розробці інтерфейсів користувача, особливо у веб-фреймворках і бібліотеках, таких як React, Vue, Angular та інші. Ці методи надають можливість керувати поведінкою компонентів на різних етапах їх життя: від ініціалізації до знищення. Вони відіграють ключову роль у керуванні даними, рендерингу, обробці подій та оптимізації продуктивності застосунків.

Значення методів життєвого циклу:

3. Контроль за станом компонентів: методи життєвого циклу дозволяють розробникам виконувати код у певні моменти життя компонента. Наприклад, ініціалізація стану або встановлення початкових даних з API можуть бути виконані при створенні компонента.
4. Взаємодія з DOM: деякі методи життєвого циклу викликаються до або після того, як компонент відображений у DOM. Це дає можливість

маніпулювати DOM або виконувати дії, які залежать від розмірів чи положення елементів.

5. Оптимізація продуктивності: методи життєвого циклу дозволяють запобігти непотрібним оновленням та перерендерингам, що важливо для оптимізації продуктивності додатків, особливо при роботі з великими обсягами даних або складними інтерфейсами.

6. Керування побічними ефектами: чистка та інші дії, які мають виконуватися після знищення компонента або перед його оновленням, можуть бути реалізовані через методи життєвого циклу.

7. Реагування на зміни: ці методи можуть виявляти зміни у властивостях або стані та відповідати на них, наприклад, оновлюючи відображення або викликаючи запити до сервера.

У процесі розробки веб-додатків, методи життєвого циклу відіграють надзвичайно важливу роль, дозволяючи розробникам втілювати різноманітні функціональності та оптимізувати поведінку додатку. Наприклад, однією з ключових задач при старті компонента є завантаження даних з сервера. Це забезпечує, що користувачі мають доступ до актуальної інформації, як тільки відкривають сторінку або компонент. Крім того, важливим аспектом є управління підписками на події або сервіси. Методи життєвого циклу дозволяють не тільки підписуватися на необхідні події, але й відписуватися від них у момент, коли компонент більше не використовується. Це допомагає уникнути витоку пам'яті та забезпечити кращу продуктивність додатку. Важливою складовою є також застосування анімацій чи транзицій. Після того, як компонент було відрендерено, можна активувати анімації чи транзиції для покращення візуального враження та забезпечення більш плавного користувацького інтерфейсу. Це додає динамічності та привабливості веб-сторінкам, роблячи їх більш захоплюючими для користувачів.

Методи життєвого циклу грають ключову роль у збереженні стану компонентів при перемиканні між різними частинами додатку. Це особливо актуально в односторінкових застосунках де користувач може переходити між

різними секціями без перезавантаження сторінки. Завдяки методам життєвого циклу, стан компонентів може бути збережено або відновлено, забезпечуючи плавний та зручний перехід для користувача.

Методи життєвого циклу в класових компонентах React є ключовими для розуміння того, як компонент взаємодіє з DOM та іншими частинами додатку. Ці методи надають розробникам контроль над різними стадіями життя компонента: від його ініціалізації та монтажу в DOM до його оновлення та знищення. Схематично основні методи зображені на рисунку 2.17. Основні методи життєвого циклу в класових компонентах React:

Ініціалізація (Initialization):

1. constructor(props)

- Це перший метод, який викликається в процесі створення компонента.
- Використовується для ініціалізації стану та прив'язки методів (наприклад, обробників подій) до екземпляра компонента.
- У конструкторі необхідно викликати super(props) перед будь-яким іншим виразом, щоб ініціалізувати this.props.

Монтаж (Mounting): ці методи використовуються при вставці компонента в DOM:

1. static getDerivedStateFromProps(props, state)

- Викликається перед кожним рендерингом компонента, як під час першого монтажу, так і під час подальших оновлень.
- Використовується для оновлення стану на основі змін у властивостях (props).
- Повинен повертати об'єкт для оновлення стану або null, щоб нічого не змінювати.

2. render()

- Обов'язковий метод для кожного класового компонента.

- Повертає React-елементи, які представляють, що має бути відображено в DOM.
- Не має виконувати побічні ефекти, такі як запити до сервера або взаємодії з браузерним API.

3. componentDidMount()
   - Викликається один раз після першого рендеринга компонента в DOM.
   - Ідеальне місце для запитів до сервера, підписки або ініціалізації бібліотек, які взаємодіють з DOM.

Оновлення (Updating): ці методи використовуються при оновленні компонента через зміни в props або state.

4. static getDerivedStateFromProps(props, state)
   - Викликається також при оновленні компонента.
   - Може бути використаний для оновлення стану відповідно до нових властивостей.
5. shouldComponentUpdate(nextProps, nextState)
   - Викликається перед кожним оновленням компонента.
   - Повертає true (за замовчуванням) або false, щоб вказати, чи повинен компонент оновлюватися.
   - Використовується для оптимізації продуктивності, запобігаючи непотрібним оновленням.
6. render()
   - Викликається знову, щоб відобразити нові дані.
7. getSnapshotBeforeUpdate(prevProps, prevState)
   - Викликається перед оновленням DOM відповідно до нових відображуваних даних.
   - Використовується для збору інформації з DOM (наприклад, положення прокрутки), яку можна використовувати після оновлення.
8. componentDidUpdate(prevProps, prevState, snapshot)

- Викликається після оновлення компонента.
- Може бути використаний для виконання побічних ефектів, які залежать від зміненого стану чи властивостей.

Знищення (Unmounting)

1. componentWillUnmount()
   - Викликається перед тим, як компонент буде видалений з DOM.
   - Використовується для виконання прибирання: відписка від подій, зупинка таймерів, скасування запитів тощо.

Рідко використовувані методи:

1. componentDidCatch(error, info)
   - Використовується в компонентах, що слугують "ловцями помилок" для їх перехоплення у дочірніх компонентах.

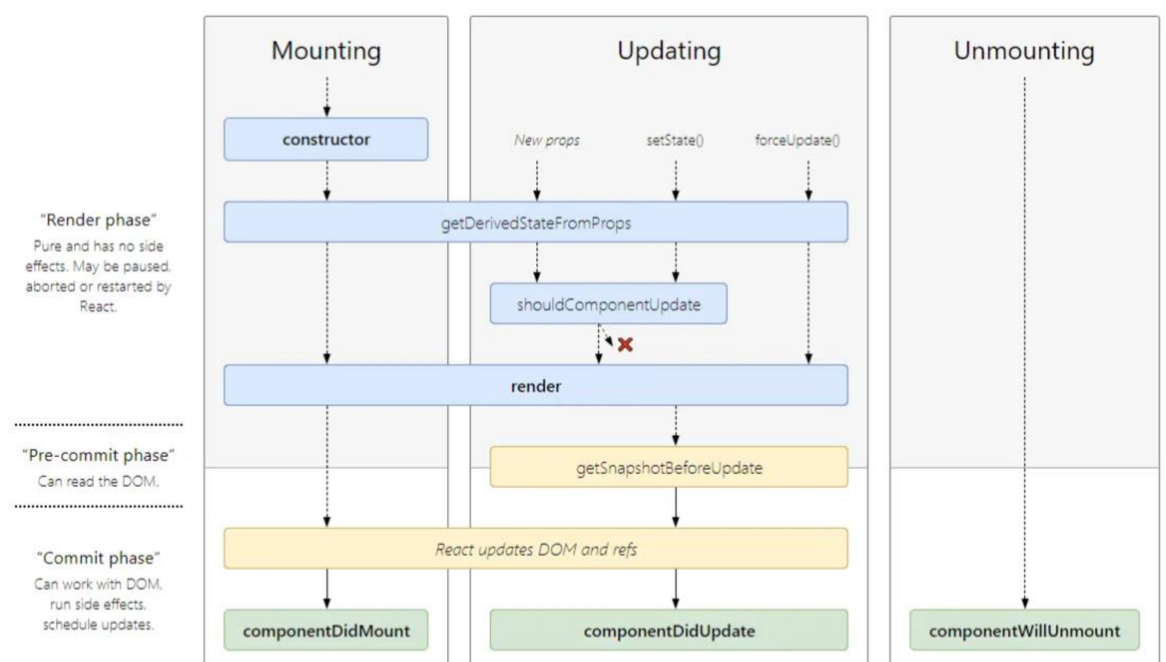

Рисунок 2.17 – Методи життєвого циклу React

Ці методи життєвого циклу надають розробникам React гнучкі інструменти для керування компонентами на всіх етапах їх життєвого циклу, від ініціалізації до знищення, дозволяючи створювати ефективні та реактивні веб-інтерфейси.

Функціональні компоненти в React, особливо після введення хуків у версії 16.8, надають потужні можливості для управління життєвим циклом компонента. Відмінність між класовими та функціональними компонентами полягає в тому, що функціональні компоненти не мають методів життєвого циклу в традиційному розумінні [6]. Замість цього, вони використовують хуки для керування побічними ефектами, станом та іншими реактивними функціями.

Хуки у функціональних компонентах:

1. useState
   - Хук useState дозволяє додавати стан до функціональних компонентів. Він повертає пару: поточне значення стану та функцію для його оновлення.
2. useEffect
   - Хук useEffect використовується для виконання побічних ефектів у функціональних компонентах. Це можуть бути операції, які впливають на DOM, запити до сервера, підписки або інші нечисті операції.
   - useEffect може імітувати поведінку componentDidMount, componentDidUpdate і componentWillUnmount залежно від того, як він використовується.
3. useContext
   - Хук useContext дозволяє компонентам підписатися на контекст React без необхідності використовувати вкладений компонент Consumer.
4. useReducer
   - Хук useReducer використовується для управління внутрішнім станом компонента з використанням редуктора. Це альтернатива useState, особливо корисна для управління складним станом.
5. useMemo і useCallback

- Хуки useMemo та useCallback використовуються для оптимізації продуктивності. useMemo зберігає результат обчислення, а useCallback - функцію обробника.

6. Custom Hooks

- Розробники можуть створювати власні хуки, щоб відокремити логіку компонентів та перевикористовувати її в інших компонентах.

Різниця між функціональними та класовими компонентами.

У функціональних компонентах усі можливості життєвого циклу реалізуються через хуки, тоді як у класових компонентах вони реалізуються через методи класу. Хуки надають більш гнучкий та елегантний спосіб управління станом та побічними ефектами, зменшуючи кількість шаблонного коду та спрощуючи рефакторинг та тестування. Функціональні компоненти з хуками зробили React більш декларативним та функціональним, надаючи розробникам потужні інструменти для створення сучасних веб-інтерфейсів.

Vue.js, як сучасний фреймворк для розробки інтерфейсів, використовує різні підходи до життєвих циклів компонентів у своїх основних API: Options API та Composition API. Кожен з цих підходів має свої особливості та способи управління життєвими циклами компонентів.

Options API.

Options API — це традиційний підхід Vue до життєвих циклів, де різні аспекти компоненту (дані, методи, вчислені властивості тощо) організовані за опціями:

1. beforeCreate: викликається відразу після ініціалізації компонента, до ініціалізації даних та подій.
2. created: викликається після створення компонента, коли дані та методи ініціалізовані, але перед його монтуванням у DOM.
3. beforeMount: викликається перед монтуванням компонента у DOM.

4. mounted: викликається після монтування компонента у DOM. Це місце для дій, які потребують доступу до DOM.

5. beforeUpdate: викликається при оновленні даних, перед віртуальним рендерингом та перерендерингом DOM.

6. updated: викликається після того, як компонент та його дочірні компоненти перерендерені.

7. beforeDestroy: викликається перед знищенням компонента, корисно для звільнення ресурсів та відписки від подій.

8. destroyed: викликається після знищення компонента.

Composition API.

Composition API, представлений у Vue 3, надає більш гнучкий спосіб управління життєвим циклом компонентів через композиційні функції:

1. setup(): центральна функція Composition API, яка викликається перед будь-якими іншими життєвими циклами компонента. Вона використовується для створення реактивних даних, вчислених властивостей та функцій.

2. onBeforeMount, onMounted, onBeforeUpdate, onUpdated, onBeforeUnmount, onUnmounted: це аналоги методів життєвого циклу з Options API, які можна використовувати всередині setup() для керування відповідними аспектами життєвого циклу компонента.

3. watchEffect і watch: функції для спостереження за реактивними залежностями та реагування на їх зміни. Вони забезпечують функціональність схожу на watchers в Options API.

Composition API відкриває можливості для більш гнучкої організації коду,

дозволяючи групувати пов'язану функціональність за логічними блоками, замість розділення її за опціями, як в Options API. Це особливо корисно в складних компонентах та дозволяє легше перевикористовувати та тестувати код.

І React, і Vue пропонують ефективні способи управління життєвими циклами компонентів, і вибір між ними часто залежить від конкретних потреб

проекту та досвіду команди розробників. Як правило, вони обоє здатні забезпечити високу продуктивність для більшості веб-додатків, коли вони правильно використовуються та оптимізовані.

2.7 Тестування

Тестування в односторінкових застосунках є критично важливим для забезпечення якості, надійності та користувацького досвіду. SPA пропонують динамічний інтерфейс, змінюючи контент без перезавантаження сторінки, що створює унікальні виклики для тестування. Ось основні типи тестування:

1. Юніт-тестування (Unit Testing): юніт-тестування зосереджено на найменших тестових одиницях, як-то функціях або компонентах. Це включає перевірку індивідуальної функціональності для виявлення помилок у логіці або реалізації. Юніт-тести важливі для раннього виявлення помилок та спрощення процесу рефакторингу.
2. Інтеграційне тестування (Integration Testing): інтеграційні тести перевіряють взаємодію між різними частинами застосунку, наприклад, між компонентами UI та менеджерами стану або API. Це допомагає забезпечити, що всі частини застосунку працюють разом, як передбачено.
3. Кінцеве тестування (End-to-End Testing, E2E): E2E-тестування імітує реальні користувацькі сценарії та перевіряє застосунок у його повному виконанні. Це включає тестування навігації, взаємодії з користувачем, асинхронних запитів та інтеграції з зовнішніми сервісами.

Інструменти, такі як Jest для юніт-тестування або Cypress для E2E тестування, є популярними серед розробників SPA.

Тестування в React та Vue.js не має суттєвих відмінностей, саме тому можна розглядати цей аспект як спільний.

Юніт-тести з використанням Jest — це потужний підхід для перевірки індивідуальних частин коду, таких як функції або компоненти, в ізоляції від інших частин програми. Jest — це тестовий фреймворк JavaScript, який надає швидке встановлення та простоту використання, роблячи його популярним вибором серед розробників. При юніт-тестуванні з Jest кожен тест фокусується на перевірці конкретної функціональності. Тести описують очікувану поведінку функцій або компонентів і перевіряють, чи відповідає реальна поведінка цим очікуванням. Це включає в себе виклик функцій з певними аргументами та перевірку виведених значень. Jest автоматично відстежує та виконує файли тестів, надаючи детальний звіт про виконані тести, їхній статус (пройшли або не пройшли) та помилки, якщо такі є. Він також підтримує "mocking" та "spying", що дозволяє імітувати взаємодії з зовнішніми модулями та відстежувати, як ці модулі використовуються.

Завдяки своїй простоті та гнучкості, Jest є ідеальним рішенням для написання та виконання юніт-тестів у сучасних JavaScript-проектах.

End-to-End (E2E) тестування з використанням Cypress є процесом, який імітує поведінку кінцевого користувача для перевірки загальної функціональності веб-застосунків. Cypress — це потужний фреймворк для автоматизованого тестування, який дозволяє розробникам легко писати, запускати та відлагоджувати тести у реальному браузері. E2E тести з Cypress включають сценарії, які автоматично виконують дії у браузері, такі як введення даних у форми, натискання кнопок, перехід між сторінками та перевірка виводу. Cypress надає чіткий та зрозумілий API для опису цих дій та перевірок, що робить написання тестів зручним і зрозумілим. Однією з ключових особливостей Cypress є його здатність відтворювати кроки тесту в реальному часі в браузері, що дозволяє розробникам бачити точно, як тести виконуються та де виникають помилки. Cypress також підтримує автоматичне очікування елементів і AJAX-запитів, що зменшує необхідність ручного налаштування таймаутів.

Загалом, Cypress робить E2E тестування більш доступним та ефективним, що є важливим для забезпечення високої якості сучасних веб-застосунків.

2.8 Масштабованість та продуктивність

Масштабованість та продуктивність — це ключові фактори, які слід враховувати при виборі між фреймворками як React та Vue. Обидва ці фреймворки мають свої сильні сторони та особливості, які впливають на їх здатність масштабуватися та високу продуктивність у різних сценаріях використання.

React.

Масштабованість:

1. Компонентний підхід: React використовує компонентний підхід, що сприяє легкому масштабуванню застосунків. Великі застосунки можна легко розширювати, додаючи нові компоненти або модифікуючи існуючі.
2. Управління станом: за допомогою бібліотек управління станом, як Redux, React може ефективно масштабуватися для складних застосунків з розширеним управлінням стану.

Продуктивність:

1. Віртуальний DOM: React використовує віртуальний DOM, який оптимізує оновлення в реальному DOM, зменшуючи кількість дороговартісних операцій маніпуляції з DOM.
2. Оптимізація продуктивності: інструменти, як React DevTools, та методи життєвого циклу компонентів дозволяють глибоко оптимізувати продуктивність.

Vue.js.

Масштабованість:

1. Легкість розширення: Vue також пропонує компонентний підхід, що робить його легким для масштабування. Він простий у вивченні, що сприяє швидкому розвитку проектів.
2. Вбудована реактивність: Vue.js має вбудовану реактивність, що сприяє легкому управлінню станом в масштабованих застосунках.

Продуктивність:

1. Ефективне управління DOM: Подібно до React, Vue оптимізує маніпуляції з DOM, але робить це через свою реактивну систему, що відстежує залежності та автоматично оновлює DOM.
2. Легкість та швидкість: Vue часто вважається трохи швидшим за React для деяких сценаріїв через більш легковажний підхід. Vue часто має менший розмір пакета порівняно з React, що робить його швидшим для завантаження та менш вимогливим до ресурсів браузера.

2.9 Висновки дослідження методів і алгоритмів роботи SPA з використанням Reavt та Vue.js

У сфері сучасної веб-розробки, біліотека і фреймворк - React та Vue.js виступають як ключові гравці, пропонуючи різні підходи до створення односторінкових застосунків.

React, розроблений Facebook, використовує компонентний підхід та віртуальний DOM. Це сприяє ефективному оновленню інтерфейсу та підвищує продуктивність, особливо у великих та складних застосунках. Управління станом у React часто здійснюється через такі бібліотеки, як Redux, що дозволяє розробникам більш гнучко керувати станами додатків.

Vue.js, набув популярномті своєю інтуїтивністю та легкістю у використанні. Його компонентна структура також забезпечує масштабованість, водночас будучи більш доступною для новачків. Vue відрізняється вбудованою реактивністю та ефективним управлінням DOM, що робить його швидким та легковажним, особливо у базових сценаріях.

Вибір між React та Vue часто базується на конкретних вимогах проекту та досвіді розробників. React може бути кращим вибором для проектів, що вимагають детального управління станом та гнучкої архітектури. Vue, у свою чергу, ідеально підходить для проектів, де важливі швидкість розвитку та легкість у використанні.

3 ОСОБЛИВОСТІ РОЗРОБКИ СИСТЕМИ ОДНОСТОРІНКОВОГО ВЕБ-ЗАСТОСУНКУ З ВИКОРИСТАННЯМ REACT І VUE.JS

3.1 Вибір інструментів та налаштування робочого середовища

Вибір інструментів для розробки односторінкового додатку (SPA) має вирішальне значення, адже він визначає, наскільки гнучким, ефективним та відповідним буде кінцевий продукт.

Зі сторони React було прийняте рішення використовувати новіший підхід, а саме функціональний підхід. Вибір функціонального підходу в React замість класового зумовлений рядом важливих переваг. Функціональні компоненти, особливо з використанням хуків, спрощують процес створення додатків, роблячи код більш читабельним і легким для розуміння.

З погляду продуктивності, функціональні компоненти часто перевершують класові, особливо коли використовуються патерни оптимізації, такі як React.memo, які допомагають уникнути непотрібних перерендерів. Це робить функціональний підхід більш ефективним для сучасних веб-додатків.

Крім того, оскільки React спільнота і сам Facebook, який підтримує React, активно рухаються в напрямку функціонального підходу, це забезпечує кращу підтримку, оновлення та документацію, які використовують цей підхід. Враховуючи ці переваги, функціональний підхід в React вважається не тільки більш ефективним і сучасним, але й вказує на напрямок майбутнього розвитку веб-розробки. Як графічну складову було обрано Material UI.

Material UI є популярною бібліотекою компонентів для фронтенд-розробки, яка використовується разом з React. Її основна ціль - надати готовий набір компонентів, які відповідають принципам дизайну Material Design, розробленому Google. Material UI часто використовується у розробці веб-додатків та додатків, де потрібен сучасний, реактивний та зрозумілий для користувача інтерфейс. Її популярність зростає завдяки легкості використання, ефективності та гнучкості, які вона пропонує в процесі розробки.

З сторони Vue.js було прийняте рішення використовувати Composition API. Composition API, який став ключовою особливістю Vue 3, значно відрізняється від традиційного Options API, що використовувався в попередніх версіях Vue. Цей новий API привносить у розробку на Vue глибшу гнучкість та ефективність, особливо у великих та складних проектах. Однією з ключових переваг Composition API є його здатність упорядковувати та управляти логікою компонентів більш ефективно. Традиційно, в Options API, різні аспекти логіки компонента, такі як методи, обчислювані властивості та відстежувані дані, розподілялися по різних секціях об'єкта компонента. Це часто призводило до розкиданої та менш інтуїтивно зрозумілої структури, особливо в великих компонентах. Composition API розв'язує цю проблему, дозволяючи розробникам групувати відповідну логіку разом, незалежно від її природи. Це сприяє створенню більш організованого та читабельного коду. Composition API також вносить поліпшення в управлінні реактивністю. Розробники отримують більше контролю над реактивними станами та їх залежностями, що дозволяє більш тонко налаштовувати поведінку компонентів та оптимізувати продуктивність додатків. Нарешті, Composition API сприяє легшій інтеграції з іншими JavaScript бібліотеками та фреймворками. Використання стандартного JavaScript-коду в рамках Composition API робить процес інтеграції більш плавним та інтуїтивно зрозумілим, відкриваючи двері для більш широких можливостей у розробці.

У підсумку, Composition API в Vue 3 надає розробникам засоби для створення більш гнучких, ефективних та масштабованих веб-додатків. Його можливості упорядкування логіки, повторного використання коду, інтеграції з TypeScript, керування реактивністю та сумісності з іншими інструментами роблять його важливим внеском у еволюцію Vue.

З сторони грфічних фреймворків, було обрано Quasar. Quasar є високопродуктивним Vue.js-фреймворком, що дозволяє розробникам створювати швидкі та реактивні веб-додатки, односторінкові додатки (SPA). Його основною особливістю є можливість створювати різноманітні типи додатків із єдиним

кодовим базисом. Quasar ідеально підходить, як універсальний інструмент для створення різних типів додатків, маючи обмежені ресурси та час.

Як редактор коду було обрано VSCode, Visual Studio Code є високопродуктивним, безкоштовним редактором коду, розробленим Microsoft. Він доступний для Windows, macOS та Linux і став одним з найпопулярніших інструментів серед розробників по всьому світу. VSCode пропонує широкий спектр функцій, які підвищують продуктивність і полегшують процес розробки.

Також було встановлено Node.js, та npm для управління пакетами.

3.2 Ініціалізація проекту

Ініціалізація проекту React на TypeScript з використанням Material-UI (MUI) та Redux вимагає декілька кроків, які забезпечать правильну налаштування та інтеграцію цих технологій: створення проекту, встановлення всіх необхідних пакетів, налаштування їх для використання, підключення та налаштування системи git.

Створення нового проекту React: перш за все, потрібно створити новий проект React. Це можна зробити за допомогою команди create-react-app, яка є офіційним генератором проектів для React - «npx create-react-app my-app --template typescript», ця команда створить новий проект React з назвою "my-app", вже налаштований для використання TypeScript.

Встановлення Material-UI (MUI): наступним кроком буде встановлення Material-UI, який надає готові компоненти для React - «npm install @mui/material @emotion/react @emotion/styled»

Встановлення пакетів React Router: спочатку потрібно встановити основний пакет React Router та його типи для TypeScript – «npm install react-router-dom @types/react-router-dom», далі необхідно його налаштувати, необхідно створити основний маршрут, використовуючи BrowserRouter та Route з react-router-dom. В

App.tsx, імпортувати необхідні компоненти та створити базову структуру маршрутів, рисунок 3.1.

```
import React from 'react';
import {
  BrowserRouter as Router,
  Switch,
  Route,
  Link
} from 'react-router-dom';

import Home from './Home';
import About from './About';

function App() {
  return (
    <Router>
      <div>
        <nav>
          <ul>
            <li>
              <Link to="/">Home</Link>
            </li>
            <li>
              <Link to="/about">About</Link>
            </li>
          </ul>
        </nav>

        <Switch>
          <Route path="/about">
            <About />
          </Route>
          <Route path="/">
            <Home />
          </Route>
        </Switch>
      </div>
    </Router>
  );
}
```

Рисунок 3.1 - Базова структура реакт-роутера

Встановлення Redux та React-Redux. Наступним кроком буде встановлення Redux для управління станом додатку та React-Redux для інтеграції Redux з React. Для цього необхідно виконати команду «npm install redux react-redux @reduxjs/toolkit @types/react-redux». Також необхідно його налаштувати, для цього

потрібно створити структуру файлів стору. Основним файлом буде store.ts структура якого зображена на рисунку 3.2.

```
import { configureStore } from '@reduxjs/toolkit';
import userReducer from './features/userSlice';

const store = configureStore({
  reducer: {
    user: userReducer
    // інші редюсери
  }
});

// Визначення типу для кореневого стану
export type RootState = ReturnType<typeof store.getState>;

// Визначення типу для використання dispatch у додатку
export type AppDispatch = typeof store.dispatch;

export default store;
```

Рисунок 3.2 - Структура файлу store.ts

Далі потрібно підключити додаток до цього сховища, для цього необхідно огорнути корневий компонент index.tsx в провайдер, що імпортується з бібліотеки react-redux, та передати в нього створене сховище. Побачити це можна на рисунку 3.3.

```
import React from 'react';
import ReactDOM from 'react-dom';
import { Provider } from 'react-redux';
import store from './store';
import App from './App';

ReactDOM.render(
  <Provider store={store}>
    <App />
  </Provider>,
  document.getElementById('root')
);
```

Рисунок 3.3 - Підключення store до додатку

Ініціалізація проекту Vue з використанням Composition API на TypeScript, Quasar та Vuex вимагає кількох кроків, щоб правильно налаштувати та інтегрувати ці технології.

Перш за все, потрібно встановити Quasar CLI, якщо він ще не встановлений: «npm install -g @quasar/cli», наступним кроком буде створення проекту за допомогою команди «quasar create my-project», Під час ініціалізації проекту буде запропоновано вибрати набір опцій. необхідно вибрати TypeScript та Vuex.

Налаштування Vuex Store. Необхідно створити Vuex store, тобто створити папку store з файлом index.ts приклад простого Vuex store зображено на рисунку 3.4.

```
import { createStore } from 'vuex';
import UserModule from './user';

export default createStore({
  modules: {
    user: UserModule
  }
});

// Додатково визначаємо тип для кореневого стану
export type RootState = {
  user: UserState;
};
```

Рисунок 3.4 - Головний файл сховища Vuex

Після створення файлу store.ts, необхідно додати його до додатку. Для додавання Vuex store до Vue додатку, потрібно імпортувати його в src/index.ts та додати його до Vue екземпляра, приклад такого файлу зображено на рисунку 3.5.

```
import { createApp } from 'vue';
import App from './App.vue';
import store from './store';

const app = createApp(App);
app.use(store);
app.mount('#app');
```

Рисунок 3.5 - Приклад підключення Vuex до додатку

3.3 Розробка функціональності

Розробка функціональності в React з використанням функціональних компонентів включає кілька ключових кроків. Функціональні компоненти в React дозволяють писати більш чистий та модульний код, використовуючи хуки для управління станом, ефектами та іншими реактивними особливостями. Хорошим прикладом буде створення простого функціонального компонента, який виконує запит до API та використовує Redux для управління станом.

Першочергово створюється новий функціональний компонент. Наприклад, компонент UserProfile, який відображає інформацію про користувача рисунок 3.6.

У прикладі вище, useEffect використовується для виконання побічного ефекту — запиту до API. Коли компонент монтується або коли змінюється userId, виконується запит до API, і отримані дані зберігаються у локальному стані компонента через useState. Проте набагато зручніше використовувати менеджер стану Redux, а саме асинхронні action для запиту та store для збереження інформації про користувача, приклад такого виконання на рисунку 3.7, також приклад action на рисунку 3.8, селектор на рисунку 3.9, а сам reduce slice на рисунку 3.10.

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

// Визначаємо типи даних, які повертає API
interface User {
  name: string;
  email: string;
  // інші поля, які очікуються
}

// Визначаємо типи для пропсів
interface UserProfileProps {
  userId: string;
}

function UserProfile({ userId }: UserProfileProps) {
  const [user, setUser] = useState<User | null>(null);

  useEffect(() => {
    // Запит до API для отримання даних про користувача
    fetch(`https://api.example.com/users/${userId}`)
      .then(response => {
        if (!response.ok) {
          throw new Error('Network response was not ok');
        }
        return response.json();
      })
      .then((data: User) => setUser(data)) // Вказуємо очікуваний тип для data
      .catch(error => console.error('Error fetching user:', error));
  }, [userId]); // Залежності useEffect

  // Відображення загрузки, якщо дані ще не отримані
  if (!user) {
    return <div>Loading...</div>;
  }

  // Якщо дані є, відображаємо їх
  return (
    <div>
      <h1>{user.name}</h1>
      <p>Email: {user.email}</p>
      {/* інші дані користувача */}
    </div>
  );
}

export default UserProfile;
```

Рисунок 3.6 - Приклад простого функційного компонента, що відображає данні користувача

```
import React, { useEffect } from 'react';
import { useSelector, useDispatch } from 'react-redux';
import { fetchUser } from '../store/actions/user';
import { getUserData, getUserLoading } from '../store/selectors/user';
import { RootState } from '../store/store';

// Визначення типу для пропсів компонента
interface UserProfileProps {
  userId: string;
}

// Визначення типів для даних користувача
interface UserData {
  name: string;
  email: string;
  // інші поля, які можуть бути частиною даних користувача
}

const UserProfile: React.FC<UserProfileProps> = ({ userId }) => {
  const dispatch = useDispatch();

  // Використовуйте типізацію селектора, якщо вона доступна
  const { name, email, ...restUserData } = useSelector<RootState, UserData>(getUserData);
  const loading = useSelector<RootState, boolean>(getUserLoading);

  useEffect(() => {
    dispatch(fetchUser(userId));
  }, [dispatch, userId]);

  return !loading ? (
    <div>
      <h1>{name}</h1>
      <p>Email: {email}</p>
      {/* інші дані користувача */}
    </div>
  ) : (
    <div>Loading...</div>
  );
};

export default UserProfile;
```

Рисунок 3.7 - Приклад функційної компоненти з використанням redux

```
import { createAsyncThunk } from '@reduxjs/toolkit';
import { push } from 'connected-react-router';

// Визначення типу для вхідного параметру функції
export const fetchUser = createAsyncThunk(
  'user/fetchUser',
  async (userId: string, { dispatch, rejectWithValue }) => {
    try {
      const response = await fetch(`https://api.example.com/users/${userId}`);
      if (!response.ok) {
        throw new Error('Server Error!');
      }
      const data: UserType = await response.json();
      return data;
    } catch (error) {
      dispatch(push('/not-found')); // Редірект на сторінку "Not Found"
      return rejectWithValue((error as Error).message);
    }
  }
);

interface UserType {
  id: string;
  name: string;
  email: string;
  // та інші поля за потреби
}
```

Рисунок 3.8 - Приклад створення асинхронного action

```
// Тип кореневого стану додатку
interface RootState {
    user: UserState;
  }

  // Селектори з використанням цих типів
  export const getUser = (state: RootState) => state.user.user;
  export const getUserLoading = (state: RootState) => state.user.loading;
```

Рисунок 3.9 - Приклад селекторів

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

// Визначення типів для стану
interface UserState {
  user: null | { name: string; email: string };
  loading: boolean;
  error: null | string;
}

// Початковий стан
const initialState: UserState = {
  user: null,
  loading: false,
  error: null
};

const userSlice = createSlice({
  name: 'user',
  initialState,
  reducers: {
    setUser: (state, action: PayloadAction<{ name: string; email: string }>) => {
      state.user = action.payload;
    },
    setLoading: (state, action: PayloadAction<boolean>) => {
      state.loading = action.payload;
    },
    setError: (state, action: PayloadAction<string>) => {
      state.error = action.payload;
    }
  }
});

export const { setUser, setLoading, setError } = userSlice.actions;
export default userSlice.reducer;
```

Рисунок 3.10 - Приклад reduce slice

Щоб розглянути створення функціональності за допомогою Vue Composition API, аналогічно до реактового компонента UserProfile, створимо Vue компонент, який виконує запит до API для отримання даних користувача та відображає їх. Використовуючи Composition API, використано ref для реактивних даних, computed для обчислюваних властивостей і onMounted для виконання ефектів після монтування компонента, компонент зображено на рисунку 3.11, приклад інтерфейсу користувача зображений на рисунку 3.12, приклад модуля vuex на рисунку 3.13, тако ж приклад компонента з використанням vuex зображено на рисунку 3.14.

```
<template>
  <div v-if="loading">Loading...</div>
  <div v-else>
    <h1>{{ user.name }}</h1>
    <p>Email: {{ user.email }}</p>
    <!-- інші дані користувача -->
  </div>
</template>

<script lang="ts">
import { defineComponent, onMounted, reactive, toRefs } from 'vue';
import { User } from './types'; // Імпортування типу User

export default defineComponent({
  props: {
    userId: {
      type: String,
      required: true,
    },
  },
  setup(props) {
    const state = reactive({
      user: null as User | null,
      loading: false,
      error: null as Error | null,
    });

    onMounted(async () => {
      state.loading = true;
      try {
        const response = await fetch(`https://api.example.com/users/${props.userId}`);
        if (!response.ok) {
          throw new Error('Failed to fetch');
        }
        const data = await response.json() as User;
        state.user = data;
      } catch (e) {
        state.error = e as Error;
      } finally {
        state.loading = false;
      }
    });

    return {
      ...toRefs(state),
    };
  },
});
</script>
```

Рисунок 3.11 - Приклад компонента Vue

```
export interface User {
    id: string;
    name: string;
    email: string;
  }
```

Рисунок 3.12 - Приклад інтерфейсу користувача

```
import { Module, Mutation, Action, VuexModule } from 'vuex-module-decorators';

// Визначення типів для користувача
export interface User {
  id: string;
  name: string;
  email: string;
}

// Визначення типу для стану користувача
export interface UserState {
  data: User | null;
  loading: boolean;
}

@Module({ namespaced: true, stateFactory: true })
export default class User extends VuexModule implements UserState {
  data = null;
  loading = false;

  @Mutation
  SET_USER(user: User) {
    this.data = user;
  }

  @Mutation
  SET_LOADING(loading: boolean) {
    this.loading = loading;
  }

  @Action
  async fetchUser(userId: string) {
    this.context.commit('SET_LOADING', true);
    try {
      const response = await fetch(`https://api.example.com/users/${userId}`);
      if (!response.ok) throw new Error('Error fetching user');
      const userData = await response.json() as User;
      this.context.commit('SET_USER', userData);
    } catch (e) {
      console.error(e);
    } finally {
      this.context.commit('SET_LOADING', false);
    }
  }

  // Геттери
  get user() {
    return this.data;
  }

  get isLoading() {
    return this.loading;
  }
}
```

Рисунок 3.13 - Приклад модуля для vuex

```
<template>
  <div v-if="isLoading">Loading...</div>
  <div v-else-if="userData">
    <h1>{{ userData.name }}</h1>
    <p>Email: {{ userData.email }}</p>
    <!-- інші дані користувача -->
  </div>
  <div v-else>
    User not found.
  </div>
</template>

<script lang="ts">
import { defineComponent, computed } from 'vue';
import { useStore } from 'vuex';
import { RootState } from '@/store'; // Імпортуємо тип для кореневого стану

export default defineComponent({
  props: {
    userId: {
      type: String,
      required: true,
    },
  },
  setup(props) {
    const store = useStore<RootState>(); // Використовуємо типізований стор

    // Використовуємо Vuex геттери та дії з Composition API
    const userData = computed(() => store.state.user.user);
    const isLoading = computed(() => store.state.user.loading);

    // Завантажуємо дані користувача
    store.dispatch('user/fetchUser', props.userId);

    return {
      userData,
      isLoading,
    };
  },
});
</script>
```

Рисунок 3.14 - Приклад компоненту vue з використанням vuex

3.4 Тестування

Тестування є критично важливим для розробки програмного забезпечення, оскільки воно гарантує надійність, підвищує якість, знижує ризики помилок та сприяє ефективному виявленню та виправленню багів, забезпечуючи кращий користувацький досвід.

Для прикладу написання тестів чудово підійде тест на Jest для компонентів з минулого розділу, приклад такого тесту для React можна побачити на рисунку 3.15. У цьому тесті:

1. Створено моковий Redux стор, який використовується для надання контексту для компонента UserProfile.
2. Рендериться компонент UserProfile в контексті мокового стору.
3. Перевіряється, що текст "Loading..." спочатку відображається.
4. Використовується waitFor з @testing-library/react для очікування, поки асинхронні операції будуть завершені.
5. Після завантаження даних йде перевірка, що ім'я та email користувача відображаються на сторінці.
6. Відбувається перевірка того, що fetch викликається з правильним URL.

```
import React from 'react';
import { Provider } from 'react-redux';
import { render, screen, waitFor } from '@testing-library/react';
import '@testing-library/jest-dom/extend-expect';
import UserProfile from './UserProfile';
import { createStore, applyMiddleware } from 'redux';
import thunk from 'redux-thunk';
import rootReducer from '../store/reducers';

// Створюємо моковий Redux стор для тесту
const store = createStore(rootReducer, applyMiddleware(thunk));

// Мок fetch API
global.fetch = jest.fn(() =>
  Promise.resolve({
    json: () => Promise.resolve({ name: 'John Doe', email: 'john@example.com' }),
  })
);

beforeEach(() => {
  fetch.mockClear();
});

test('renders user data', async () => {
  const userId = '1'; // Припустимо, що у нас є ID користувача

  // Рендеримо компонент з моковим стором та пропсами
  render(
    <Provider store={store}>
      <UserProfile userId={userId} />
    </Provider>
  );

  // Перевіряємо, що спочатку показується текст завантаження
  expect(screen.getByText(/loading/i)).toBeInTheDocument();

  // Чекаємо на завершення асинхронних дій
  await waitFor(() => {
    // Перевіряємо, що текст завантаження більше не показується
    expect(screen.queryByText(/loading/i)).not.toBeInTheDocument();
  });

  // Перевіряємо, що ім'я та email користувача відображаються на сторінці
  expect(screen.getByText('John Doe')).toBeInTheDocument();
  expect(screen.getByText('john@example.com')).toBeInTheDocument();

  // Перевіряємо, що fetch був викликаний з правильним URL
  expect(fetch).toHaveBeenCalledWith(`https://api.example.com/users/${userId}`);
});
```

Рисунок 3.15 - Приклад тесту React компоненту

Також приклад тесту на Jest, створено і для Vue компонента, рисунок 3.16.

У цьому тесті:

1. Створено моковий Vuex стор із потрібним модулем user.
2. Використано моковий fetch для імітації відповіді від API.
3. Змонтовано компонент UserProfile з моковим стором і моковим параметром маршруту.
4. Використано flushPromises для чекання завершення асинхронних операцій у компоненті.
5. Перевірено, чи була викликана дія fetchUser із правильним параметром.
6. Перевірено, чи правильно відображаються ім'я та email користувача після завантаження даних.

```
import { mount, flushPromises } from '@vue/test-utils';
import { createStore } from 'vuex';
import UserProfile from '@/components/UserProfile.vue';

// Мок для fetch API
global.fetch = jest.fn(() =>
  Promise.resolve({
    ok: true,
    json: () => Promise.resolve({ name: 'John Doe', email: 'john@example.com' }),
  })
) as jest.Mock;

describe('UserProfile.vue', () => {
  it('displays user data after fetch', async () => {
    // Створюємо моковий Vuex стор з модулем user
    const store = createStore({
      modules: {
        user: {
          namespaced: true,
          state: {
            user: null,
            loading: false,
          },
          getters: {
            getUserData: (state) => state.user,
            getUserLoading: (state) => state.loading,
          },
          actions: {
            fetchUser: jest.fn(), // Мок для дії fetchUser
          },
        },
      },
    });

    // Монтуємо компонент з моковим стором
    const wrapper = mount(UserProfile, {
      global: {
        plugins: [store],
        mocks: {
          $route: {
            params: { userId: '123' }, // Передаємо моковий параметр маршруту
          },
        },
      },
      props: {
        userId: '123',
      },
    });

    // Чекаємо на завершення асинхронних операцій
    await flushPromises();

    // Перевіряємо чи була викликана дія fetchUser
    expect(store.dispatch).toHaveBeenCalledWith('user/fetchUser', '123');

    // Перевіряємо чи відображаються дані користувача
    expect(wrapper.text()).toContain('John Doe');
    expect(wrapper.text()).toContain('john@example.com');
  });
});
```

Рисунок 3.16 - Приклад тесту компонента Vue

3.5 Висновки дослідження особливостей розробки систем односторінкових веб-додатків

Розробка односторінкових веб-застосунків (SPA) з використанням React та Vue.js охоплює спектр підходів та практик, які відображають сучасні тенденції у веб-розробці.

Аналізуючи створені компоненти та їх тести для React та Vue з використанням відповідних екосистем інструментів, можна зробити висновок, що обидва фреймворки забезпечують ефективні та сучасні засоби для розробки SPA. React компоненти, з їхніми хуками та контекстом Redux, демонструють потужність управління станом і сайд ефектами, в той час як Vue використовує Composition API для досягнення аналогічної модульності та реактивності. Тести для цих компонентів, написані з використанням Jest та інструментів, специфічних для кожного фреймворку, обидві платформи підтримують надійне тестування з моками та асинхронними запитами. Використання моків для API запитів дозволяє ізолювати компоненти від зовнішніх залежностей, тим самим забезпечуючи точність та предбачуваність тестів. Це важливо для підтримки високої якості коду та легкості обслуговування застосунків.

В кінцевому підсумку, як React, так і Vue, пропонують розробникам потужні інструменти не тільки для створення, але й для тестування односторінкових застосунків.

4 ОХОРОНА ПРАЦІ ТА БЕЗПЕКА В НАДЗВИЧАЙНИХ СИТУАЦІЯХ

4.1 Охорона праці

Охорона праці включає комплекс заходів та засобів, що охоплює правові, соціально-економічні, а також організаційно-технічні та лікувально-профілактичні аспекти, які мають на меті забезпечити безпеку особи під час виконання нею трудових обов'язків.

Темою дипломної роботи є розробка односторінкового веб-застосунку

з використанням Vue.js і React: порівняльний аналіз продуктивності, користувацького досвіду та доцільності. Враховуючи, що виконання наукової роботи відбувається за допомогою персонального комп'ютера, важливо дотримуватися встановлених правил та норм безпеки, пов'язаних з роботою на комп'ютерних системах. Саме тому було важливо вивчити та врахувати відповідні нормативні документи, які регулюють умови робочих місць і приміщень, де застосовуються персональні комп'ютери.

Під час дослідження та розробки односторонніі веб застосунків Важливо вживати заходів для запобігання шкідливому впливу факторів виробництва. Це необхідно для створення безпечних умов роботи в приміщенні, де розташоване робоче місце, для цього необхідно дотримуватися норм і правил що вказані в НПАОП 0.00-7.15-18 «Вимоги щодо безпеки та захисту здоров’я працівників під час роботи з екранними пристроями» та ДСанПіН 3.3.2.007-98 “Гігієнічні вимоги до організації роботи з візуальними дисплейними терміналами електронно-обчислювальних машин”.

Робоче місце знаходиться в приміщенні на 4 поверсі, оскільки згідно з ДСанПіН 3.3.2.007-98, не може знаходитися у підвальному приміщенні чи на цокольному поверсі. Площа одного робочого місця 10 кв. м, а об’єм 30 куб. м., що є допустимою площею для одного робочого місця. Приміщення має природнє та штучне освітлення відповідно до СНиП II-4-79. Природнє освітлення здійснюється через світлові прорізи, орієнтовані переважно на північ, саме це забезпечує

коефіцієнт природньої освітленості не нижчий 1.5%. Джерелом штучного освітлення є підвісні та настільні лампи. Вікна в приміщенні обладнані шторами та зовнішніми козирками. Приміщення не межує з приміщеннями в яких рівні шуму і вібрації перевищують допустимі значення (виробничі цехи, майстерні тощо) за СН 3223-85, СН 3044-84, ГР 2411-81, ГОСТ 12.1.003-2014. Параметри шуму в приміщенні відповідають вимогам СН 3223-85, ГОСТ 12.1.003-2014, ГОСТ 12.1.012:2008. Приміщення в якому знаходиться робоче місце обладнане системою опалення та кондиціонування, а нормовані параметри мікроклімату, іонного складу повітря, вмісту шкідливих речовин відповідають вимогам СН 4088-86, СН 2152-80, ГОСТ 12.1.005-88, НПАОП 24.0-1.01-08. В приміщенні проводиться щоденне вологе прибирання. Для внутрішнього оздоблення приміщення використані дифузно-відбивні матеріали з коефіцієнтами відбиття для стелі 0,7, для стін 0,6, а покриття підлоги є матове з коефіцієнтом відбиття 0,3. Поверхня підлоги є рівною, неслизькою, з антистатичними покриттям. Для оздоблення інтер'єру не використовувалися полімерні матеріали (деревинно-стружкові плити, шпалери, що миються, рулонні синтетичні матеріали, шаруватий паперовий пластик тощо), що виділяють у повітря шкідливі хімічні речовини.

Згідно НПАОП 0.00-7.15-18 «Вимоги щодо безпеки та захисту здоров'я працівників під час роботи з екранними пристроями», для безпеки життєдіяльності під час роботи з персональним комп'ютером необхідно дотримуватись вимог до робочих місць, а саме:

1. Робочі місця працівників з екранними пристроями мають бути спроектовані так і мати такі розміри, щоб працівники мали простір для зміни робочого положення та рухів.
2. Для забезпечення безпеки та захисту здоров'я працівників усе випромінювання від екранних пристроїв має бути зведене до гранично допустимого рівня (вплив на людину факторів довкілля - шуму, вібрації, забруднювачів, температури тощо, який не спричиняє соматичних або психічних розладів, а також змін стану здоров'я, працездатності, поведінки,

що виходять за межі пристосувальних реакцій) з погляду безпеки та охорони здоров'я працівників.

3. Організація робочого місця працівника з екранними пристроями має забезпечувати відповідність усіх елементів робочого місця та їх розташування ергономічним, антропологічним, психофізіологічним вимогам, а також характеру виконуваних робіт.

4. Освітлення робочого місця працівника з екранними пристроями має створювати відповідний контраст між екраном і навколишнім середовищем (з урахуванням виду роботи) та відповідати вимогам ДСанПІН 3.3.2.007-98.

5. Мікроклімат виробничих приміщень з робочими місцями працівників з екранними пристроями має підтримуватись на постійному рівні та відповідати вимогам Санітарних норм мікроклімату виробничих приміщень ДСН 3.3.6.042-99

6. Робочий стіл або робоча поверхня повинні бути достатнього розміру та мати поверхню з низькою відбивною здатністю, допускати гнучкість під час розміщення екрана, клавіатури, документів і відповідного устаткування.

7. Робоче крісло має бути стійким і дозволяти працівнику з екранними пристроями легко рухатися та займати зручне положення. Сидіння має регулюватися по висоті, спинка сидіння - як по висоті, так і по нахилу. Слід передбачати підніжку для тих, кому це необхідно для зручності.

Також законом передбачено мінімальні вимоги безпеки під час роботи з екранними пристроями, а саме:

1. Щодня перед початком роботи необхідно очищати екранні пристрої від пилу та інших забруднень.

2. Після закінчення роботи екранні пристрої слід відключати від електричної мережі.

3. У разі виникнення аварійної ситуації необхідно негайно відключити екранний пристрій від електричної мережі.

4. Не допускається:

- Виконувати технічне обслуговування, ремонт і налагодження екранних пристроїв безпосередньо на робочому місці працівника під час роботи з екранними пристроями;
- Відключати захисні пристрої, самочинно проводити зміни у конструкції та складі екранних пристроїв або їх технічне налагодження;
- Працювати з екранними пристроями, у яких під час роботи виникають нехарактерні сигнали, нестабільне зображення на екрані та інші несправності.

5. Під час виконання робіт операторського типу, пов'язаних з нервово-емоційним напруженням, у приміщеннях під час роботи з екранними пристроями, на пультах і постах керування технологічними процесами та в інших приміщеннях мають дотримуватися оптимальні умови мікроклімату відповідно до вимог ДСН 3.3.6.042-99.

Згідно НПАОП 0.00-7.15-18 «Вимоги щодо безпеки та захисту здоров'я працівників під час роботи з екранними пристроями», в екранних пристроїв є свої мінімальні вимоги безпеки, а саме:

1. Екранні пристрої не мають бути джерелом ризику для працівників.
2. Усе випромінювання, за винятком видимої частини електромагнітного спектра, має бути зведене до незначного рівня з погляду безпеки і охорони здоров'я працівників.
3. Символи на екранних пристроях мають бути чіткими, відповідного розміру. Між символами і рядками символів має бути належна відстань.
4. Зображення на екрані має бути стабільним, без миготінь або інших видів нестабільності.

5. Яскравість та/або контрастність символів має легко регулюватися працівником під час роботи з екранними пристроями, а також швидко адаптуватися до навколишніх умов.

6. Вибираючи екрани, слід надавати перевагу таким екранам, які легко та вільно повертаються і нахиляються відповідно до потреби працівника.

7. За необхідності може використовуватись окрема підставка або регульований стіл для розміщення екрана.

8. Екран не має відблискувати або відбивати світло, щоб не викликати дискомфорту у працівника під час роботи з екранними пристроями.

9. Вибираючи клавіатуру, слід надавати перевагу такій клавіатурі, яка відкидається і є автономною (відокремленою від екрана), щоб працівник міг вибрати зручну робочу позу й уникнути втоми рук (кисті і верхньої частини руки).

10. Поверхня клавіатури має бути матовою, щоб уникнути віддзеркалювання. Розташування клавіш і самі клавіші мають полегшувати роботу із клавіатурою. Позначення клавіш повинно бути достатньо контрастним і розбірливим.

11. Устаткування, яке входить до робочої станції, не має виділяти надлишкового тепла, що може спричинити незручності працівникам під час роботи з екранними пристроями.

12. Під час розробки, вибору, замовлення та модифікації програмного забезпечення, а також під час розробки завдань, що передбачають використання устаткування з екранними пристроями, роботодавець має керуватися таким програмним забезпеченням, яке відповідає розв'язуваним завданням і є простим у використанні, а де необхідно - адаптованим до рівня знань і досвіду працівника.

Приміщення оснащене порошковим переносним вогнегасником відповідно до основних вимог пожежної безпеки приміщення, у яких знаходиться

персональний комп’ютер. Підходи до засобів пожежогасіння є вільними. Також, згідно вимог НАПБ Б.06.004-2005 «Перелік однотипних за призначенням об'єктів, які підлягають обладнанню автоматичними установками пожежогасіння та пожежної сигналізації», у приміщенні де здійснюється робота з персональним комп’ютером, робочі місця обладнанні системою автоматичної пожежної сигналізації з димовим пожежним сповіщувачем.

Дослідження та розробку односторінкового веб застосунку проведено з дотриманням вимог техніки безпеки, стандартів і правила щодо влаштування робочих місць де використовують комп’ютерну техніку.

4.2 Шкідливі та небезпечні фактори при використанні комп’ютерних систем та захист від них користувачів.

Користувачі, до чиїх обов’язків відноситься постійна або періодична праця за комп’ютером, піддаються шкідливому впливу виробничих факторів, основними з яких є фізичні, хімічні та психофізіологічні.

Небезпечні і шкідливі виробничі чинники:

1. Підвищена температура поверхонь ПК.
2. Підвищена або знижена температура повітря робочої зони.
3. Виділення в повітря робочої зони ряду хімічних речовин.
4. Підвищена або знижена вологість повітря.
5. Підвищений або знижений рівень негативних і позитивних аероіонів.
6. Підвищене значення напруги в електричному ланцюзі, замикання.
7. Підвищений рівень статичної електрики.
8. Підвищений рівень електромагнітних випромінювань.
9. Підвищена напруженість електричного поля.

10. Відсутність або недолік природного світла.
11. Недостатня штучна освітленість робочої зони.
12. Підвищена яскравість світла.
13. Підвищена контрастність.
14. Прямі і непрямі відблиски.
15. Зорова напруга.
16. Монотонність трудового процесу.
17. Нервово-емоційні перевантаження.
18. Фізично шкідливі і небезпечні чинники.

До фізичних шкідливих і небезпечних чинників відносяться: підвищені рівні електромагнітного, рентгенівського, ультрафіолетового і інфрачервоного випромінювання; підвищений рівень статичної електрики і запиленої повітря робочої зони; підвищений вміст позитивних аеронів і понижений вміст негативних аеройонов в повітрі робочої зони; підвищений рівень бліків і засліпленої; нерівномірність розподілу яскравості в полі зору; підвищене значення напруги в електричному ланцюзі, замикання якого може статися через тіло людини.

Хімічні шкідливі і небезпечні чинники наступні: підвищений вміст в повітрі робочої зони двоокису вуглецю, озону, аміаку, фенолу і формальдегіду.

Психофізіологічні шкідливі і небезпечні чинники: напруга зору і уваги; інтелектуальні, емоційні і тривалі статичні навантаження; монотонність праці; великий об'єм інформації, що обробляється в одиницю часу; нераціональна організація робочого місця.

Типовими відчуттями, які відчувають до кінця робочого дня оператори ПЕОМ, є: головний біль, різь в очах, що тягнуть болі в м'язах шиї, рук і спини, сверблячка шкіри обличчя і т.д. Пережиті день за днем, ці нездужання призводять до мігрені, часткової втрати зору, сколіозу, шкірним запаленням та іншим небажаним явищам. Розвиваються нездужання не тільки знижують працездатність, але і підривають здоров'я людей. На стан здоров'я оператора ЕОМ можуть впливати й такі шкідливі фактори, як тривалий незмінне положення тіла, що викликає м'язово-скелетні порушення; постійне напруження очей; вплив радіації; вплив

електростатичних і електромагнітних полів та інші. Тривала і інтенсивна робота на комп'ютері може стати джерелом важких професійних захворювань, таких, як травма навантажень, що повторюються, що є нездужаннями, що поступово накопичуються, перехідними в захворювання нервів, м'язів і сухожиль руки.

До професійних захворювань, пов'язаних з тривалою і інтенсивною роботою, відносяться:

1. Тендовагініт — запалення сухожиль кисті, зап'ястя, плеча;
2. Тендосиновіт — запалення синовіальной оболонки сухожильної підстави кисті і зап'ястя;
3. Синдром зап'ястного каналу (СЗК) – викликається утиском серединного нерва в зап'ястному каналі. Травма, що накопичується, викликає утворення продуктів розпаду в області зап'ястного каналу, внаслідок чого спочатку виникає набряк, а потім СЗК.

Засоби захисту. Режим праці та відпочинку при роботі з персональною електронно-обчислювальною машиною залежить від категорії трудової діяльності. Всі роботи з ПЕОМ ділять на три категорії. Перша - епізодичне зчитування і робота з інформацією не більше 2-х годин за 8-годинну робочу зміну. Друга - зчитування інформації або творча робота не більше 4-х годин за восьми годинну зміну. Третя - зчитування інформації або творча робота тривалістю більше 4-х годин за зміну.

Якщо у приміщенні експлуатується більше одного комп'ютера, то треба врахувати, що на користувача одного комп'ютера можуть впливати випромінювання від інших, в першу чергу бокових, а також і задньої стінки сусіднього дисплея. Тому необхіден захист спеціальними фільтрами і щоб користувач розміщався від бічних і задніх стінок інших дисплеїв на відстані не ближче одного метра.

4.3 Оцінка стійкості системи управління підприємством.

Забезпечення техногенної безпеки на об’єктах господарювання здійснюється шляхом проведення комплексу заходів щодо запобігання можливим НС.

Запобігання НС проводиться щляхом зниження ризиків виникнення (відвернення) та пом’якшення наслідків (зменшення втрат та збитків) за такими напрямками:

1. Моніторинг і прогнозування НС.
2. Раціональне розміщення продуктивних сил на території з урахуванням природної і техногенної безпеки.
3. Відвернення, у межах можливого, деяких несприятливих і небезпечних природних явищ та процесів шляхом систематичного зниження накопиченого руйнівного потенціалу.
4. Відвернення аварій і техногенних катастроф шляхом підвищення технологічної безпеки виробничих процесів та експлуатаційної надійності обладнання.
5. Розробка і здійснення інженерно-технічних заходів, спрямованих на усунення джерел НС, пом’якшення їх наслідків, захист населення і матеріальних засобів.
6. Підготовка об’єктів економіки і систем життєзабезпечення населення до роботи в умова НС.
7. Декларування промислової безпеки.
8. Ліцензування діяльності ОПН.
9. Страхування відповідальності за завдану шкоду внаслідок експлуатації ОПН.
10. Проведення державної експертизи у сфері запобігання НС.
11. Державний нагляд і контроль з питань природної і техногенної безпеки.

12. Інформування населення про потенційні природні та техногенні загрози на території, де воно проживає.

13. Підготовка населення у сфері захисту від НС.

Кожен напрям реалізується шляхом планування і виконання відповідних заходів.

Ефективність економіки держави залежить від того, наскільки окремі галузі господарства здатні стійко працювати не тільки в звичайних умовах, а і в умовах надзвичайних ситуацій мирного та воєнного часу.

Значні руйнування, пожежі та втрати серед населення, викликані наслідками надзвичайних ситуацій, можуть стати причиною різкого скорочення випуску промислової та сільськогосподарської продукції, а отже і зниження економічного потенціалу держави. Виникає необхідність завчасного прийняття заходів щодо забезпечення стійкої роботи промислових об’єктів на випадок виникнення надзвичайних ситуацій.

Вивчення можливих надзвичайних ситуацій, характерних для даної місцевості та даного виробництва, дозволяє диференційовано і найбільш спрямовано підходити до розробки та здійснення заходів, які можуть запобігти або пом’якшити наслідки аварій, катастроф та стихійного лиха.

Під стійкістю роботи об’єкта промисловості розуміють його здатність в умовах надзвичайних ситуацій випускати продукцію в запланованому об'ємі, а при отриманні слабких і середніх руйнувань, при пожежах, повенях, зараженні місцевості, а також, при порушенні зв'язків по кооперації і постачанню відновлювати виробництво в мінімальні терміни.

Стійкість роботи об'єктів що не виробляють матеріальні цінності, визначається їх здатністю виконувати свої функції в умовах надзвичайних ситуацій.

На стійкість роботи промислового об’єкта впливають такі фактори :

1. Захищеність робітників та службовців від уражаючих факторів надзвичайних ситуацій.

2. Здатність інженерно-технічного комплексу об’єкта (будівель, споруд, обладнання та комунально-енергетичних систем) протистояти руйнуючій дії уражаючих факторів аварій, катастроф, стихійного лиха та сучасної зброї.

3. Надійність постачання об’єкта електроенергією, водою, паливом, комплектуючими та сировиною.

4. Підготовленість об’єкта до проведення аварійно-рятувальних та відбудовних робіт.

5. Оперативність управління виробництвом та здійсненням заходів ЦЗ в надзвичайних ситуаціях.

Дані фактори визначають основні шляхи підвищення стійкості роботи об'єктів промисловості в умовах надзвичайних ситуацій, це:

1. Забезпечення надійного захисту робітників і службовців від уражаючих чинників в надзвичайних ситуаціях.

2. Захист основних виробничих фондів від руйнуючого впливу аварій, катастроф, стихійного лиха і засобів ураження.

3. Забезпечення стійкого постачання всім необхідним для випуску запланованої продукції.

4. Підготовка до відновлення порушеного виробництва.

5. Підвищення надійності і оперативності управління виробництвом і цивільним захистом.

Способи підвищення стійкості (надійності) роботи об’єктів промисловості в умовах надзвичайних ситуацій:

1. Нагромадження фонду захисних споруд ЦЗ і засобів індивідуального захисту.

2. Будівництво важливих підприємств за межами зон можливих руйнувань.

3. Будівництво підприємств-дублерів.

4. Розширення шляхів сполучення і розвиток всіх видів транспорту.

5. Підсилення і дублювання енергетичних потужностей.

6. Розширення зв’язків між галузями промисловості і підприємствами.
7. Створення матеріально-технічних резервів.
8. Підтримання сил ЦЗ в постійній готовності.

Захист робітників і службовців /населення/ досягається трьома основними способами:

1. Застосуванням засобів індивідуального захисту.
2. Укриттям людей в захисних спорудах.
3. Проведенням евакуаційних заходів для робітників і службовців та членів їх сімей.

Засоби індивідуального захисту забезпечують захист людей при знаходженні на виробничих місцях і на місцевості, яка заражена РР, ОР, НХР і БЗ.

Укриття в захисних спорудах - найбільш ефективний спосіб захисту виробничого персоналу працюючої зміни. Захисні споруди повинні будуватися на кожному об'єкті своєчасно і забезпечувати укриття найбільшої працюючої зміни.

Евакуаційні заходи забезпечують захист членів сімей робітників, службовців і виробничого персоналу непрацюючих змін.

Надійність захисту виробничого персоналу досягається застосуванням всіх трьох способів захисту з урахуванням конкретної обстановки.

Захист засобів виробництва полягає в підвищенні опірності /міцності/ будівель, споруд і конструкцій об’єкта до впливу можливих уражаючих чинників і захисту виробничого обладнання, засобів зв'язку та інших засобів, які складають матеріальну основу виробничого процесу.

Забезпечення стійкого постачання досягається проведенням заходів щодо захисту комунально-енергетичних мереж, транспортних комунікацій і джерел постачання, а також створенням необхідних запасів палива, сировини, напівфабрикатів і комплектуючих виробів.

Підготовка до відновлення порушеного виробництва здійснюється своєчасно. Вона передбачає планування відновних робіт по різних варіантах, підготовку ремонтних бригад, створення необхідного запасу матеріалів,

обладнання і направлена на поновлення випуску необхідної продукції в мінімальні терміни.

Підвищення надійності і оперативності управління виробництвом досягається створенням на об'єкті стійкої системи зв'язку, високою професійною підготовкою керівного складу до виконання функціональних обов'язків по керівництву виробництвом і заходами ЦЗ в повсякденній діяльності і в умовах надзвичайних ситуацій, а також своєчасним прийняттям правильних рішень і постановкою задач підлеглим відповідно до обстановки.

Таким чином, підвищення стійкості роботи об'єктів промисловості в умовах надзвичайних ситуацій досягається завчасним проведенням комплексу інженерно-технічних, технологічних і організаційних заходів, спрямованих на максимальне зниження впливу уражаючих чинників і створення умов для ліквідації наслідків надзвичайних ситуацій.

Інженерно-технічні заходи включають комплекс робіт, направлених на підвищення стійкості виробничих будівель, споруд, технологічного обладнання, комунально-енергетичних систем.

Технологічні заходи забезпечують підвищення стійкості об’єкта шляхом спрощення технологічного процесу виробництва кінцевої продукції та виключення або обмеження розвитку аварій.

Організаційні заходи передбачають розробку і планування дій керівного складу, штабу, служб і формувань ЦЗ по захисту робітників і службовців, проведенню рятувальних і невідкладних робіт, відновленню виробництва, а також випуску продукції на обладнанні, що збереглося.

Систематичне виконання норм охорони праці є запорукою передбачення надзвичайних ситуацій.

ВИСНОВОК

В першому розділі розглянуто предметну область розробки веб-застосунків, а саме: типи веб-застосунків, область застосування односторінкових веб-застосунків, розкрито основну проблематику використання SPA, порівняно фреймворки та бібліотеки, наведено ключові відмінності, проведено аналіз необхідного програмного та апаратного забезпечення для розгортання та використання односторінкового застосунку.

В другому розділі проаналізовано методи та алгоритми розробки односторінкових застосунків з використанням бібліотеки React та фреймворку Vue.js. Зокрема розглянуто архітектуру та основні принципи React та Vue.js, проаналізовано компоненти, що створюються на базі Ract та Vue.js, розглянуто стан компонентів, та методи роботи з ним, проведено аналіз найпопулярніших стейт менеджерів, порівняно реактивність Vue та React, розглянуто основні методи життєвого циклу, проаналізовано методи тестування, порівняно масштабованість та продуктивність.

В третьому розділі висвітлено особливості розробки систем односторінкових додатків з використанням React та Vue.js. розглянуто вибір інструментів та процес налаштування робочого середовища, висвітлені способи ініціалізації проектів на базі React та Vue, показано приклади розробки функціональності, а саме: роботу з компонентами, роботу з менеджерами стану, асинхронні запити. Наведено приклад написання юніт тестів, для компонентів на базі Vue та React.

В четвертому розділі було детально розглянуто важливість і необхідність забезпечення охорони праці та безпеки в надзвичайних ситуаціях. Аналіз показав, що дотримання правил та стандартів охорони праці є ключовим для запобігання ризикам, пов'язаним з виробничою діяльністю.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. React The library for web and native user interfaces [Електронний ресурс] – Режим доступу до ресурсу: https://react.dev/.
2. The Progressive JavaScript Framework [Електронний ресурс] – Режим доступу до ресурсу: https://vuejs.org/.
3. Stefanov S. React: Up & Running / Stoyan Stefanov, 2021. – (O\'Reilly Media, Inc.).
4. Macrae C. Vue.js: Up and Running / Callum Macrae, 2018.
5. Adam F. Pro React 16 / Freeman Adam, 2019.
6. Robin W. The Road to React / Wieruch Robin, 2017.
7. Haverbeke M. Eloquent JavaScript: A Modern Introduction to Programming / Marijn Haverbeke, 2018.
8. Wilson G. Software Design by Example: A Tool-Based Introduction with JavaScript / Greg Wilson, 2022. – 339 с.
9. Sekuloski R. JavaScript From Zero to Hero: The Most Complete Guide Ever, Master Modern JavaScript Even If You're New to Programming / Rick Sekuloski., 2022. – 390 с.
10. Cherny B. Programming TypeScript: Making Your JavaScript Applications Scale / Boris Cherny, 2019.
11. Design Patterns: Elements of Reusable Object-Oriented Software / E.Gamma, R. Helm, R. Johnson, J. Vlissides., 1995. – 396 с. – (1).
12. Стручок В. С. БЕЗПЕКА В НАДЗВИЧАЙНИХ СИТУАЦІЯХ / В. С. Стругок. – Тернопіль. – 156 с. – (ФОП Паляниця В. А).
13. М.Р. Петрик, Д.М. Михалик, О.Ю. Петрик, Г.Б. Цуприк. Методичні вказівки до виконання атестаційної роботи магістра за спеціальністю 121 – "Інженерія програмного забезпечення" для усіх форм навчання [Текст] – Тернопіль : Тернопільський національний технічний університет імені Івана Пулюя – 2020 – 27 с.

14. Нікольський Ю.В. Дискретна математика / Ю.В.Нікольський, В.В.Пасічник, Ю.М.Щербина – Львів: Магнолія Плюс, 2007. – 608 с.

15. Інформаційні технології видобутку даних (Data mining, високопродуктивні обчислення у складних системах): навчальний посібник ІВ Бойко, МР Петрик, Г Цуприк - 2020

16. Дискретні структури (Алгебраїчні та числові системи, комбінаторний аналіз) : навчально-методичний посібник для студентів спеціальності 121 «Інженерія програмного забезпечення», аспірантів та викладачів вищих навчальних закладів / Укладач : Бойко І.В., Петрик М.Р., Цуприк Г.Б. – Тернопіль : Тернопільський національний технічний університет імені Івана Пулюя, 2017 – 64 с.

17. Моделювання та видобуток даних (висопродуктивні обчислення у великих алгебраїчних та числових системх, комбінаторному аналізі): навчальний посібник. Тернопіль: : ТНТУ 2019 – 62 с.

18. Д.Р. Пасіка, Г.Б. Цуприк - Збірник тез доповідей VII науково-технічна конференція «Інформаційні моделі, системи та технології». Розробка одностороннього вкб-застосунку з використанням Vue.js і React: порівняльний аналіз продуктивності, користувацького досвіду та доцільності. – Тернопіль 13-14 грудня 2023

ДОДАТКИ

ДОДАТОК А

Апробація результатів роботи

УДК 004.41
Д.Р. Пасіка, Г. Б. Цуприк, канд. техн. наук, доц.
Тернопільський національний технічний університет імені Івана Пулюя, Україна

РОЗРОБКА ОДНОСТОРІНКОВОГО ВЕБ-ЗАСТОСУНКУ З ВИКОРИСТАННЯМ VUE.JS І REACT: ПОРІВНЯЛЬНИЙ АНАЛІЗ ПРОДУКТИВНОСТІБ КОРИСТУВАЦЬКОГО ДОСВІДУ ТА ДОЦІЛЬНОСТІ

D.R. Pasika, H. B. Tsupryk, PhD, Assoc.Prof.

DEVELOPING A SINGLE-PAGE WEB APPLICATION USING VUE.JS AND REACT: A COMPARATIVE ANALYSIS OF USER EXPERIENCE PERFORMANCE AND EXPEDIENCY

Цифрова трансформація все частіше використовується в багатьох сферах життєдіяльності людини: банківська справа, військові розробки, освіта, медицина, торгівля, державні послуги. Одним з світових лідерів цифрової трансформації є і Україна. Серед причин такої тенденції є незліченні переваги цифрових сервісів перед аналоговими, зокрема: ефективність та продуктивність, конкурентоспроможність, зручність користування, аналітика, гнучкість використання. У зв'язку з потребою розробки великої кількості користувацьких інтерфейсів для різного роду застосунків, ринок наповнили багато фреймворків і бібліотек. Класичним і найбільш популярним прикладом такої бібліотеки для веб розробки є React, а фреймворку Vue.js

Односторінковий веб-застосунок – це застосунок який працює в браузері та не вимагає перезавантаження сторінок під час взаємодії з користувачем. Основні переваги які можна отримати від використання такого типу додатків це:

1) швидкодія: сторінка не перезавантажується - це забезпечує швидке реагування на дії користувача і зменшує залежність від якості інтернет з'єднання;
2) динамічність: односторінкові додатки надають змогу оновлювати вміст сторінки без перезавантажень за допомогою AJAX-запитів, що є запорукою плавності та інтерактивної взаємодії з користувачем;
3) управління станом: за рахунок використання фронтенд-бібліотек або фреймворків досягається ефективне керування станом застосунку та компонентів;
4) оптимізація: завдяки використанню провідних методів розробки, досягається хороша оптимізація в наслідок чого застосунок має підтримку великої кількості пристроїв, що в свою чергу збільшує вибірку потенційних користувачів;
5) доступність: використання актуальних технологій веб розробки дає змогу отримати хороший рівень доступності для користувачів з обмеженими можливостями що використовують спеціальні пристрої для взаємодії з застосунком.

Бібліотеки і фреймворки – ключова різниця:

1) бібліотека – колекція функцій та компонентів, що можуть бути використані під час розробки застосунку. Бібліотека надає конкретні інструменти для розв'язання задач пов'язаних з розробкою але не має жорсткого впливу на загальну архітектуру застосунку;
2) фреймворк – шаблон або «скелет», який вказує конкретну структуру застосунку. Безпосередньо впливає на архітектуру додатку, визначаючи як саме застосунок повинен організовуватись та яких правил повинен дотримуватись. В свою чергу фреймворки надають більший рівень

абстракції ніж бібліотеки, та як правило, вже містять певні готові рішення для типових задач.

Використання бібліотек дозволяє розробнику дотримуватись довільної архітектури застосунку, на противагу фреймворки мають свою жорстко визначену архітектуру та вимоги до структури застосунку.

Розробка односторінкового застосунку поділяється на такі етапи:

1) планування та дизайн: на цьому етапі визначається бізнес логіка і вимоги до застосунку, розробляється дизайн додатку;
2) вибір технології: обираються фреймворки, бібліотеки, інструменти, які будуть використовуватися для розробки;
3) розробка структури і компонентів: створення основної структури додатку, компонентів, модулів в наслідок взаємодії яких буде формуватися інтерфейс користувача;
4) робота з API: інтеграція серверного API для реалізації необхідної бізнес логіки;
5) тестування та налагодження додатку: тестування в різних середовищах та з використанням різних пристроїв, перевірка сумісності, написання тестів для забезпечення найкращої якості продукту;
6) підтримка: відбувається після релізу продукту на основі відгуків користувачів та вимог бізнесу.

Розробка односторінкового застосунку вимагає ретельного підходу до вибору технологій та інструментів під конкретні завдання, оскільки саме цей аспект гарантує необхідний рівень функціональності, масштабованості, продуктивності та ефективності готового рішення. В свою чергу досягнення високої якості цих критеріїв дозволяє спростити розробку, підняти продуктивність, покращити досвід користувачів та полегшити подальшу підтримку продукту.

Література

1. Fedosejev A. React.js Essentials. Packt Publishing, 2015.

2. Listwon B., Hanchett E. Vue.js in Action. Manning, 2018.

3. Martin R. C. Clean Code: A Handbook of Agile Software Craftsmanship. Prentice Hall, 2019.

4. Simpson K. You don`t know JS: Up & Going. o'reilly, 2015.

5. М.Р. Петрик, Д.М. Михалик, О.Ю. Петрик, Г.Б. Цуприк. Методичні вказівки до виконання атестаційної роботи магістра за спеціальністю 121 – "Інженерія програмного забезпечення" для усіх форм навчання [Текст] – Тернопіль : Тернопільський національний технічний університет імені Івана Пулюя – 2020 – 27 с.

6. Бойко І.В., М.Р. Петрик, Г.Б. Цуприк. Інформаційні технології видобутку даних (Data mining, високопродуктивні обчислення у складних системах): навчальний посібник. Тернопіль: : ТНТУ 2020 – 62 с.

7. Бойко І.В., М.Р. Петрик, Г.Б. Цуприк Моделювання та видобуток даних (високпродуктивні обчислення у великих алгебраїчних та числових системх, комбінаторному аналізі): навчальний посібник. Тернопіль: : ТНТУ 2019 – 62 с

ДОДАТОК Б

Лістинг коду

src\App.tsx

```
import React from 'react';
import './App.css';
import {RootRouter} from "./routers";
import {Provider} from "react-redux";
import {appStore} from "./store";

const App: React.FC = () => (
  <Provider store={appStore}>
    <RootRouter/>
  </Provider>
)

export default App;
```

src\index.tsx

```
import React from 'react';
import ReactDOM from 'react-dom/client';
import './index.css';
import App from './App';
import reportWebVitals from './reportWebVitals';

const root = ReactDOM.createRoot(
  document.getElementById('root') as HTMLElement
);
root.render(
  <React.StrictMode>
    <App />
  </React.StrictMode>
);

reportWebVitals();
```

src\App.css

```
.App {
  text-align: center;
}
```

```
.App-logo {
  height: 40vmin;
  pointer-events: none;
}

@media (prefers-reduced-motion: no-preference) {
  .App-logo {
    animation: App-logo-spin infinite 20s linear;
  }
}

.App-header {
  background-color: #282c34;
  min-height: 100vh;
  display: flex;
  flex-direction: column;
  align-items: center;
  justify-content: center;
  font-size: calc(10px + 2vmin);
  color: white;
}

.App-link {
  color: #61dafb;
}

@keyframes App-logo-spin {
  from {
    transform: rotate(0deg);
  }
  to {
    transform: rotate(360deg);
  }
}
```

src\index.css

```
body {
  margin: 0;
  font-family: -apple-system, BlinkMacSystemFont, 'Segoe UI',
'Roboto', 'Oxygen',
    'Ubuntu', 'Cantarell', 'Fira Sans', 'Droid Sans', 'Helvetica
Neue',
    sans-serif;
  -webkit-font-smoothing: antialiased;
  -moz-osx-font-smoothing: grayscale;
}

code {
  font-family: source-code-pro, Menlo, Monaco, Consolas, 'Courier
New',
    monospace;
```

```
}
```

src\routers\index.tsx

```
import { useSelector } from "react-redux";
import { AppRouter } from "./AppRouter";
import { AuthRouter } from "./AuthRouter";
import { getSession } from "../store/selectors/session";


export const RootRouter = () => {
  const isAuthenticated = useSelector(getSession)

  return isAuthenticated ? <AppRouter /> : <AuthRouter />
}
```

src\routers\AuthRouter.tsx

```
import { BrowserRouter as Router, Routes, Navigate, Route } from
"react-router-dom";
import { LoginPage } from "../pages/login";
import { Paths } from "../constants/navigation";
import { SingUpPage } from "../pages/singUp";


export const AuthRouter = () => (
  <Router>
    <Routes>
      <Route path={Paths.LOG_IN} element={<LoginPage />} />
      <Route path={Paths.SING_UP} element={<SingUpPage />} />
      <Route path={"/"} element={<Navigate to={Paths.LOG_IN} />} />
    </Routes>
  </Router>
)
```

src\routers\AppRouter.tsx

```
import {BrowserRouter as Router, Routes, Route} from "react-router-
dom";
import {Paths} from "../constants/navigation";
import {HomePage} from "../pages/home";
import {NotFoundPage} from "../pages/notFound";

export const AppRouter = () => (
  <Router>
    <Routes>
      <Route path={Paths.HOME_PAGE} element={<HomePage/>}/>
```

```
      <Route path={'/'} element={<HomePage/>}/>
      <Route path={'*'} element={<NotFoundPage/>}/>
    </Routes>
  </Router>
)
```

src\pages\home\HomePage.tsx

```
import { Box, Container, CssBaseline, Grid } from '@mui/material';
import React, { useEffect } from 'react';
import { NavigationBar } from '../../componnents/navigationBar';
import { WeeklyPerformance } from
'../../componnents/weeklyPerformance';
import { useDispatch, useSelector } from 'react-redux';
import { getWeeklyPerformance } from '../../store/selectors/metrics';
import { getUserAvatarUrl, getUserEmail, getUserId, getUserName } from
'../../store/selectors/session';
import { fetchWeeklyPerformance } from '../../store/actions/metrics';
import { AppDispatch } from '../../store/app/appStore';
import { DailyActivity } from '../../componnents/dailyActivity';
import { getAnnouncements, getTodaysActivities } from
'../../store/selectors/activities';
import { fetchAnnouncment, fetchTodaysActivites } from
'../../store/actions/activities';
import styles from './HomePage.module.css'
import moment from 'moment';
import { UserCard } from '../../componnents/userCard';
import Announcements from
'../../componnents/announcments/Annpouncments';

const HomePage: React.FC = () => {
  const dispatch = useDispatch<AppDispatch>()

  const userId = useSelector(getUserId)
  const fetchWeeklyPerformanceData =
useSelector(getWeeklyPerformance)
  const todaysActivities = useSelector(getTodaysActivities)
  const userAvatarUrl = useSelector(getUserAvatarUrl)
  const userEmail = useSelector(getUserEmail)
  const userName = useSelector(getUserName)
  const announcements = useSelector(getAnnouncements)

  const todaysDate = moment().format('DD.MM.YYYY');

  useEffect(() => {
    dispatch(fetchWeeklyPerformance(userId));
    dispatch(fetchTodaysActivites(userId))
    dispatch(fetchAnnouncment())
  }, [userId, dispatch])
```

```
  return (
    <Box className={styles.mainContainer}>
      <CssBaseline />
      <Container maxWidth="xl">
        <Grid container spacing={2}>
          <Grid item xs={3}>
            <NavigationBar />
          </Grid>
          <Grid item xs={6}>
            <Box sx={{ mb: 2 }}>
              <WeeklyPerformance data={fetchWeeklyPerformanceData} />
            </Box>
            <Box>
              <DailyActivity            activities={todaysActivities}
date={todaysDate} />
            </Box>
          </Grid>
          <Grid item xs={3}>
            <UserCard   avatarUrl={userAvatarUrl}   email={userEmail}
name={userName} />
            <Announcements announcements={announcements} />
          </Grid>
        </Grid>
      </Container>
    </Box>
  );
}

export default HomePage;
```

src\pages\home\HomePage.module.css

```
.mainContainer {
    display: flex;
    background-image: url('../../../public/backgroundImage.svg');
    background-size: cover;
    background-position: center;
    background-repeat: no-repeat;
    height: 100vh;
    width: 100%;
}
```

src\pages\home\index.ts

```
export {default as HomePage} from "./HomePage";
```

src\pages\login\LoginPage.tsx

```
import { Box, Button, TextField, Typography } from '@mui/material';
import React, { useState } from 'react';
import { EMAIL, NOT_REGISTRED_YET, PASSWORD, SIGN_IN, SIGN_UP } from
'../../constants/strings';
import styles from './Login.module.css'
import { Link, redirect } from 'react-router-dom';
import { Paths } from '../../constants/navigation'
import { login } from '../../store/actions/session';
import { AppDispatch } from '../../store/app/appStore';
import { useDispatch } from 'react-redux';

interface LoginFormData {
  email: string;
  password: string;
}

const LoginPage: React.FC = () => {
  const dispatch = useDispatch<AppDispatch>()
  const [formData, setFormData] = useState<LoginFormData>({
    email: '',
    password: '',
  });

  const handleChange = (e: React.ChangeEvent<HTMLInputElement>) => {
    const { name, value } = e.target;
    setFormData({
      ...formData,
      [name]: value,
    });
  };

  const handleSubmit = async (e: React.FormEvent) => {
    e.preventDefault();
    try {
      await dispatch(login(formData))
      redirect(Paths.HOME_PAGE)
    } catch (e) {
      console.error(e)
    }
  };

  return (
    <Box className={styles.mainContainer}>
      <form onSubmit={handleSubmit} className={styles.form}>
        <img src="./kidsLogo.svg" className={styles.logo} />
        <Typography variant="h3" align='center'>
          {SIGN_IN}
        </Typography>

        <TextField
```

```
          id="email"
          label={EMAIL}
          variant="outlined"
          type='text'
          name='email'
          value={formData.email}
          onChange={handleChange}
          required
        />


        <TextField
          id="password"
          label={PASSWORD}
          variant="outlined"
          type='password'
          name='password'
          value={formData.password}
          onChange={handleChange}
          required
        />

        <Button type="submit" variant="contained">{SIGN_IN}</Button>
        <Typography align='center'>
          {NOT_REGISTRED_YET}
          {' '}
          <Link to={Paths.SING_UP}>
            {SIGN_UP}
          </Link>
        </Typography>
      </form>
    </Box>
  );
};

export default LoginPage;
```

src\pages\login\Login.module.css

```
.mainContainer {
    display: flex;
    background-image: url('../../../public/backgroundImage.svg');
    background-size: cover;
    background-position: center;
    background-repeat: no-repeat;
    height: 100vh;
    width: 100%;
}

.form {
```

```
    display: flex;
    flex-direction: column;
    gap: 16px;
    margin: auto;
    background-color: white;
    padding: 20px;
    border-radius: 10px;
}

.logo {
    width: 100%;
}
```

src\pages\login\index.ts

```
export {default as LoginPage} from "./LoginPage";
```

src\pages\notFound\NotFoundPage.tsx

```
import React, {useCallback} from 'react';
import {Box, Button, Typography} from '@mui/material';
import {purple} from '@mui/material/colors';
import {useNavigate} from "react-router-dom";
import {Paths} from "../../constants/navigation";

const primary = purple[500]; // #f44336

const NotFoundPage: React.FC = () => {
  const navigate = useNavigate();

  const handleNavigateHome = useCallback(() => {
    navigate(Paths.HOME_PAGE)
  }, [])

  return (
    <Box
      sx={{
        display: 'flex',
        justifyContent: 'center',
        alignItems: 'center',
        flexDirection: 'column',
        minHeight: '100vh',
      }}
    >
      <Typography variant="h1">
        404
      </Typography>
      <Typography variant="h6">
```

```
        The page you’re looking for doesn’t exist.
      </Typography>
      <Button  variant="contained"  onClick={handleNavigateHome}>Back
Home</Button>
    </Box>
  );
}

export default NotFoundPage
```

src\pages\notFound\index.ts

```
export {default as NotFoundPage} from "./NotFoundPage";
```

src\pages\singUp\SingUp.tsx

```
import   {   Box,   Button,   Checkbox,   FormControlLabel,   TextField,
Typography } from '@mui/material';
import React, { useState } from 'react';
import        {         ALREADY_REGISTERED,         CONFIRM_PASSWORD,
CONFIRM_PERSONAL_INFORMATION, EMAIL, FIRST_NAME_AND_NAME, PASSWORD,
REGISTRATION, SIGN_IN, SIGN_UP } from '../../constants/strings';
import styles from './SingUp.module.css'
import { Link, redirect } from 'react-router-dom';
import { Paths } from '../../constants/navigation'
import { register } from '../../store/actions/session';
import { AppDispatch } from '../../store/app/appStore';
import { useDispatch } from 'react-redux';

interface RegistrationFormData {
  firstNameAndLastName: string;
  email: string;
  password: string;
  confirmPassword: string;
  agreeToTerms: boolean;
}

const SingUpPage: React.FC = () => {
  const dispatch = useDispatch<AppDispatch>()
  const [formData, setFormData] = useState<RegistrationFormData>({
    firstNameAndLastName: '',
    email: '',
    password: '',
    confirmPassword: '',
    agreeToTerms: false,
  });

  const handleChange = (e: React.ChangeEvent<HTMLInputElement>) => {
```

```
    const { name, value, checked, type } = e.target;
    setFormData({
      ...formData,
      [name]: type === 'checkbox' ? checked : value,
    });
  };

  const handleSubmit = async (e: React.FormEvent) => {
    e.preventDefault();
    try {
      await dispatch(register(formData))
      redirect(Paths.HOME_PAGE)
    } catch (e) {
      console.error(e)
    }
  };

  return (
    <Box className={styles.mainContainer}>
      <form onSubmit={handleSubmit} className={styles.form}>
        <img src="./kidsLogo.svg" className={styles.logo} />
        <Typography variant="h3" align='center'>
          {REGISTRATION}
        </Typography>
        <TextField
          id="firstNameAndLastName"
          label={FIRST_NAME_AND_NAME}
          variant="outlined"
          type='text'
          name='firstNameAndLastName'
          onChange={handleChange}
          value={formData.firstNameAndLastName}
          required
        />

        <TextField
          id="email"
          label={EMAIL}
          variant="outlined"
          type='text'
          name='email'
          value={formData.email}
          onChange={handleChange}
          required
        />


        <TextField
          id="password"
          label={PASSWORD}
          variant="outlined"
          type='password'
```

```
          name='password'
          value={formData.password}
          onChange={handleChange}
          required
        />


        <TextField
          id="confirmPassword"
          label={CONFIRM_PASSWORD}
          variant="outlined"
          type='password'
          name='confirmPassword'
          value={formData.confirmPassword}
          onChange={handleChange}
          required
        />

        <FormControlLabel required control={
          <Checkbox       id="agreeToTerms"       name="agreeToTerms"
checked={formData.agreeToTerms} onChange={handleChange} />
        }
          label={CONFIRM_PERSONAL_INFORMATION}
        />

        <Button type="submit" variant="contained">{SIGN_UP}</Button>
        <Typography align='center'>
          {ALREADY_REGISTERED}
          {' '}
          <Link to={Paths.LOG_IN}>
            {SIGN_IN}
          </Link>
        </Typography>
      </form>
    </Box>
  );
};

export default SingUpPage;
```

src\pages\singUp\SingUp.module.css

```
.mainContainer {
    display: flex;
    background-image: url('../../../public/backgroundImage.svg');
    background-size: cover;
    background-position: center;
    background-repeat: no-repeat;
    height: 100vh;
    width: 100%;
```

```
}

.form {
    display: flex;
    flex-direction: column;
    gap: 16px;
    margin: auto;
    background-color: white;
    padding: 20px;
    border-radius: 10px;
}

.logo {
    width: 100%;
}
```

src\pages\singUp\index.ts

```
export {default as SingUpPage} from "./SingUp";
```

src\componnents\announcments\Annpouncments.tsx

```
import React from 'react';
import { Card, CardContent, Typography, List, ListItem, ListItemText,
Divider } from '@mui/material';

interface Announcement {
  date: string;
  content: string;
}

interface AnnouncementsProps {
  announcements: Announcement[];
}

const Announcements: React.FC<AnnouncementsProps> = ({ announcements
}) => {
  return (
    <Card raised sx={{ height: '100%' }}>
      <CardContent>
        <Typography variant="h6" component="div" gutterBottom>
          Оголошення
        </Typography>
        <List>
          {announcements.map((announcement, index) => (
            <React.Fragment key={index}>
              {index > 0 && <Divider variant="inset" component="li"
/>}
```

```
              <ListItem alignItems="flex-start">
                <ListItemText
                  primary={announcement.date}
                  secondary={
                    <Typography
                      sx={{ display: 'inline' }}
                      component="span"
                      variant="body2"
                      color="text.primary"
                    >
                      {announcement.content}
                    </Typography>
                  }
                />
              </ListItem>
            </React.Fragment>
          ))}
        </List>
      </CardContent>
    </Card>
  );
};

export default Announcements;
```

src\componnents\announcments\index.ts

```
export { default as Announcments } from './Annpouncments'
```

src\componnents\dailyActivity\DailyActivity.tsx

```
import React from 'react';
import { Box, Card, CardContent, Typography, Grid, Divider, Rating }
from '@mui/material';

interface Activity {
  subject: string;
  rating: number;
  comment: string;
}

interface DailyActivityProps {
  date: string;
  activities: Activity[];
}

const   DailyActivity:   React.FC<DailyActivityProps>   =   ({   date,
activities }) => {
```

```
  return (
    <Card raised sx={{ height: '100%', display: 'flex', flexDirection: 'column' }}>
      <CardContent sx={{ flexGrow: 1 }}>
        <Typography variant="h5" component="div" gutterBottom>
          Успішність за день
        </Typography>
        <Divider sx={{ my: 2 }} />
        <Typography variant="subtitle1" color="text.secondary" gutterBottom>
          {date}
        </Typography>
        <Grid container spacing={2} direction="column">
          {activities.map((activity, index) => (
            <Grid item key={index}>
              <Box display="flex" justifyContent="space-between" alignItems="center">
                <Typography variant="body1">{activity.subject}</Typography>
                <Rating name={`activity-rating-${index}`} value={activity.rating} readOnly />
              </Box>
              <Typography variant="body2" color="text.secondary">
                {activity.comment}
              </Typography>
            </Grid>
          ))}
        </Grid>
      </CardContent>
    </Card>
  );
};

export default DailyActivity;
```

src\componnents\dailyActivity\index.ts

```
export { default as DailyActivity } from './DailyActivity'
```

src\componnents\navigationBar\NavigationBar.tsx

```
import React from 'react';
import { List, ListItem, ListItemIcon, ListItemText } from '@mui/material';
import HomeIcon from '@mui/icons-material/Home';
import ScheduleIcon from '@mui/icons-material/Schedule';
import MenuBookIcon from '@mui/icons-material/MenuBook';
import LogoutIcon from '@mui/icons-material/Logout';
```

```
import styles from './NavigationBar.module.css'
import { useDispatch } from 'react-redux';
import { resetSession } from '../../store/slices/session';

const NavigationBar: React.FC = () => {
  const dispatch = useDispatch()
  return (
    <List component="nav">
      <ListItem button>
        <ListItemIcon>
          <HomeIcon />
        </ListItemIcon>
        <ListItemText primary="Головна" />
      </ListItem>
      <ListItem button>
        <ListItemIcon>
          <ScheduleIcon />
        </ListItemIcon>
        <ListItemText primary="Розклад" />
      </ListItem>
      <ListItem button>
        <ListItemIcon>
          <MenuBookIcon />
        </ListItemIcon>
        <ListItemText primary="Меню" />
      </ListItem>
      <ListItem button onClick={() => dispatch(resetSession)}>
        <ListItemIcon>
          <LogoutIcon />
        </ListItemIcon>
        <ListItemText primary="Вийти" />
      </ListItem>
    </List>
  );
};

export default NavigationBar;
```

src\componnents\navigationBar\NavigationBar.module.css

```
.mainconatainer {
    background-color: white;
}
```

src\componnents\navigationBar\index.ts

```
export { default as NavigationBar } from './NavigationBar'
```

src\componnents\userCard\UserCard.tsx

```
import React from 'react';
import { Card, CardContent, Typography, Avatar, IconButton } from
'@mui/material';
import EmailIcon from '@mui/icons-material/Email';
import PhoneIcon from '@mui/icons-material/Phone';
import ChatIcon from '@mui/icons-material/Chat';
import { Box } from '@mui/system';

interface UserCardProps {
  name: string;
  email: string;
  avatarUrl: string; // URL аватара користувача
}

const UserCard: React.FC<UserCardProps> = ({ name, email, avatarUrl
}) => {
  return (
    <Card sx={{ display: 'flex', alignItems: 'center', padding: 2 }}>
      <Avatar sx={{ width: 56, height: 56 }} src={avatarUrl} />
      <Box sx={{ marginLeft: 2, flexGrow: 1 }}>
        <Typography variant="subtitle1">{name}</Typography>
        <Typography                                   variant="body2"
color="text.secondary">{email}</Typography>
      </Box>
      <Box>
        <IconButton aria-label="email">
          <EmailIcon />
        </IconButton>
        <IconButton aria-label="phone">
          <PhoneIcon />
        </IconButton>
        <IconButton aria-label="chat">
          <ChatIcon />
        </IconButton>
      </Box>
    </Card>
  );
};

export default UserCard;
```

src\componnents\userCard\index.ts

```
export { default as UserCard } from './UserCard'
```

src\componnents\weeklyPerformance\WeeklyPerformance.tsx

```
import React from 'react';
import { Card, CardContent, Typography } from '@mui/material';
import { Line } from 'react-chartjs-2';
import 'chart.js/auto';

export interface WeeklyPerformanceProps {
  data: {
    id: string;
    labels: string[];
    datasets: {
      label: string;
      data: number[];
      fill: boolean;
    }[];
  };
}

const WeeklyPerformance: React.FC<WeeklyPerformanceProps> = ({ data })
=> {
  const options = {
    scales: {
      y: {
        beginAtZero: true,
      },
    },
  };

  return (
    <Card>
      <CardContent>
        <Typography variant="h5" component="div" gutterBottom>
          Weekly Performance
        </Typography>
        <Line data={data} options={options} />
      </CardContent>
    </Card>
  );
};

export default WeeklyPerformance;
```

```
export { default as WeeklyPerformance } from './WeeklyPerformance'
```

src\store\index.ts

```
import {appStore} from './app/appStore'
import type {RootState} from './app/appStore'
```

```
export {appStore}
export type {RootState}
```

src\store\slices\activities.ts

```
import { createSlice } from '@reduxjs/toolkit'
import { fetchAnnouncment, fetchTodaysActivites } from
'../actions/activities'

interface activity {
  subject: string
  rating: number
  comment: string
}

interface Announcement {
  date: string;
  content: string;
}

interface initialState {
  announcements: Announcement[]
  todaysActivites: activity[]
  loading: boolean
}

const initialState = {
  todaysActivites: [
    {
      subject: 'Гімнастика',
      rating: 3,
      comment: 'Хороша витримка та гнучкість'
    },
    {
      subject: 'Мовлення',
      rating: 5,
      comment: 'Чудове висловлювання думок'
    },
  ],
  announcements: [
    {
      date: '21.11.2023',
      content: 'Шановні батьки! Запрошуємо вас на батьківські збори'
    },
    {
      date: '15 грудня',
      content: 'Благодійний ярмарок'
    },
  ],
```

```
  loading: false,
}

export const metricsSlice = createSlice({
  name: 'activites',
  initialState,
  reducers: {
    resetActivites: (state): void => {
      state = initialState
    },
    [fetchTodaysActivites.pending.type]: (state: initialState) => {
      state.loading = true;
    },
    [fetchTodaysActivites.fulfilled.type]:  (state:  initialState,  {
payload }) => {
      state.loading = false;
      state.todaysActivites = payload.todaysActivites
    },
    [fetchTodaysActivites.rejected.type]: (state: initialState) => {
      state.loading = false;
      state = initialState;
    },
    [fetchAnnouncment.pending.type]: (state: initialState) => {
      state.loading = true;
    },
    [fetchAnnouncment.fulfilled.type]:   (state:   initialState,   {
payload }) => {
      state.loading = false;
      state.announcements = payload.announcements
    },
    [fetchAnnouncment.rejected.type]: (state: initialState) => {
      state.loading = false;
      state = initialState;
    },
  },
})

export const { resetActivites } = metricsSlice.actions

export default metricsSlice.reducer


src\store\slices\metric.ts


import { createSlice } from '@reduxjs/toolkit'
import { fetchWeeklyPerformance } from '../actions/metrics';

interface initialState {
  weeklyPerformance: {
    id: string;
    labels: string[];
```

```
    datasets: {
      label: string;
      data: number[];
      fill: boolean;
    }[]
  },
  loading: boolean,
}

const initialState = {
  weeklyPerformance: {
    id: '',
    labels: [''],
    datasets: [{
      label: '',
      data: [1],
      fill: false,
    }]
  },
  loading: false,
}

export const metricsSlice = createSlice({
  name: 'metrics',
  initialState,
  reducers: {
    resetMetrics: (state): void => {
      state = initialState
    },
    [fetchWeeklyPerformance.pending.type]: (state: initialState) => {
      state.loading = true;
    },
    [fetchWeeklyPerformance.fulfilled.type]: (state: initialState, {
payload }) => {
      state.loading = false;
      state.weeklyPerformance = payload.weeklyPerformance
    },
    [fetchWeeklyPerformance.rejected.type]: (state: initialState) =>
{
      state.loading = false;
      state = initialState;
    },
  },
})

export const { resetMetrics, } = metricsSlice.actions

export default metricsSlice.reducer
```

src\store\slices\session.ts

```
import { createSlice } from '@reduxjs/toolkit'
import { login, register } from '../actions/session'

export interface child {
  childId: string
  name: string
  group: string
}

export interface sessionSlice {
  session: string
  userName: string
  userRole: string
  loading: boolean
  usersChildren: child[]
  userId: string
  avatarUrl: string
  email: string
}

const initialState: sessionSlice = {
  session: '',
  userName: '',
  userRole: '',
  usersChildren: [],
  loading: false,
  userId: '',
  avatarUrl: '',
  email: '',
}

export const sessionSlice = createSlice({
  name: 'session',
  initialState,
  reducers: {
    resetSession: (state): void => {
      state = initialState
    },
    [register.pending.type]: (state: sessionSlice) => {
      state.loading = true;
    },
    [register.fulfilled.type]: (state: sessionSlice, { payload }) =>
{
      state.loading = false;
      state.userName = payload.name;
      state.session = payload.sessionId;
      state.userRole = payload.userRole;
      state.usersChildren = payload.usersChildren;
      state.userId = payload.userId
      state.avatarUrl = payload.avatarUrl
```

```
    },
    [register.rejected.type]: (state: sessionSlice) => {
      state.loading = false;
      state = initialState;
    },
    [login.pending.type]: (state: sessionSlice) => {
      state.loading = true;
    },
    [login.fulfilled.type]: (state: sessionSlice, { payload }) => {
      state.loading = false;
      state.userName = payload.name;
      state.session = payload.sessionId;
      state.userRole = payload.userRole;
      state.usersChildren = payload.usersChildren;
      state.userId = payload.userId
      state.avatarUrl = payload.avatarUrl
    },
    [login.rejected.type]: (state: sessionSlice) => {
      state.loading = false;
      state = initialState;
    },
  },
})

export const { resetSession, setSession } = sessionSlice.actions

export default sessionSlice.reducer
```

src\store\slices\index.ts

```
import sessionReducer from './session'
import metricsReducer from './metric'
import activitesReducer from './activities'

export default { sessionReducer, metricsReducer, activitesReducer }
```

src\store\selectors\activities.ts

```
import { RootState } from "../index";

export const getTodaysActivities = (state: RootState) =>
state.activites.todaysActivites
export const getAnnouncements = (state: RootState) =>
state.activites.announcements
```

src\store\selectors\metrics.ts

```
import { RootState } from "../index";

export const getWeeklyPerformance = (state: RootState) =>
state.metrics.weeklyPerformance
```

src\store\selectors\session.ts

```
import { RootState } from "../index";

export const getUserName = (state: RootState) =>
state.session.userName
export const getUserRole = (state: RootState) =>
state.session.userRole
export const getSession = (state: RootState) => state.session.session
export const getUserId = (state: RootState) => state.session.userId
export const getUserAvatarUrl = (state: RootState) =>
state.session.avatarUrl
export const getUserEmail = (state: RootState) => state.session.email
```

src\store\actions\activities.ts

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

export const fetchTodaysActivites = createAsyncThunk(
  'activites/fetchTodaysActivites',
  async (userId: string, { rejectWithValue }) => {
    try {
      const response = await
fetch(`http://localhost:3001/todaysActivites/${userId}`);
      if (!response.ok) {
        throw new Error('Network response was not ok');
      }
      const data = await response.json();
      return data;
    } catch (error: any) {
      return rejectWithValue(error.message);
    }
  }
);

export const fetchAnnouncment = createAsyncThunk(
  'activites/fetchAnnouncment',
  async (_, { rejectWithValue }) => {
    try {
      const response = await
fetch(`http://localhost:3001/announcment`);
      if (!response.ok) {
        throw new Error('Network response was not ok');
      }
```

```
      const data = await response.json();
      return data;
    } catch (error: any) {
      return rejectWithValue(error.message);
    }
  }
);
```

src\store\actions\session.ts

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

export const register = createAsyncThunk(
  'user/register',
  async (userData: any, { rejectWithValue }) => {
    try {
      const response = await
fetch('http://localhost:3001/auth/registration', {
        method: 'POST',
        headers: {
          'Content-Type': 'application/json',
        },
        body: JSON.stringify(userData)
      });

      if (!response.ok) {
        throw new Error('Server response wasn\'t OK');
      }

      const data = await response.json();
      return data;
    } catch (error: any) {
      return rejectWithValue(error?.message);
    }
  }
);

export const login = createAsyncThunk(
  'user/login',
  async (userData: any, { rejectWithValue }) => {
    try {
      const response = await
fetch('http://localhost:3001/auth/login', {
        method: 'POST',
        headers: {
          'Content-Type': 'application/json',
        },
        body: JSON.stringify(userData)
      });
```

```
      if (!response.ok) {
        throw new Error('Server response wasn\'t OK');
      }

      const data = await response.json();
      return data;
    } catch (error: any) {
      return rejectWithValue(error?.message);
    }
  }
);
```

src\store\actions\metrics.ts

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

export const fetchWeeklyPerformance = createAsyncThunk(
  'metrics/fetchWeeklyPerformance',
  async (userId: string, { rejectWithValue }) => {
    try {
      const  response  =  await  fetch(`http://localhost:3001/weekly-
performance/${userId}`);
      if (!response.ok) {
        throw new Error('Network response was not ok');
      }
      const data = await response.json();
      return data;
    } catch (error: any) {
      return rejectWithValue(error.message);
    }
  }
);
```

src\store\app\appStore.ts

```
import { configureStore } from '@reduxjs/toolkit'
import reducers from "../slices/index"

export const appStore = configureStore({
  reducer: {
    session: reducers.sessionReducer,
    metrics: reducers.metricsReducer,
    activites: reducers.activitesReducer,
  },
})

export type RootState = ReturnType<typeof appStore.getState>
```

```
export type AppDispatch = typeof appStore.dispatch
```

src\constants\strings.js

```
export const REGISTRATION = 'Реєстрація'
export const FIRST_NAME_AND_NAME = 'Прізвище та ім’я'
export const EMAIL = 'email'
export const PASSWORD = 'пароль'
export const CONFIRM_PERSONAL_INFORMATION = 'Погоджуюсь на обробку персональних даних'
export const TO_REGISTER = 'Зареєструватися'
export const ALREADY_REGISTERED = 'Вже зареєстровані?'
export const SIGN_IN = 'Увійти'
export const NOT_REGISTRED_YET = 'Ще не зареєстровані?'
export const SIGN_UP = 'Зареєструйтеся'
export const CONFIRM_PASSWORD = 'Підтвердіть пароль'

export const REVIEW = 'огляд'
export const SCHEDULE = 'розклад'
export const MENU = 'меню'
export const MESSAGE = 'повідомлення'
export const SETTINGS = 'налаштування'
```

src\constants\navigation\navigation.tsx

```
export const LOG_IN = '/login'
export const HOME_PAGE = '/home'
export const SING_UP = '/singup'
```

src\constants\navigation\index.tsx

```
import * as Paths from './navigation'

export {Paths};
```

package.json

```
{
  "name": "kinder-garden-react-v1",
  "version": "0.1.0",
  "private": true,
  "dependencies": {
    "@emotion/styled": "^11.11.0",
```

```
    "@fontsource/roboto": "^5.0.8",
    "@mui/icons-material": "^5.14.18",
    "@mui/material": "^5.14.18",
    "@reduxjs/toolkit": "^1.9.7",
    "@testing-library/jest-dom": "^5.17.0",
    "@testing-library/react": "^13.4.0",
    "@testing-library/user-event": "^13.5.0",
    "@types/jest": "^27.5.2",
    "@types/node": "^16.18.65",
    "@types/react": "^18.2.38",
    "@types/react-dom": "^18.2.17",
    "chart.js": "^4.4.1",
    "moment": "^2.29.4",
    "react": "^18.2.0",
    "react-chartjs-2": "^5.2.0",
    "react-dom": "^18.2.0",
    "react-redux": "^8.1.3",
    "react-router-dom": "^6.20.0",
    "react-scripts": "5.0.1",
    "typescript": "^4.9.5",
    "web-vitals": "^2.1.4"
  },
  "scripts": {
    "start": "react-scripts start",
    "build": "react-scripts build",
    "test": "react-scripts test",
    "eject": "react-scripts eject"
  },
  "eslintConfig": {
    "extends": [
      "react-app",
      "react-app/jest"
    ]
  },
  "browserslist": {
    "production": [
      ">0.2%",
      "not dead",
      "not op_mini all"
    ],
    "development": [
      "last 1 chrome version",
      "last 1 firefox version",
      "last 1 safari version"
    ]
  }
}
```