МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

ПрАТ «ПВНЗ «ЗАПОРІЗЬКИЙ ІНСТИТУТ ЕКОНОМІКИ ТА ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ»

Кафедра інформаційних технологій

ДО ЗАХИСТУ ДОПУЩЕНА

Зав. кафедри _____

д.е.н., доц. С.І. Левицький

# МАГІСТЕРСЬКА ДИПЛОМНА РОБОТА
# ОСОБЛИВОСТІ ВИКОРИСТАННЯ ФУНКЦІОНАЛЬНОГО ПІДХОДУ ДО ПРОГРАМУВАННЯ ВЕБ-ЗАСТОСУНКІВ

Виконав

ст. гр. ІПЗ – 211м _____ А.І. Іжиков

Науковий керівник

доцент _____ О.А. Жеребцов

Запоріжжя

2023 р.

ПРАТ «ПВНЗ «ЗАПОРІЗЬКИЙ ІНСТИТУТ ЕКОНОМІКИ
ТА ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ»
Кафедра інформаційних технологій

ЗАТВЕРДЖУЮ
Зав. кафедрою
д.е.н., доцент
Левицький С.І.
_____
03.10.2022 р.

З А В Д А Н Н Я
НА МАГІСТЕРСЬКУ ДИПЛОМНУ РОБОТУ

студента гр. ІПЗ – 211м, спеціальності 121 «Інженерія програмного забезпечення» ОП «Інженерія програмного забезпечення»
Іжикова Антона Івановича

1.Тема: Особливості використання функціонального підходу до програмування веб-застосунків
затверджена наказом по інституту № 02-16 від 03.10.2022 р.
2. Термін здачі студентом закінченої роботи: 12.01.2023 р.
3. Перелік питань, що підлягають розробці
1. Виконати огляд найбільш популярних парадигм програмування
2. Порівняти функціональну парадигму з ООП з установленням спільних та різних властивостей
3. Здійснити розбір механізмів та можливостей зміни програмної парадигми при інкапсуляції ФП у JavaScript
4. Провести огляд використання функціональної парадигми у роботі з бібліотекою React та сучасними фреймворками
5. Реалізувати практичний перехід класового компоненту у відповідність до ФП

6. Проаналізувати отримані результати

7. Оформити звіт за результатами роботи

4. Календарний графік підготовки бакалаврської дипломної роботи

| № етапу | Зміст | Терміни виконання | Готовність по графіку %, підпис керівника | Підпис керівника про повну готовність етапу, дата |
|---|---|---|---|---|
| 1. | Формулювання теми магістерської дипломної роботи (збір практичного матеріалу за темою магістерської дипломної роботи) | 20.10.22 | | |
| 2. | I атестація<br>I розділ магістерської дипломної роботи | 27.10.22 | | |
| 3. | II атестація<br>II розділ магістерської дипломної роботи | 17.11.22 | | |
| 4. | III атестація<br>III та IV розділ магістерської дипломної роботи, висновки та рекомендації, додатки, реферат, перевірка програмою «Антиплагіат» | 29.12.22 | | |
| 5. | Доопрацювання магістерської дипломної роботи, підготовка презентації, отримання відгуку керівника і рецензії | 10.01.22 | | |
| 6. | Попередній захист магістерської дипломної роботи | 12.01.22 | | |
| 7. | Подача магістерської дипломної роботи на кафедру | за 3 дні до захисту | | |
| 8. | Захист магістерської дипломної роботи | 19.01.22 | | |

Дата видачі завдання: 03.10.2022 р.

Керівник бакалаврської роботи _____ О.А. Жеребцов
(підпис) (прізвище та ініціали)

Завдання отримав до виконання _____ А.І. Іжиков
(підпис студента) (прізвище та ініціали)

# РЕФЕРАТ

Магістерська робота містить 144 сторінки, 7 таблиць, 75 рисунків, один додаток, 15 лістингів, 45 бібліографічних посилань.

Об’єктом дослідження є функціональна парадигма програмування. Предметом дослідження є методи і підходи функціональної парадигми та практичні можливості їх використання у веб-програмуванні.

У першому розділі здійснено детальний огляд предметної області та існуючих парадигм програмування. Виявлено схожості та відмінності функціонального та об’єктно-орієнтованого підходів. Оглянуто використання функціональної парадигми у сучасному веб-програмуванні.

У другому розділі проведено огляд наявних можливостей функціональної парадигми у мові програмування JavaScript. Здійснено аналіз механізмів та можливостей зміни програмної парадигми при інкапсуляції функціонального підходу у JavaScript.

У третьому розділі проведено порівняння та аналіз класових і функціональних компонентів бібліотеки React.js. Здійснено огляд практичного використання найпоширених хуків.

У четвертому розділі оглянуто можливості фреймворку Next.js у порівнянні з React. Проведено практичний детально описаний перехід класових компонентів існуючого веб-додатку до функціонального стилю.

ВЕБ-РОЗРОБКА, ФУНКЦІОНАЛЬНЕ ПРОГРАМУВАННЯ, ФП, JAVASCRIPT, JS, REACT.JS, REACT-HOOKS, NEXT.JS

ЗМІСТ

# ПЕРЕЛІК УМОВНИХ СКОРОЧЕНЬ

| Скорочення | Повна назва | Пояснення/переклад |
|---|---|---|
| БД | База даних | |
| ОЗП | Оперативний Запам’ятовувальний Пристрій | Пам’ять комп’ютера |
| ООП | Об'єктно-орієнтоване програмування | Об'єктно-орієнтована парадигма, або підхід програмування |
| ПЗ | Програмне забезпечення | |
| СУБД | Система управління базами даних | |
| ХОК | Компонент вищого порядку | НОС |
| ЦП | Центральний процесор | |
| ФП | Функціональна парадигма | Функціональний підхід |
| AJAX | Asynchronous Javascript and XML | Технологія взаємодії з сервером без перезавантаження сторінки |
| API | Application Programming Interface | |
| App | Application | Додаток |
| BaaS | Backend-as-a-Service | Бекенд як сервіс |
| CMS | Content management system | Система керування/управління вмістом/контентом |
| CRUD | Create, Read, Update і Delete | Створення, зчитування, оновлення та видалення |
| CSR | Client Side Rendering | Рендерінг на клієнті – це рендерінг програми у браузері за допомогою DOM |
| CSS | Cascading Style Sheets | Каскадні таблиці стилів |
| FCP | First Contentful Paint | Одна з шести метрик, що відображаються в розділі «Продуктивність» звіту Lighthouse |
| DDoS | Distributed Denial of Service | Відмова в обслуговуванні |
| DOM | Document Object Model | Об'єктна модель документа |
| ES(-2015/-6) | ECMAScript | Стандарт мови програмування, затверджений міжнародною організацією ECMA згідно зі специфікацією ECMA-262 |
| FaaS | Firebase-as-a-Service | Firebase як сервіс |
| HOC | High Order Component | Компонент вищого порядку |
| HTML | HyperText Markup Language | Мова розмітки гіпертексту |
| HTTP | Hyper Text Transfer Protocol | Протокол передачі гіпертекстових |

| | | |
|---|---|---|
| | | документів |
| HTTPS | Hyper Text Transfer Protocol Secure | Розширення протоколу HTTP |
| IDE | Integrated development environment | Інтегроване середовище розробки |
| JS | JavaScript | |
| JSON | JavaScript Object Notation | Запис об'єктів JS |
| JSX | JavaScript XML | Розширення синтаксису JS для опису структури інтерфейсу |
| LISP | LISt Processing language | мова обробки списків |
| MIT | Massachusetts Institute of Technology | Массачусетський технологічний інститут |
| ML | Meta Language | Сімейство строгих мов функціонального програмування з розвиненою параметрично-поліморфною системою типів і модулями, що параметризуються |
| MVC | Model View Controller | Модель–представлення–контролер, архітектурний шаблон |
| MVP | Minimum viable product | Мінімально життєздатний продукт |
| npm | Node Package Manager | Менеджер пакунків |
| PaaS | Platform-as-a-Service | Платформа як сервіс |
| PDF | Portable Document Format | |
| PWA | Progressive Web Application | Прогресивний web-додаток |
| SEO | Search Engine Optimization | Пошукова оптимізація |
| SML | Standard ML | Перша самостійна компільована мова в сімействі ML |
| SPA | Simple Page Application | Односторінковий додаток |
| SSG | Static Site Generation | Генерація статичних сайтів |
| SSR | Server Side Rendering | Рендерінг клієнтської частини програми на сервері |
| SQL | Structured query language | Мова структурованих запитів |
| TC39 | Ecma International, Technical Committee 39 - ECMAScript - Ecma TC39 | Комітет, який займається розвитком JavaScript |
| TS | TypeScript | |
| UML | Unified Modeling Language | Уніфікована мова моделювання |
| URL | Uniform Resource Locator | Уніфікований локатор ресурсів або адреса ресурсу |

Магістерська робота спрямована на аналіз та розбір функціональної парадигми та функціональних підходів у веб-розробці. В якості початкових даних використана відкрита інформація про мову програмування JavaScript, бібліотеку React.js, та фреймворк Next.js, що міститься в офіційних джерелах.

У наш час розробка веб-застосунків за допомогою фреймворків є актуальною задачею та має великий попит. Використання різних парадигм у програмуванні може якісно вплинути на процес веб-розробки. Але у складних веб-додатках низькорівневі деталі коду javascript можуть вплинути на загальну продуктивність системи та ускладнити підтримку продукту. Функціональне програмування як підхід до створення кодової бази дозволяє створити загальний погляд на проект, що спрощує розробку та підтримку додатку.

Актуальність даної роботи обґрунтована, насамперед, необхідністю винаходження або створення набору практичних методик для покращення веб-застосунків, поліпшення їх продуктивності, розширюваності, модульності, та надання можливості повторного використання.

Сьогодні, у світі високих технологій, розумних будинків, кібернетизації та космічної експансії, індустрія розваг диктує свої умови технічному сектору промисловості, а інформаційно-комунікативні технології розвиваються експоненційно. Щоб встигати за новими тенденціями та максимально ефективно використовувати можливості апаратної конфігурації, програмування як галузь підлаштовується під загальний потік.

І навпаки, весь світ сьогодні залежить від програм. Кожен новий автомобіль - це міні-суперкомп'ютер на колесах, і помилки в програмному забезпеченні можуть призвести до реальних аварій і коштувати життя реальним людям. Хакери, використовуючи вразливості програмного забезпечення, шпигують за людьми, крадуть дані кредитних карток, використовують обчислювальні ресурси для DDoS атак, підбору паролів і навіть для маніпулювання виборами.

Тож програмісти повинні щосили намагатися виконувати свою роботу краще. Сьогодні, щоб бути затребуваним та корисним у своєму галузевому секторі, важливо розуміти суть сучасних процесів розвитку ІТ. Підходи проектування, послуги, патерни програмування, методології моделювання – ніхто та ніщо не стоїть на місці. Важливо розбиратися та вміти своєчасно та зважено обирати необхідний і дієвий підхід із сотень можливих, і це також обґрунтовує актуальність роботи.

Об'єктом дослідження є функціональна парадигма програмування. Із поняттям об'єкта дослідження нерозривно пов'язане поняття предмета дослідження. Предмет конкретизує об'єкт і дозволяє точно виділити цілі, завдання та тему роботи. Предметом дослідження у цій роботі є методи і підходи функціональної парадигми та практичні можливості їх використання у веб-програмуванні.

Об'єкт дослідження цієї роботи розбирається у публікаціях авторів J. Hughes [1], Хендерсон П. [2], Заяць В.М., Заяць М.М [3], Кайл Симпсон [4]. Автори наголошують на необхідності глибокого розуміння та правильного використання практичних функціональних методик і підходів при інкапсуляції функціональної парадигми у роботі з мультипарадигмовими мовами програмування, а також пропонують свої шляхи досягнення даного завдання.

Об'єкт та предмет дослідження сформульовані виходячи з необхідності одержання практичної можливості проведення аналізу над наявними веб-застосунками задля подальшого покращення таких їх показників, як продуктивність, швидкодія, пошукова оптимізація та захищеність.

Метою даної роботи є аналіз функціональної парадигми у галузі веб-розробки та дослідження механізмів і можливостей зміни програмної парадигми при інкапсуляції ФП у JavaScript задля покращення якості та продуктивності веб-додатків.

Виходячи із мети, завданнями роботи є:

- виконати огляд найбільш популярних парадигм програмування;
- порівняти функціональну парадигму з ООП з установленням спільних та різних властивостей;
- провести аналіз стану та розвитку мови JavaScript з огляду функціональної парадигми програмування;
- здійснити розбір механізмів та можливостей зміни програмної парадигми при інкапсуляції функціонального підходу у JavaScript;
- провести огляд використання функціональної парадигми у роботі з бібліотекою React та сучасними фреймворками;
- виконати практичну імплементацію переводу класового компоненту у відповідність до функціональної парадигми;
- проаналізувати отримані результати та оформити звіт.

Структура кваліфікаційної роботи включає введення, в якому описується об'єкт, конкретизується предмет дослідження та описується постановка комплексної задачі даної роботи, огляд методологічних основ використання функціонального підходу у галузі веб-програмування, теоретичний аналіз механізмів та можливостей зміни програмної парадигми при інкапсуляції функціонального підходу у JavaScript, імплементацію програмного переводу класового компоненту у відповідність до вимог функціонального підходу, висновок та рекомендації щодо використання оглянутих можливостей функціональної парадигми.

Для досягнення поставленої мети використано теоретичні основи щодо методик і підходів функціональної парадигми у веб-розробці. У процесі дослідження використано такі методи та підходи:

- функції першого класу (використання функції як даних, тобто передавання функції як вхідних параметрів, повернення функції та присвоювання функції змінним та властивостям об'єктів);
- функції вищого порядку (функції, які приймають чи повертають інші функції);

- замикання (комбінування функції з її лексичним оточенням);

- карування (перетворення функції на набір функцій з єдиним параметром);

- підхід чистих функцій (чисті функції завжди повертають той самий результат для тих самих параметрів).

Наукову новизну роботи становлять такі результати:

- на основі аналізу існуючих досліджень щодо об’єкта дослідження зроблено порівняння та детально описано схожості і відмінності функціонального підходу з концепцією ООП з урахуванням сучасних вимог до галузі веб-розробки;

- на підставі порівняльного аналізу виявлено та обґрунтовано переваги та недоліки існуючих засобів функціонального підходу у роботі з бібліотекою React;

- на підставі аналізу початкових даних та порівняльного аналізу виявлено та обґрунтовано переваги та недоліки існуючих засобів рендерінгу веб-додатків на стороні клієнта та сервера;

- на підставі розібраної концепції використання функціональної парадигми та отриманих результатів щодо використання різних моделей рендерінгу, формалізовано необхідний набір вимог до сучасного веб-додатку та вироблено набір методик та рекомендацій щодо якісної імплементації функціональної парадигми у веб-програмуванні задля досягнення цільового набору вимог;

- реалізовано детально описане перетворення класового компонента до функціональної парадигми з урахуванням сучасних вимог до розробки веб-додатків, що розширює можливості програми, забезпечує її подальшу підтримку, модернізацію та тестування, та сприяє покращенню пошукової оптимізації, швидкодії та безпеки кінцевого продукту.

Практична цінність роботи заснована на отриманих результатах і полягає у наданні необхідного набору інформації, рекомендацій та прикладів

правильного і якісного використання функціонального підходу у веб-програмуванні, необхідного для покращення існуючих веб-додатків або створення нових з урахуванням сучасних проблем та вимог галузі веб-розробки.

Апробація. Основні положення магістерської роботи доповідалися на міжнародній конференції ICISSE (International Conference on Innovative Solutions in Software Engineering) 30 листопада 2022 року. Результати роботи було опубліковано у збірнику матеріалів конференції [5, 161-169 с.].

13 грудня 2022 року у ході XXIV науково-практичної студентської конференції у Запорізькому Інституті Економіки та Інформаційних Технологій в рамках пленарного засідання проведено доповідь, і результати роботи були опубліковані у збірнику тез конференції [6, 72-73 с.].

# РОЗДІЛ 1
# ФУНКЦІОНАЛЬНЕ ПРОГРАМУВАННЯ ЯК ПАРАДИГМА

## 1.1 Поняття парадигми програмування

Парадигма програмування - це набір концепцій, правил та абстракцій, що визначають стиль програмування. Відповідно до них у кожній парадигмі закладено підхід до використання ключових конструкцій [7]. Наприклад:

- об'єктно-орієнтоване програмування спирається на класи, об'єкти та взаємодію між ними;
- у функціональному програмуванні все тримається на використанні чистих функцій;
- узагальнене програмування становить в основу алгоритми і контейнери, які приймають типи як параметри.

У таблиці 1.1 відображено базові парадигми та їх відмінності.

Таблиця 1.1 – Базові парадигми та їх відмінності

| **Парадигма** | **Ключовий концепт** | **Програма** | **Робота програми** | **Результат** |
|---|---|---|---|---|
| Імперативна | Команда | Послідовність інструкцій | Виконання інструкцій | Підсумковий стан пам'яті |
| Об'єктно-орієнтована | Об'єкт та клас | Набір класів та об'єктів | Обмін даними між об'єктами через виклик їх методів | Підсумковий стан об'єктів |
| Функціональна | Функція | Набір функцій | Обчислення функцій | Підсумкове значення головної функції |
| Логічна | Факти та правила | Логічні співвідношення | Логічний доказ | Результат доказу |

Узагальнивши перевірені практично знання, розробники програмного забезпечення змогли розділити їх у парадигми програмування. Було створено ефективні підходи та рішення майже на всі випадки життя. Стало достатньо класифікувати завдання, щоб дізнатися, яка парадигма, і, відповідно, мова програмування, найкраще підходить для її вирішення.

Парадигми також багато в чому визначають стандарти написання коду та побудови архітектури додатків. Тому розробники, які пишуть різними мовами програмування, але використовують одну й ту саму парадигму, за потреби досить швидко долають «мовний бар'єр».

Парадигми різноманітні у своїх видах. Наприклад, існують: метапрограмування; узагальнене та структурне програмування; компонентно-орієнтовані, прототипно- та агентно-орієнтовані парадигми; аплікативне, доказове, породжуюче програмування, та інші.

Парадигми програмування перехрещуються між собою, доповнюють одна одну, виробляють одна одну, тому їх дуже багато. Іноді в одній програмі використовують декілька схожих парадигм. Тому можна зустріти ситуації, коли різні автори, кажучи про те саме явище, вживають назви з різних парадигм.

## 1.2 Імперативна парадигма програмування

Програмний код в імперативному стилі організовано як послідовність окремих команд, інструкцій, що описують логіку роботи програми. Читаючи такий код, можна зрозуміти, як змінюватиметься стан програми в той чи інший момент — залежно від того, які фрагменти коду буде запущено.

Розробники почали усвідомлено використовувати імперативну парадигму приблизно із середини XX століття. На той момент вже були поширені низькорівневі мови програмування, в яких ця парадигма була

інтуїтивно реалізована. Завдяки інтуїтивній простоті імперативна парадигма набула популярності [8].

Тоді ж почали з'являтися перші високорівневі мови, побудовані на цій парадигмі. Це дозволило парадигмі стати де факто стандартом комерційної розробки на багато років. Послідовниками імперативної парадигми з високорівневих мов можна назвати C/C++, C#, Java, Python, JavaScript, PHP та Ruby. Але це не означає, що ці мови не можуть підтримувати інших парадигм. Імперативна парадигма містить процедурне програмування та ООП.

Переваги:

• вивчати імперативний підхід до розробки програм досить легко. Це особливо важливо, коли людина вперше стикається із програмуванням;

• можна писати код, що легко читається, і налагоджувати його, працюючи над невеликими проектами. У такому коді буде просто розібратися навіть фахівцю, не знайомому з проектом;

• імперативна парадигма є основою багатьох топових мов програмування та має комерційний успіх.

Недоліки:

• слід зазначити, що зі зростанням проекту код стає важко підтримувати і взагалі виникають проблеми з масштабуванням додатків;

• так звані побічні ефекти здатні змінювати стан змінних. Найчастіше розробникам дуже важко передбачити, де та коли виникне така ситуація. Тому потрібно витратити чимало часу, щоб узяти побічні ефекти під контроль;

• у великих проектах через обмеження архітектури розробники змушені писати надлишковий код.

### 1.2.1 Процедурне програмування

Процедурна парадигма це підвид імперативної парадигми. Алгоритм виконання програми також представлено у вигляді послідовності інструкцій, але набори однотипних інструкцій організовано у спеціальні блоки коду, процедури. Їх ще іноді називають підпрограмами, щоб зазначити наявність деякої головної програми, у якій ці процедури створені та описані. Процедури можна викликати багато разів. Їм можна надсилати параметри. Процедуру можна викликати з будь-якої точки головної програми, з іншої процедури чи всередині себе [9].

Безумовно, ця парадигма успадкувала деякі недоліки імперативної парадигми, але завдяки оформленню коду процедур, стає простіше повторно використовувати код, розуміти логіку роботи програми і масштабувати проект. Тому для інженерів середини XX століття поява цієї парадигмі була важливим кроком уперед.

Процедурне програмування раніше використовувалося в таких мовах, як Cobol, Algol, Perl, Fortran, Pascal, Basic та C. Сьогодні процедури так чи інакше реалізовано в більшості сучасних мов, зокрема у JavaScript. Рисунок 1.1 містить приклад JavaScript-коду з умовно-процедурним підходом.

helpers > JS example.js > ...

```
1   const sum = (a, b) => {
2     return a + b;     // процедура повертає суму двох чисел
3   }
4   const multiply = (a, b) => {
5       return a * b;     // процедура повертає множину двох чисел
6   }
7
8   // Основна програма
9   const number1 = 2;
10  const number2 = 4;
11
12  // по черзі викликаємо процедури та зберігаємо результат у змінну result
13  let result  = sum (number1, number2);
14  result  += multiply (number1, number2);
15  console.log( 'result', result );
16
```

Рисунок 1.1 – Приклад процедурного підходу у js

### 1.2.2 Об'єктно-орієнтоване програмування

Об'єктно-орієнтоване програмування (ООП) ще далі просунулося в пізнанні навколишнього світу, розділивши його на об'єкти та класи. У тому числі було переосмислено і процедури [10].

Так, у сімдесятих роках було сформульовано головні ідеї ООП:

• об'єкт — це елементарна сутність, що має властивості (атрибути) та поведінку (методи, вони ж — колишні процедури);

• клас – це тип, шаблон, що визначає структуру, набір атрибутів та методів для об'єктів одного типу – тобто екземплярів класу;

• клас може успадкувати атрибути та методи батьківського класу, та мати при цьому свої власні. Так формується ієрархія класів, вона дозволяє моделювати предметну область різних рівней абстракції і деталізації, вирішуючи завдання частинами;

• поліморфізм — це механізм, що дозволяє використовувати одну й ту саму функцію чи метод для різних типів (класів);

• інкапсуляція – це механізм, що дозволяє приховувати деякі деталі реалізації усередині класу. Часто стороннім сутностям, які працюють з об'єктом ні до чого знати нюанси реалізації його класу та мати доступ до якихось його атрибутів та методів. Тому нерідко розробник створює для класу інтерфейс, який відповідає за взаємодію із зовнішніми сутностями, відкриваючи спеціально обрані для цього атрибути та методи.

Парадигма ООП реалізована як у сучасних, так і у старих мовах. Але це різне ООП. Наприклад, Perl та Fortran – це початково процедурні мови. Їх розробники просто додали до них деякі елементи ООП. Modula-2 і Oberon вирішили обійтися без синтаксичних конструкцій для методів класу. Розробники запропонували власноруч імітувати їх за допомогою звичайних процедур. У Java, C++, Python та JS реалізовано ООП, але у поєднанні з процедурним підходом. І тільки Smalltalk, C# та Ruby – можна назвати «чистими» об'єктно-орієнтованими мовами. Приклад JavaScript-коду в об'єктно-орієнтованому підході наведено на рисунку 1.2


Рисунок 1.2 – Приклад ООП у js

## 1.3 Декларативна парадигма програмування

Декларативна парадигма, на відміну від імперативної, описує не послідовність інструкцій, а проблему (завдання) і модель (набір виразів) на її вирішення. Тобто будь-який припустимий набір вхідних даних буде оброблено відповідно до моделі. При цьому послідовність, в якій ці вирази буде обчислено, не важлива.

Чи зможе програма швидко знайти ймовірне рішення задачі, буде залежати від того, наскільки правильно підібрано набір виразів для моделі, наскільки короткими, «красивими» з точки зору математики виявляться ланцюжки обчислень у процесі пошуку рішення.

Наприклад, потрібно приготувати куряче філе і помідори чері у тушкованому вигляді. Бажано, щоб блюдо було солоне і не пересушене.

Імперативна парадигма визначає рішення так:

1. купити філе курки;
2. купити помідори чері;
3. купити сіль та соняшникову олію;
4. нарізати філе;
5. поставити сковорідку на плиту;
6. закинути у сковорідку філе курки та помідори;
7. налити соняшникову олію та посолити;
8. увімкнути плиту на середньому вогні;
9. через 40 хвилин вимкнути плиту.

Декларативна парадигма:

0. хочу солоне тушковане філе курки з помідорами чері та олією.

До декларативних мов програмування відносять Prolog, LISP, SQL, Haskell, Scala, Erlang, Clojure, Elixir, F#. Часто це вузькоспеціалізовані мови,

що вирішують свої завдання. Декларативна парадигма містить логічне та функціональне програмування [11].

### 1.3.1 Логічне програмування

Реалізація декларативної парадигми у логічному програмуванні будується на двох сутностях – факти та правила виведення [12]. Програма в ході своєї роботи застосовує до заздалегідь відомих фактів описані розробником правила формальної логіки, підтверджуючи або спростовуючи висунуті гіпотези.

Це дає можливість сформулювати завдання у цих термінах, розбити його на підзавдання, якщо потрібно, і рішенням буде загальний результат, зібраний після перевірки гіпотез щодо кожної підзадачі. Як саме проходитиме перевірка гіпотез, які правила будуть у ній задіяні — самостійно вирішує компілятор логічної мови, якою написана програма.

Роботу програми можна порівняти з картою маршрутів. Щоб кудись дістатися не можна йти всіма маршрутами, які намальовані на карті, одразу. На кожному умовному перехресті потрібно вибрати якийсь варіант руху, ухвалюючи рішення на основі відомих фактів (підказок). А розробник при написанні коду задає цю карту, якою можна прийти в різні пункти призначення — залежно від конкретних вхідних даних.

Наприклад, завданням програми є аналіз, які об'єкти є птахом, за заданим набором ознак (фактів):

1) голуб – це птах;
2) у ворони є крила;
3) ворона вміє літати;
4) пінгвін має крила;
5) пінгвін вміє плавати;
6) об'єкт є птахом, якщо він вміє літати і має крила.

Висловлювання 1–5 вважаються фактами, які приймаються без доказів. Вислів під номером 6 є правилом виведення. В даному випадку це правило визначає, у яких випадках об'єкт можна вважати птахом: вміє літати та має крила. Якщо між ними стоїть логічне І, значить повинні бути виконані обидві умови одночасно.

Отримана модель може обробляти, наприклад, такі запити:

• Хто вміє плавати? (Відповідь: пінгвін)

• Хто птах? (Відповіді: голуб, ворона)

• Хто має крила? (Відповіді: ворона, пінгвін)

Програма на логічній псевдомові на кшталт Prolog виглядатиме так:

птах (голуб).

є_крила (ворона).

вміє_літати (ворона).

є_крила (пінгвін).

вміє_плавати (пінгвін).

птах (Об'єкт): - вміє_літати (Об'єкт), є_крила (Об'єкт).

Найвідоміші мови логічного програмування це Prolog, Mercury та Alice. Prolog явили світові у 1970 року та спочатку використовували для аналізу природної мови. Виявилося, що ця мова дозволяє створювати експертні системи з автоматизацією видачі інформації у відповідь на запитання користувачів. Для цього достатньо лише описати факти та придумати для них правила виведення. У середині XX століття логічні мови брали участь у автоматичному доказі теорем.

Наразі логічні мови програмування використовуються рідко і дуже виборче. Проте вони добре зарекомендували себе у розробці трансляторів, оптимізаторів коду та систем штучного інтелекту. Часто в комерційній розробці програмного забезпечення логічна мова використовується у доповнення до імперативної.

## 1.4 Порівняння ФП з ООП з установленням спільних та різних властивостей

Сьогодні вибір парадигми залежить від вимог до програми та мови, якою вона писатиметься, тому кращу виділити вже неможливо. Деякими мовами можна писати і в тому, і в іншому стилі. Будь-який сучасний веб-додаток може активно використовувати мікс із ФП та ООП. Компоненти можуть являти собою функції, структури даних, класи і т.д. Різні мови програмування сприяють використанню різних базових елементів для компонентів. Наприклад, Haskell передбачає використання функцій, Java - класів.

Виходить, що протиставлення ООП та ФП абсолютно штучне. Якщо розглядати проект як сукупність блоків, то зручно використовувати ООП, а для потоку перетворень даних зручніше функціональний підхід. Якщо це не підходить, можна зробити гібрид. Все питання в тому, чи заохочує мова написання в такому стилі.

Коли коду стає багато, його потрібно розбити за процедурами. Коли стає багато процедур, їх потрібно згрупувати за обов'язками і рознести по модулям. Якщо кілька процедур і функцій пов'язані роботою з тими самими даними, їх зручніше згрупувати в об'єкт разом із цими даними. Це ідея об'єктно-орієнтованого підходу. Тобто, ООП - це підхід до компонування коду. Це парадигма, вигадана для моделювання об'єктів реального світу, вона націлена на поділ обов'язків та приховування інформації. Адже якщо взяти купу коду і просто перенести процедури до класів, це не стане об'єктно-орієнтованим кодом. Так само, якщо хтось програмує процедурно, бо не знає патернів ООП, то це не обов'язково буде функціональне програмування.

ФП теж про поділ обов'язків, і теж покликане структурувати кодову базу, але трохи по-іншому: дані → функція1 → дані → функція2 → дані → функція3 → результат. Наприклад, у будь-якій соцмережі, алгоритм пошуку

користувача може виглядати так: профілі → f1 → спільноти профілів → f2 → спільні спільноти → f3 → кількість учасників → f4 → спільноти більше 1000 → f5 → статистика спільнот → f6 → розбір за демографією → f7 → розбір за віком.

Можна додати у фільтри, наприклад, місто, а фільтри з угрупованнями поміняти місцями – неважливо. Об'єкти із методами тут нікуди не впишеш, тому замість об'єктів, що поєднують дані з поведінкою, все рознесено окремо, і самі дані, і їх обробники. Кожен обробник є функцією, що приймає вхідні дані та повертає результат. Тут ідеально підходять асоціативні масиви та інші примітивні структури. Вони не навантажують ОЗП та ЦП створенням тисяч та мільйонів об'єктів для кожного елемента. Але якщо фільтрацій буде багато? Щоб не копіювати мільйони масивів знову і знову, зручніше передавати всі значення за посиланням. Або зробити структури у вигляді класів з полями, щоб усі значення зберігалися в пам'яті в одному екземплярі та передавалися за вказівником.

Чим це відрізняється від імперативного підходу? Розбиття такого коду на процедури і функції в процедурній парадигмі служить як інструмент абстракції. Функції розраховують результат, а процедури його записують. Немає сенсу викликати процедуру, яка нічого не повертає і нічого не робить. У прикладі про пошук у соціальних мережах нічого записувати не треба і імперативна покроковість не потрібна. Одні дані перетворюються на інші в одному потоці, не перезаписуючи старі значення. Тому виходить, що у функціональній парадигмі можна позбавитись процедур та змінних за непотрібністю, і залишити лише константи та функції.

Отже, в ООП основний компонент програми – об'єкт, у функціональному програмуванні – функція. Відомо, що у функціях прописується, яку інформацію вона має отримати як вхідне значення, а яку має віддати. Функції можуть бути вкладеними, коли одна функція є аргументом іншої. Тобто можна зробити висновок, що функціональні

програми, як і об'єктно-орієнтовані, є композицією функцій перетворення даних.

Але об'єкти та класи – не структури даних. Об'єкти можуть використовувати структури даних, та їх деталі реалізації приховані. Ось чому є приватні члени класів. Ззовні доступні лише методи (функції), тому, виявляється, ООП про поведінку, а не про внутрішній стан. Використання об'єктів як структури даних – ознака поганого проектування. Тоді у чому концептуальні відмінності між ООП і ФП? Та чи є різниця між f(o), o.f() і (f o)?

Відома точка зору, що ФП нав'язує дисципліну присвоєння (immutability). У «чистому» функціональному програмуванні немає оператора присвоєння. Термін «змінна» взагалі не застосовується до функціональних мов, тому що одного разу надавши значення, його не можна змінити. Часто вказується, що функції – це об'єкти першого класу. Але у мові Smalltalk функції теж є об'єктами першого класу, та Smalltalk – об'єктно-орієнтована, а не функціональна мова. Чи означає це, що у ФП взагалі немає стану, що змінюється? Ні. У функціональних мов є прийоми, що дозволяють працювати зі станом, що змінюється. Однак, щоб зробити це, доведеться здійснити певну церемонію. Зміна стану виглядає складною, громіздкою і чужорідною у чистому ФП. Це винятковий захід, до якого вдаються лише зрідка і неохоче.

З іншого боку, можна сказати, що ООП нав'язує дисципліну у роботі з вказівниками на функції. ООП пропонує поліморфізм як заміну вказівників на функції. На низькому рівні поліморфізм реалізується за допомогою покажчиків (pointer). Об'єктно-орієнтовані мови просто роблять цю роботу за програміста. І це чудово, тому що працювати з вказівниками на функції безпосередньо дуже незручно: всій команді необхідно дотримуватися складних та незручних угод у кожному випадку. У Java, наприклад, всі

функції віртуальні. Це означає, що всі функції Java викликаються не безпосередньо, а за допомогою вказівників на функції.

## 1.5 Висновки розділу

Сьогодні можна написати об'єктно-орієнтовані функціональні програми, і навпаки, принципи та патерни ООП можуть використовуватись і у функціональних програмах, якщо прийняти дисципліну «вказівників на функції». Але навіщо це «функціональникам»? Які нові переваги це дасть їм? І що можуть отримати об'єктно-орієнтовані програми від незмінності?

У поліморфізму є одна значна перевага, це інверсія вихідного коду та рантайм-залежностей. У більшості систем, коли одна функція викликає іншу, рантайм-залежності та залежності на рівні вихідного коду односпрямовані. Викликаючий модуль залежить від модуля, що викликається. У випадку поліморфізму модуль, що викликає, все ще залежить від викликаного в рантаймі, але вихідний код модуля, що викликає, не залежить від вихідного коду викликаного модуля. Натомість обидва модулі залежать від поліморфного інтерфейсу. Ця інверсія дозволяє модулю поводитися як плагіну. Насправді плагіни так і працюють. Архітектура плагінів дуже надійна, тому що стабільні та важливі бізнес-правила можуть зберігатися окремо від схильних до змін і не настільки важливих правил. Таким чином, для надійності системи слід застосовувати поліморфізм, щоб створити значні архітектурні межі.

Переваги незмінних даних є очевидними – ви не зіткнетеся з проблемами одночасних оновлень, якщо ви ніколи нічого не оновлюєте. Оскільки більшість функціональних МП не пропонує зручного оператора присвоєння, у таких програмах немає значних змін внутрішнього стану. Мутації зарезервовані для специфічних ситуацій. Секції, що містять пряму зміну стану, може бути відокремлено від багатопотокового доступу. Отже,

функціональні програми безпечніші в багатопоточному та багатопроцесорному середовищах.

Подальший огляд використання ФП у веб-програмуванні та аналіз ФП у мові JavaScript визнано доцільною та актуальною задачею.

# РОЗДІЛ 2
# ФП У ВЕБ-ПРОГРАМУВАННІ ТА JAVASCRIPT

## 2.1 Огляд JavaScript як функціональної мови програмування

Сьогодні скриптовий підхід JavaScript виграв у "компонентного" підходу, і Java, Flash і ActiveX розширення вже не підтримуються в більшості браузерів. Єдиною мовою, безпосередньо підтримуваною браузером став JavaScript. Це означає, що браузери менше перевантажені і містять менше помилок, тому що їм потрібно підтримувати лише одну мову. Положення єдиного стандарту мови програмування загального призначення для веб дозволило JavaScript опанувати найбільшу хвилю популярності в історії програмного забезпечення. За багатьма параметрами JavaScript зараз є найпопулярнішою мовою програмування у світі. Це відмінний інструмент для створення великих систем величезними розподіленими командами, де кожна окрема команда може мати свої власні уявлення про те, як писати код. У JavaScript найбільший реєстр ПЗ з відкритим вихідним кодом у світі npm. Справжня сила JavaScript - у різноманітності думок та розробників в екосистемі.

JavaScript був спроектований для використання різними людьми з власним досвідом для різних цілей. Брендан Ейх, автор мови JavaScript, згадує, що 1995 року його взяли до Netscape для створення схеми браузера [13]. Вихідний намір Netscape полягав у підтримці двох різних мов, і скриптова мова, ймовірно, мала нагадувати Scheme, діалект мови Lisp. Хоча пізніше довелося погодитися на вмовляння керівництва, налаштованого отримати продукт «схожий на Java», розробка зберегла ряд важливих властивостей мов функціонального програмування, наприклад, функції першого класу. Її охрестили, нехай і з перебільшенням, «LISP із синтаксисом

C». "Я був найнятий компанією Netscape з обіцянкою "реалізувати Scheme" у браузері. Диктат вищого керівництва полягав у тому, що мова має бути схожою на Java. Це відразу залишило за бортом Perl, Python та Tcl разом зі Scheme." Таким чином, задум Брендана Ейха з самого початку виглядав як Scheme у браузері, який виглядає як Java. "Я зовсім не пишаюся, але щасливий, що вибрав як основні інгредієнти scheme-подібні функції першого класу і self-подібні, хоча і поодинокі, прототипи. Вплив Java, особливо проблема y2k, а також поділ на примітивні типи та об'єкти, був невдалим."

До списку "невдалих" Java-подібних особливостей Ейх також відносить:

- функцію-конструктор та ключове слово new, з семантикою та засобом виклику, відмінним від функцій-конструкторів;
- ключове слово class разом з extends для успадкування від одного з батьків як основний механізм успадкування;
- тенденцію розробників думати про клас як про статичний тип, яким він не є.

Мову JavaScript (технічно ECMAScript) класифікують як скриптову прототипну (підмножина об'єктно-орієнтованої) МП. Окрім прототипної, наявна підтримка імперативної та функціональної парадигм програмування. Мова має динамічну та слабку, типізацію, підтримує функції першого класу, надає можливості керувати браузером. Наразі мова використовується у веб-розробці, серверному програмуванні, прикладному ПЗ, всередині PDF-документів, тощо [14].

У нульових здавалося очевидним, що ключову роль у більшості веб-розробок відіграє сервер, тоді як клієнтські скрипти виконують допоміжні функції. Але окремі програми на Ajax, наприклад, Gmail, продемонстрували, наскільки зручно використовувати програми у вигляді окремої сторінки, що працює в самому браузері, для якої сервер – насамперед централізоване

сховище даних. Швидкі веб-програми, що нагадують звичні стаціонарні програми, виявилися на порядок вище громіздких попередників.

Після випуску Google Chrome у 2008 році і, зокрема, після презентації двигуна V8 від JavaScript, тенденція лише посилилася. Створення надшвидкісного JavaScript стало попередньою умовою розробки повноцінних односторінкових додатків. V8 також зміцнив позиції Node.js, окремій JS платформі, що застосовується в першу чергу для розробки серверних продуктів. Завдяки AngularJS та цілій родині аналогічних розробок, а з ними і Node, інтерес до JS швидко зріс, що зумовило неймовірний попит на цю мову програмування протягом останніх кількох років.

Зараз JavaScript є однією з найпопулярніших мов програмування в Інтернеті, і продовжує розвиватися. Існує TC39, Технічний комітет 39 – це група експертів із JavaScript, які працюють над стандартизацією ECMAScript.

Робочий процес TC39 складається з п’яти етапів:

0) (Ідея): прийом вхідних даних специфікації;

1) (Пропозиція): пропозиція розгляду високорівного API;

2) (Чернетка): точний опис синтаксису та семантики з використанням формальної мови специфікацій;

3) (Кандидат): призначені рецензенти підписують поточний текст специфікації;

4) (Завершення): додаток готовий до включення до формального стандарту ECMAScript.

Після етапу 4 пропозиції додають до наступної редакції ECMAScript. Коли специфікація проходить щорічну ратифікацію як стандарт, пропозиція затверджується. У 2011 році до JS були додані функціональні команди «stalwarts map» та «reduce». У ECMAScript 2015 (ECMAScript 6) з його лаконічним синтаксисом і абсолютно раціональною структурою, з'явився ряд інших функціональних інструментів, включаючи створення об'єктів і синтаксис стрілочних функцій, так звану «жирну стрілку», що полюбилася

розробникам [15]. Не всі функції ES2015 підтримуються браузерами, але завдяки транскомпілятору Babel (найпопулярніший JavaScript компілятор, який використовується для компіляції ES6 в ES5) можна використовувати функціонал ES2015 (і навіть ES2016), а потім компілювати отримані результати у вигляді універсального ES5.

## 2.2 Монади, аплікатори, та функтори

Як доведено, ФП передбачає обгортання у функції практично всього поспіль. Доводиться писати багато маленьких функцій, що багаторазово використовуються. Виклик здійснюється почергово, і виглядає так: (function1.function2.function3), або комбінується і має вигляд типу func1(func2(func3())). Але якщо все можна зробити шляхом комбінування набору функцій, тоді як працювати з null-винятками? Яким чином обробляти умови типу if-else? Як переконатися, переконатися, що функції, які багаторазово використовуються, чисті та дійсно можуть використовуватися скрізь? Та якщо функція потребує декількох значень, то як викликати таку функцію у ланцюжку (chaining)?

Задля вирішення таких питань існують математичні інструменти і рішення на кшталт функтора, аплікатива та монади [16]. Деякі мови, наприклад Haskell, надають підтримку таких рішень «із коробки», та загалом з ними можуть працювати будь-які мови, не лише чисто-функціональні. Існують певні специфікації, дотримання яких надає можливостей функціональності, як у стандартній бібліотеці Haskell. Для мови JS це Fantasy-Land, ця специфікація описує, як повинні виглядати та діяти класи та функції [17]. Загалом, монада повинна підтримувати як мінімум методи map, join, chain та ap. Це дозволяє з'єднувати різні (або навіть однакові) монади в ланцюжок, оскільки всі вони дотримуються єдиного стандарту [18]. На рисунку 2.1 показано приклад залежностей Fantasy-Land.

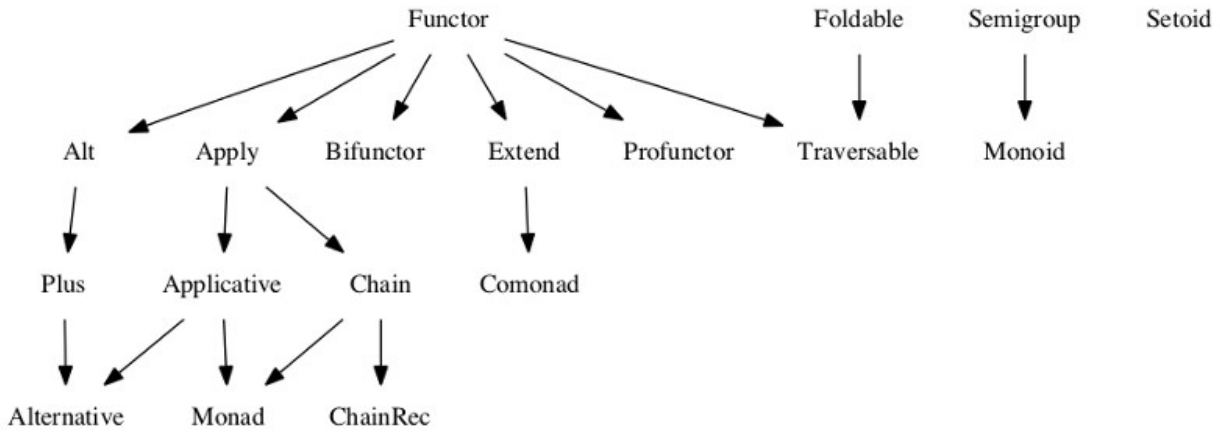



Рисунок 2.1 – Специфікація Fantasy-Land

Haskell підтримує монади на рівні синтаксису у вигляді do-нотації. Це просто синтаксичний цукор. У Haskell вираз «>>=» - це «bind» мовою JavaScript, а «return» - не ключове слово, а функція. Приклад наведено на рисунку 2.2.

```
computation = push 4 >>= \_ ->
              push 5 >>= \_ ->
              pop >>= \a ->
              pop >>= \b ->
              return $ (show a) ++ ":" ++ (show b)
```

Рисунок 2.2 – Синтаксис Haskell

Мона́да згідно до древніх греків, позначала «божество», або «перша істота», «одиниця» або «єдине, як неподільне». Пізніше це стало багатозначним терміном у різних філософських системах сучасності, у програмуванні, психології та езотериці.

У вченні піфагорійців з монади виникла діада, з діади – числа, із чисел – точки, потім двомірні лінії, потім тривимірні сутності, тіла, в яких чотири

основи, чотири елементи, земля, вогонь, повітря і вода, з яких потім було створено решту світу. У Джордано Бруно монада є основною одиницею буття, у діяльності якої зливаються тілесне та духовне, об'єкт та суб'єкт. Найвища субстанція є «монада монад», або Бог.

Але у функціональному програмуванні, як і з більшістю інших речей, академічні визначення монад зрозуміти складніше, ніж сприйняти ідею. Академічною ознакою монади у ФП є ось це: "Монада в безлічі X це моноід з категорії ендофункторів X, в якому морфізм, званий "твор", замінений композицією ендофункторів, а морфізм, званий "одиниця", замінений ендофунктором "тотожність". Через те, що «монада» — це дуже абстрактне поняття, яке не має прямої аналогії з об'єктами реального світу, наукова ознака не дуже легка для розуміння. Іноді монади називають обчисленнями. Але казати, шо «монади обчислюють» було б неправильним. Всі ці ланцюжки дій та значень нічого не обчислюють, поки їм не скажуть це зробити. Тому саме «монади – це обчислення» [19].

Оглянемо основні терміни. Для зручності та легшого сприйняття скористаємось синтаксисом мови TypeScript, яка додасть типізацію динамічному JS та допоможе розглядати інтерфейси як тайпклас [20].

### 2.2.1 Функтор

JS-клас - це функтор (Functor), якщо він працює дотримуючись специфікації та підтримує реалізацію методу Map. Відомий приклад – це array. Він може зберігати значення і має map-метод, який застосовується до значень, що зберігаються. Такий клас дозволяє займатися мапінгом значення всередині контейнера, перетворюючи T<A> в T<B>, де map дозволяє абстрагуватися від структури контейнера, даючи спосіб змінювати вміст контейнера, нічого про цю структуру не знаючи [21]. Абстрактне сприйняття функції map та приклади реалізації функторів мовами C# та JavaScript відображено на рисунках 2.3 – 2.5.

```
public shape Functor<T> where T : <>
{
    static T<B> Map<A, B>(T<A> source, Func<A, B> mapFunc);
}
```

Рисунок 2.3 – Реалізація функтору мовою C#

```
const add1 = ( a ) => a + 1;
class MyFunctor { // кастомний функтор
  constructor( value ) {
    this.val = value;
  }
  map( fn ) { // застосовує функцію до this.val + повертає новий Myfunctor
    return new Myfunctor( fn( this.val ) );
  }
}
let temp = new MyFunctor( 1 ); //temp - це екземпляр функтора, що зберігає значення 1
temp.map( add1 ); //-> temp дозволяє нам перетворити (map) "add1"
```

Рисунок 2.4 – Реалізація функтору мовою JavaScript

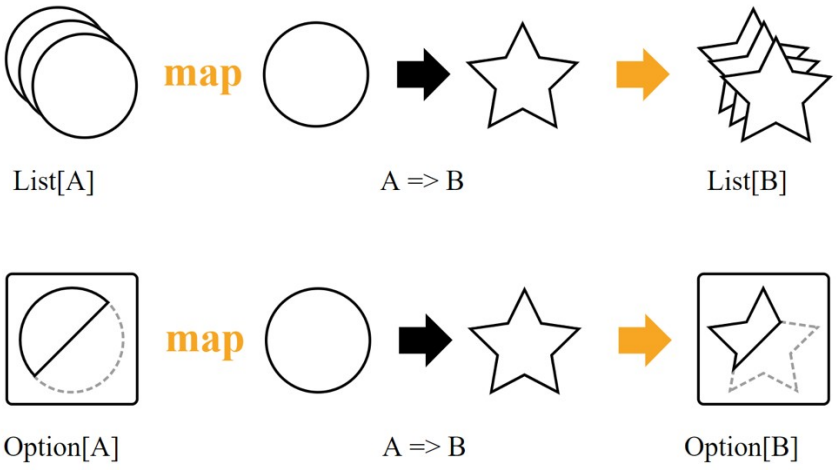



Рисунок 2.5 – Абстрактний опис роботи методу map

### 2.2.2 Аплікативний функтор

Аплікативний функтор, або аплікатив – це будь-який клас, що реалізує пару функцій Pure та LiftA2, для яких виконуються їх прості правила,

переважно пов'язані з композицією. "Аплікативний" означає, що з ним можливо застосовувати упаковані функції до упакованих значень. Такі класи можуть використовуватися у функціях, які працюють з null-значеннями як з лівого боку рівняння, так і з правого. Монади Maybe (як і всі монади) теж реалізують специфікацію ap, а значить вони теж аплікативні, а не просто монади. Приклад аплікативу мовою C# наведено у рисунку 2.6.

```
public shape Applicative<T> where T : <>
{
    static T<A> Pure<A>(A a);
    static T<C> LiftA2<A, B, C>(T<A> ta, T<B> tb, Func<A,B,C> map2);
}
```

Рисунок 2.6 – Аплікативний функтор мовою C#

Pure дозволяє створити контейнер, що містить єдине значення, LiftA2 дозволяє використовувати функцію від двох аргументів, маючи на руках два контейнери з відповідними типами, упакованими всередині. Назва LiftA2 походить з того, що обчислення як би "піднімаються" над двома голими змінними A і B у обчислення над аплікативами T<A> і T<B> відповідно. Абстрактний приклад роботи методів наведено на рисунках 2.7 та 2.8.

Реалізація цих методів гарантує автоматичну реалізацію тайпкласу "Functor". Цей клас дозволяє комбінувати разом пару незалежних обчислень T<A> і T<B> у загальний T<C>. Адже якщо є два значення T<A> і T<B> і функція, що перетворює пару значень A, B в C, то можна отримати T<C>.

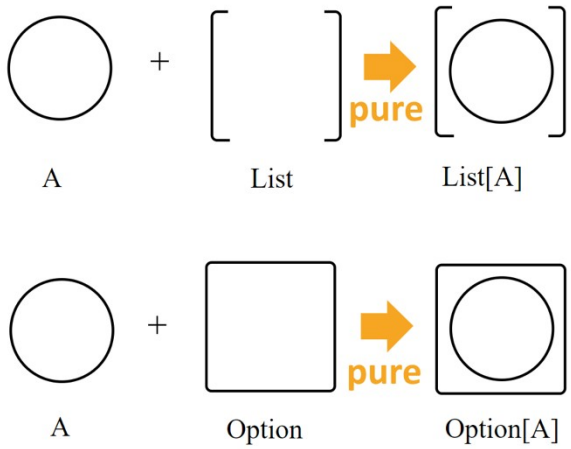



Рисунок 2.7 – Схема роботи методу Pure

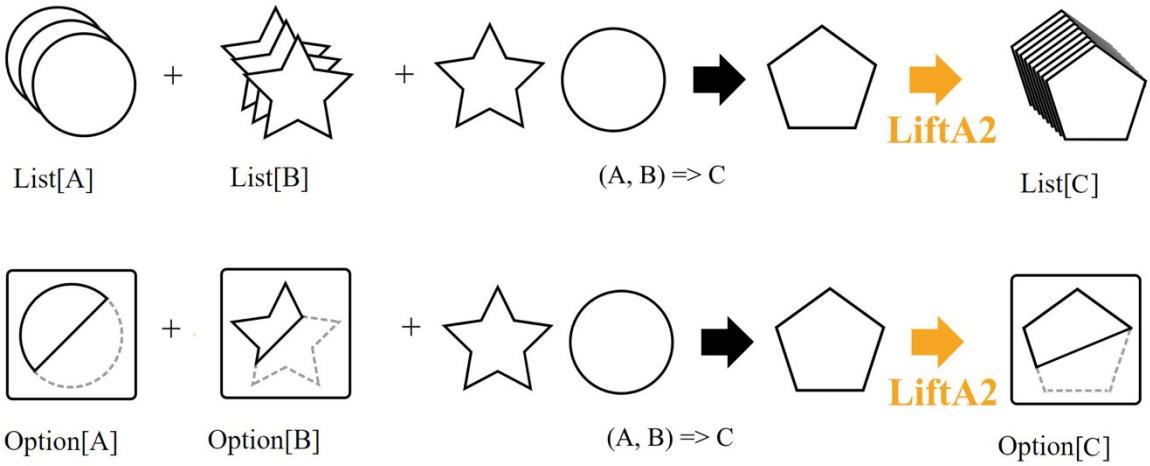



Рисунок 2.8 – Схема роботи методу LiftA2

### 2.2.3 Монада

Монади – це теж функтори, тому що в них є метод map. Але вони реалізують не лише його. Схема залежностей з рисунку 2.1 показує, що монади повинні реалізовувати різні функції з різних специфікацій, наприклад Functor, Apply, Applicative, Chain і самої Monad. Залежність може бути успадкованою, але необов'язково. Наприклад, монада реалізує обидві

специфікації - Applicative і Chain. Загальний приклад реалізації монади мовою JavaScript показано на рисунку 2.9.

```
class Monad {
  constructor( val ) {
    this.__value = val;
  }
  static of( val ) { // Monad.of простіше, ніж new Monad(val)
    return new Monad( val );
  }
  map( f ) { // застосовує функцію, але повертає другу монаду
    return Monad.of( f( this.__value ) );
  }
  join() { // використовується для отримання значення з монади
    return this.__value;
  }
  chain( f ) { // допоміжна функція, що перетворює (map), а потім витягує значення
    return this.map( f ).join();
  }
  ap( someOtherMonad ) { // використовується для роботи з кількома монадами
    return someOtherMonad.map( this.__value );
  }
}
```

Рисунок 2.9 – Монада мовою JavaScript

Монада – це будь-який клас з функціями Pure і Bind, який приймає аргумент типу T<A>  і функцію, що перетворює розпаковане значення A в T<B>, і повертає значення того ж типу T<B>. Ідея сигнатури проста: є упаковане в контейнер значення типу A, є функція перетворення з голого A у такий самий контейнер, але вже зі значенням B. Функція Bind дозволяє "зв'язати" ці два вирази разом, отримавши з пари (T<A>, A => T<B>) значення T<B>. Реалізація цих методів гарантує автоматичну реалізацію тайпкласу "Applicative". Клас дозволяє комбінувати залежні обчислення, де T залежить від A, який у свою чергу знаходиться в контейнері того ж типу T<A>.

```
function Maybe( x ) { // <-- головний конструктор, що повертає монаду Maybe підкласу Just або Nothing
  return x == null? _nothing : Maybe.Just( x );
}
function Just( x ) {
  this.value = x;
}
util.extend( Just, Maybe );
Just.prototype.isJust = true;
Just.prototype.isNothing = false;
function Nothing() {}
util.extend( Nothing, Maybe );
Nothing.prototype.isNothing = true;
Nothing.prototype.isJust = false;
var _nothing = new Nothing();
Maybe.Nothing = function() {
  return _nothing;
};
Maybe.Just = function( x ) {
  return new Just( x );
};
Maybe.of = Maybe.Just;
Maybe.prototype.of = Maybe.Just;
// Функтор
Just.prototype.map = function( f ) { // <-- робить map, коли Just виконує функцію,
  return this.of( f( this.value ) ); // і повертає Just на підставі результату
};
Nothing.prototype.map = util.returnThis; // <-- робить map, коли Nothing нічого не робить
Just.prototype.getOrElse = function() {
  return this.value;
};
Nothing.prototype.getOrElse = function( a ) {
  return a;
};
module.exports = Maybe;
```

Рисунок 2.10 – Часткова реалізація монади Maybe з ramda-fantasy

Рисунок 2.10 демонструє фрагменти реалізації монади Maybe з бібліотеки ramda-fantasy, а на рисунку 2.11 зображено реалізацію монади мовою C#.

```
public shape Monad<T> where T : <>
{
    static T<A> Pure<A>(A a);
    static T<B> Bind<A, B>(T<A> ta, Func<A, T<B>> mapInner);
}
```

Рисунок 2.11 – Монада мовою C#

Все це може виглядати малокорисним при виконанні простих операцій над атомарним стеком, але це дуже корисно при виконанні кількох асинхронних операцій, залежних один від одного. Наприклад, при проектуванні, бібліотеки, яка комбінує парсери із контекстно-залежною граматикою. Це дозволяє автору бібліотеки надавати лише кілька примітивних функцій для монади парсера, а потім користувач бібліотеки може змішувати ці примітиви, як йому потрібно.

## 2.3 Огляд механізмів та можливостей зміни програмної парадигми при інкапсуляції ФП у JavaScript

У екосистемі JS і без монад чи аплікаторів JS є чимало прийомів функціонального програмування: композиція, часткове застосування, карування, чисті функції, та багато інших методик та підходів. Одні можливості стають базою для інших, більш складних. Завдяки функціям першого класу стають можливими функції вищого порядку, які, в скою чергу, уможливлюють замикання. А через замикання з’являється мемоізація. Оглянемо найбільш застосовні суттєвості та розберемо все складне простою мовою зі зрозумілими прикладами.

### 2.3.1 Прості поняття та визначення

Функція – поняття, близьке до математичного. Вона щось одержує на вхід і завжди щось повертає. Процедура ж викликається заради побічних ефектів.

Параметри – це змінні, створені в оголошенні функції. Аргументи – конкретні значення, передані під час виклику. На рисунку 2.12 зображено приклади. F – це функція, x – параметр, print – це процедура, тому що викликається лише для того, щоб вивести в консоль свої аргументи помаранчевим кольором і розділити їх символом нового рядка. Але у

функціональному програмуванні намагаються якнайбільше використовувати функції, які явно щось повертають. Те, що зазвичай вважається процедурою, тут є функцією без return, тому умовно у JS немає процедур. Навіть якщо опустити return, функція все одно неявно повертає undefined і залишається функцією.

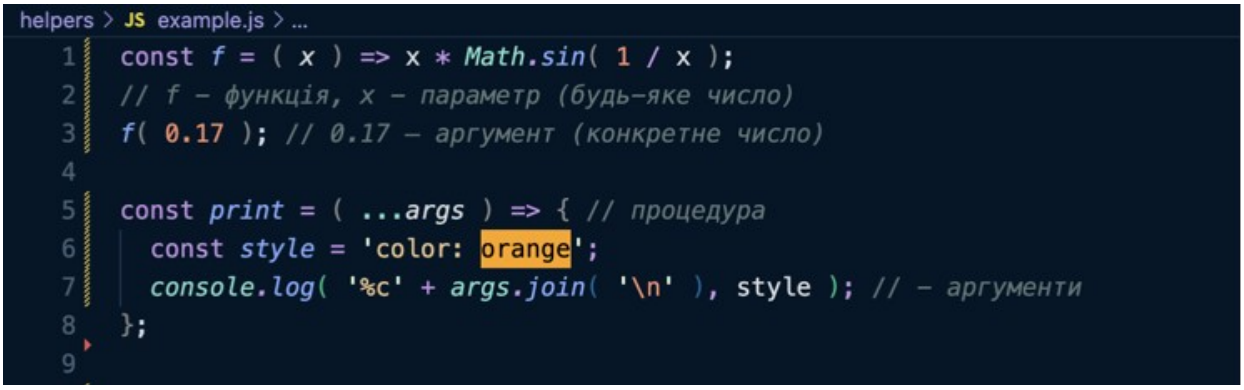



Рисунок 2.12 – Функція, процедура, параметр та аргумент

Сигнатура – це кількість, тип та порядок параметрів. Оголошення функції JS не містить інформації про тип параметрів через динамічну типізацію. Якщо не використовується TypeScript, цю інформацію можна вказати через JSDoc, як зображено на рисунку 2.13 [22].

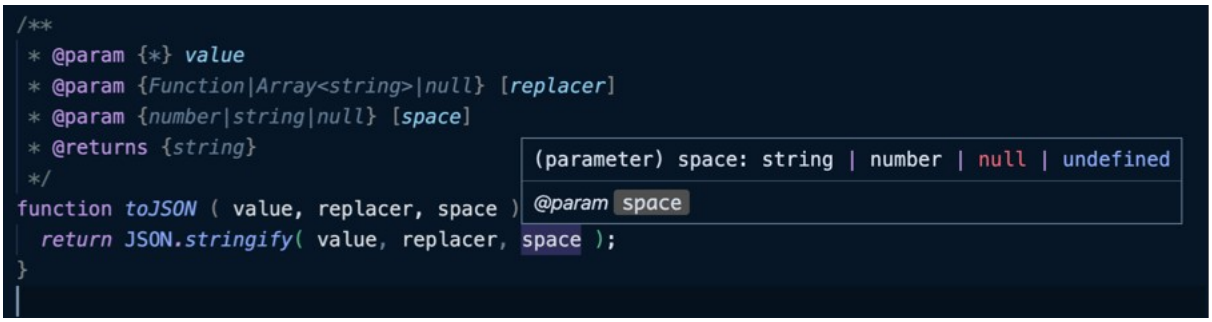



Рисунок 2.13 – Оголошення сигнатури функції у JSDoc

Арність – кількість параметрів, які приймає функція. У JavaScript арність функції можна визначити за допомогою властивості length. На

рисунку 2.14 зображено практичне використання властивості length у різних ситуаціях, та повернуте значення.

```
const awesome = ( good, better, theBest ) => {};
awesome.length; // 3

const defaultParams = ( answer = 42 ) => {}; // аргументи за замовчанням
defaultParams.length; // 0

const restParams = ( ...args ) => {}; // остаточні параметри
restParams.length; // 0

const destructuring = ( { target } ) => {}; // деструктуризація
destructuring.length; // 1
```

Рисунок 2.14 – Особливості властивості length

Предикат – це функція, яка повертає логічне значення. Найпоширеніший приклад - використання предикату всередині функцій filter, some, every та інших. Приклад наведено у рисунку 2.15.

```
const array = [4, 8, 15, 16, 23, 42];
const isEven = ( x ) => x % 2 === 0; // isEven – це предикат
const even = array.filter( isEven );
```

Рисунок 2.15 – Предикат isEven

### 2.3.2 Рекурсія, замикання, функції першого класу та вищого порядку

Рекурсія – це здатність функції викликати саму себе. У деяких функціональних мовах рекурсія є єдиною можливістю виконати цикл, бо у таких мовах немає конструкцій на кшталт for, while чи do...while. Коли функція викликає сама себе, відбувається рекурсивний виклик. Для його коректної роботи необхідно, щоб усередині функції була хоча б одна рекурсивна умова, на яку виконання циклу програми обов'язково рано чи пізно вийде. Якщо цього не станеться, програма зациклиться. Проблема в

тому, що у разі рекурсії з дуже великою глибиною може статися переповнення стеку. Це можна було б виправити за допомогою хвостової рекурсії. Тоді кожен наступний рекурсивний виклик буде заміщений в поточному стеку. Щоб хвостова рекурсія стала можливою, необхідно щоб функція не використовувала замикання і явно повертала рекурсивний виклик як останню операцію. Хвостова рекурсія дозволяє оптимізувати виклики компілятором і вже є в стандарті ES6, але незважаючи на привабливі можливості, підтримка браузерами досі бажає кращого. І кожен новий виклик створює новий кадр у стеку, незважаючи на те, що умови рекурсії виконуються [23]. Приклади наведено у рисунку 2.16.

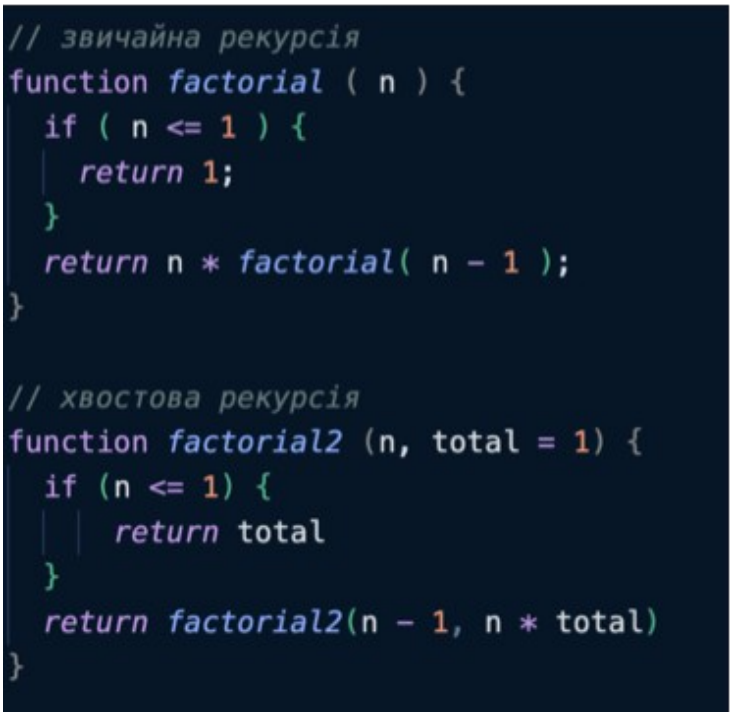



Рисунок 2.16 – Приклади звичайної та хвостової рекурсії

Функції першого класу. Це використання функції як даних, тобто передавання функції як вхідних параметрів, повернення функції та присвоювання функції змінним та властивостям об'єктів, як зображено на рисунку 2.17. Ця властивість уможливлює існування функцій вищого

порядку, що, у свою чергу, уможливлює появу часткового застосування, карування та композиції. Це один з важливіших кроків мови у функціональному напрямку.

```
// присвоєння
const assign = () => {}
// передавання
const passFn = (fn) => fn()
// повернення
const returnFn = () => () => {}
```

Рисунок 2.17 – Присвоєння, передавання та повернення функції

Функції вищого порядку – це функції, які приймають чи повертають інші функції. Наступний функціональний крок. Сьогодні js-розробники працюють з ними щодня. При цьому вищим порядком можуть бути вже не тільки функції, а й, наприклад, компоненти React, що приймають або повертають інші компоненти. Вони відповідно називаються компонентами вищого порядку. Приклади наведено на рисунку 2.18.

```
// map, filter, reduce и т.д.
[0, NaN, Infinity].filter(Boolean)

// обіцянки (проміси)
new Promise((res) => setTimeout(res, 300))

// обробники подій
document.addEventListener('keydown', ({code, key}) => {
    console.log(code, key)
})
```

Рисунок 2.18 – Приклади функцій вищого порядку

Замикання – це комбінація функції та її лексичного оточення. Замикання створюється заново щоразу під час виклику функції та дозволяє отримати значення змінних, оголошених у зовнішній функції. Коли функція створюється всередині іншої функції, то вона має доступ до змінних, оголошених у зовнішній функції, навіть після того, як буде здійснено повернення цієї зовнішньої функції. Замикання це те, що дозволяє працювати фіксованим аргументам часткових застосувань. Фіксований аргумент - це аргумент, заданий у контексті замикання функції що повертається. Наприклад, у виразі «add(1)(2)» аргумент 2 є фіксованим аргументом для функції, що повертається під час виклику add(1). Приклад замикання у js зображено на рисунку 2.19. Тут всередині замикання зберігаються дві змінні: tag і count. Кожного разу, коли всередині іншої функції створюється та повертається нова змінна, функція находить змінну, оголошену у зовнішній функції, через замикання [24]. Ще один приклад корисного використання замикання, - функція мемоізації, яка кешує результати свого виклику, - зображено на рисунку 2.20.

```
const createCounter = tag => count => ({
  inc () { ++count },
  dec () { --count },
  val () {
      console.log(`${tag}: ${count}`)
  }
})

const pomoCounter = createCounter('pomo')

const work = pomoCounter(0)
work.inc()
work.val() // pomo: 1

const rest = pomoCounter(4)
rest.dec()
rest.val() // pomo: 3
```

Рисунок 2.19 – Приклад замикання у js

```
const memo = (fn, cache = new Map) => param => {
  if (!cache.has(param)) {
    cache.set(param, fn(param))
  }
  return cache.get(param)
}

const f = memo((x) => x * Math.sin(1 / x))
f(0.314) // обчислити
f(0.314) // взяти з кешу
```

Рисунок 2.20 – Приклад мемоізації js

### 2.3.3 Конвеєр, композиція, часткове застосування та карування

Конвеєр і композиція — це наступний виклик функції з результатами попередньої. В залежності від того, в яку сторону передано дані: зліва направо або справо наліво, отримується або конвейер, або композиція. Конвеєри частіш за все зустрічаються при роботі в командному рядку, або у виклику скриптів при запуску додатків, як показано на рисунку 2.21.

```
"scripts": {
  "dev": "rollup -w -c --no-treeshake",
  "test": "jest",
  "coverage": "npm run test -- --coverage",
  "build": "npm run build:js && npm run build:types",
  "build:js": "rollup -c",
  "build:types": "tsc --project tsconfig.json"
},
```

Рисунок 2.21 – Скриптові команди з package.json

Можливо, в JavaScript теж з'явиться щось схоже. В одній з пропозицій для ESNext описується конвеєрний оператор |>. Це синтаксичний цукор, що використовується у форматі «вираз |> функція». Він створює ланцюгові

виклики функцій у легкому для сприйняття вигляді, як показано у рисунку 2.22. Наразі |> перебуває у стадії 1 TC39.

```
const double = (n) => n * 2
const increment = (n) => n + 1

// без конвеєрного оператора
double(increment(double(double(5)))) // 42
// з конвеєрним оператором
5 |> double |> double |> increment |> double // 42
```

Рисунок 2.22 – Конвеєрний оператор

Якщо запустити конвеєр у зворотному напрямку, вийде композиція. Композицію функцій можна створити без операторів, просто викликаючи кожну наступну функцію з результатами попередньої, що зображено на рисунку 2.23. Зовні все залишилося майже так само, але місце виклику функції increment змінилося, тому що тепер ланцюжок обчислень почав працювати справа наліво: 42 <- 21 <- 20 <- 10 <- 5.

```
const double = ( n ) => n * 2;
const increment = ( n ) => n + 1;

// композиція функцій в чистому вигляді
double( increment( double( double( 5 ) ) ) ); // 42
// те саме через допоміжну функцію compose
compose( double, increment, double, double )( 5 );
```

Рисунок 2.23 – Приклад композиції

Композиція визначає більш природний порядок виклику функцій. З умовою, що назви функцій свідчать про черговість їх виконання, оригінальний ланцюжок викликів виглядав би так: three( two( one( x ) ) ).

Розгорнутий перебіг виконання можна виразити так:

const a = one(x); => const b = two(a); => const c1 = three (b).

Аналог без композиції чи конвеєра: const c2 = three (two (one (x))).

Якщо розмістити всі три функції в рядок, та результат виклику передати у наступну функцію, такий виклик був би найбільш природним з погляду читання: const c3 = pipe(one, two, three)(x).

Таким чином, конвеєр та композиція – це два напрями одного потоку даних. Але це не просто шаблон для організації потоку обчислень, а й фабрика з виробництва нових деталей. Коли нова функція створюється шляхом обертання іншої функції, передачі даних із зовнішньої функції у внутрішню потрібно один чи кілька параметрів. Коли використовується композиція, необхідність у цьому відпадає, тому що результат однієї функції передається далі по ланцюжку.

Насправді не так багато випадків, де можна застосувати композицію. Крім того, застосовність обмежена відсутністю в JS вбудованих механізмів: потрібно використовувати бібліотеки або самостійно реалізовувати необхідні функції. Але композиція входить, наприклад, до складу бібліотеки Redux. На проекті з Redux композиція буде використовуватися для middleware, тому що createStore приймає лише один підсилювач (enhancer), а їх, як правило, потрібно хоча б кілька. Приклад композиції Redux наведено на рисунку 2.24. Тут DevTools додаються до застосування middleware, щоб можна було коректно дебагувати асинхронний код. Інший кейс, де може стати в нагоді композиція - фільтрація або перетворення потоку даних, що зображено на рисунку 2.25.

```
// композиція в redux
const store = createStore(
    reducer,
    compose(
        applyMiddleware(...middleware),
        DevTools.instrument(),
    )
)
```

Рисунок 2.24 – Приклад композиції Redux

```
const notifications = [
    { text: 'Warning!', lang: 'en', closed: true },
    { text: 'Увага!', lang: 'ua', closed: false },
    { text: 'Attention!', lang: 'en', closed: false }
]
// добре
notifications.filter((notification) => {
    // ...перевірити всі умови
})
// краще
notifications
    .filter(isOpen)
    .filter(isLang)
// the best
notifications.filter(compose(
    isLang,
    isOpen,
))
```

Рисунок 2.25 – Композиція у фільтрації

Функція повертає одне значення, отже, всередині конвеєра чи композиції можна передати далі лише один аргумент. Але що робити, якщо функція визначена з кількома параметрами, необхідними для роботи? Тут на допомогу приходить часткове застосування та каріювання, або карування

(curry). Вони перетворюють функцію з вихідним набором параметрів на іншу функцію з меншим числом параметрів, але роблять це по-різному. Часткове застосування перетворює функцію на одну функцію з меншим числом параметрів. Карування перетворює функцію на набір функцій з єдиним параметром. Незважаючи на те, що в класичному розумінні карування перетворює функцію на набір функцій з єдиним параметром, на практиці реалізації карування можуть приймати кілька аргументів за один раз. Приклади наведено у рисунку 2.26, а на рисунку 2.27 зображено реалізацію конвеєра та композиції за допомогою оглянутих методів.

```
const sum = ( x, y, z ) =>
  console.log( x + y + z );

// часткове застосування
const partialSum = partial( sum, 8 );
partialSum( 13, 21 ); // 42

// карування
const curriedSum = curry( sum );
curriedSum( 8 )( 13 )( 21 ); // 42
curriedSum( 8, 13 )( 21 ); // 42
curriedSum( 8, 13, 21 ); // 42
```

Рисунок 2.26 – Приклади часткового застосування та карування

```
// Реалізувати конвеєр можна було б так:
function pipe (...fns) {
    return (x) => fns.reduce((v, f) => f(v), x)
}
// Щоб реалізувати композицію, достатньо замінити reduce на
reduceRight:
function compose (...fns) {
    return (x) => fns.reduceRight((v, f) => f(v), x)
}
```

Рисунок 2.27 – Конвеєр та композиція

```
const sum = ( x, y, z ) => console.log( x + y + z );

// часткове застосування
const partialSum = partial( sum, 42 );
partialSum(); // NaN, тому що 42 + undefined + undefined

// карування
const curriedSum = curry(sum)
curriedSum( 8 ); // Нова функція – sum(8)
curriedSum( 8 )( 13 ); // Ще одна нова функція – sum(8, 13)
curriedSum( 8 )( 13 )( 21 ); // 42, тому що набралося потрібної кількості аргументів
```

Рисунок 2.28 – Різниця між частковим застосуванням та каруванням

Коли передаються аргументи у кількості меншій, ніж арність функції, при частковому застосуванні відбувається виклик функції. У той час як карування повертатиме нові функції доти, доки не набереться достатньо аргументів. Це відображено на рисунку 2.28.

Часткове застосування та карування чутливі до порядку даних. Існує два підходи до порядку оголошення параметрів. Спершу ітерація, потім дані (iterate-first, data-last): const translate => (lang, text) => /* */. Або спершу дані, потім ітерація (data-first, iterate-last): const translate => (text, lang) => /* */.

Крім застосування в композиції для налаштування сигнатури функції, часткове застосування та каріювання мають іншу корисну особливість. З їхньою допомогою можна виконувати функції більш специфічно. Приклад наведено на рисунку 2.29. У цьому випадку так само, як при створенні деталі через композицію, можна створити нову, але цього разу специфічнішу деталь за допомогою карування або часткового застосування.

```
const fetchApi = (baseUrl, path) =>
    fetch(`${baseUrl}${path}`)
        .then(res => res.json())
// часткове застосування
function partial (fn, ...apply) {
  return (...args) => fn(...apply, ...args)
}
const fetchUnsplash = partial(fetchApi, 'https://api.unsplash.com')
const fetchRandomPhoto = partial(fetchUnsplash, '/photos/random')
// каріювання
function curry (fn) {
  return (...args) => args.length >= fn.length?
      fn(...args) : curry(fn.bind(null, ...args))
}
const fetchCurry = curry(fetchApi)
const fetchUnsplash = fetchCurry('https://api.unsplash.com')
const fetchRandomPhoto = fetchUnsplash(fetchApi, '/photos/random')
// або з використанням bind
const fetchUnsplashBinded = fetchApi.bind(null, 'https://api.unsplash.com')
const fetchRandomPhotoBinded = fetchUnsplash.bind(null, '/photos/random')
```

Рисунок 2.29 – Використання часткового застосування та карування

## 2.4 Практичне використання, проблеми та рішення JavaScript у ФП

У деяких функціональних мовах "чистота" підтримується за замовчанням, а вирази із побічними ефектами неприпустимі. У JavaScript "чистота" може бути досягнута лише за згодою команди, тобто всі учасники повинні домовитися використовувати лише чисті функції. Якщо для більшої частини програми не використовується композиція чистих функцій, то це вважається недотриманням функціонального стилю.

### 2.4.1 Мутації та незмінність

Мутація – це зміна значення структури даних без створення нової змінної та переприсвоєння значення. Деякі ФП-мови блокують таку поведінку. Замість мутацій існуючих значень структур даних, типу об'єктів і масивів, вирази поміщають результат у нові структури даних. В JavaScript

речі, які здаються незмінними, насправді такими не є. Можливо ненавмисно мутувати масив або об'єкт. Оголошення через const захищає від змін лише посилання, а сам об'єкт залишається відкритим для мутацій. Приклад мутації наведено у рисунку 2.30.

```
// мутація об'єкта
const object = {} // const означає константне посилання
object = {} // TypeError: Assignment to constant variable
object.value = 42 // але сам об'єкт можна безперешкодно змінювати

// ненавмисна мутація масиву
function sortArray (array) {
  return array.sort()
}
const fruits = ['orange', 'pineapple', 'apple']
const sorted = sortArray(fruits)
// вихідний масив також змінився
console.log(fruits) // ['apple', 'orange', 'pineapple']
console.log(sorted) // ['apple', 'orange', 'pineapple']
```

Рисунок 2.30 – Мутації об'єкту і масиву

Незмінні чи іммутабельні дані стійкі до змін (мутацій). Щоразу, коли в даних потрібно щось змінити, створюється копія, а вихідні дані залишаються без змін. Такий підхід допомагає уникнути помилок. До переваг незмінних структур даних відносять передбачувану зміну стану, швидке порівняння за посиланням, кешування, та легкість у тестуванні. Але у незмінних структур є два великі недоліки: треба пам'ятати про те, що дані треба копіювати, коли це необхідно, і, відповідно, з'являються витрати на копіювання.

Об'єкти та масиви завжди передаються за посиланням. Під час надання або передачі параметра відбувається копіювання посилання, але не самих даних. Навіть якщо застосувати засоби метапрограмування і, наприклад, заморозити об'єкт, у цьому випадку все одно можливо змінити вкладені об'єкти за посиланням. Замість заморозки можна скористатися Proxy API, але в цьому випадку також доведеться додатково обробляти вкладені структури, тому що вони все ще відкриті для змін. Приклад наведено на рисунку 2.31.

```
// ---------------
const array = []
const ref = array // копія посилання
ref.push('apple')
const append = (ref) => ref.push('orange')
append(array) // ще одна копія посилання
console.log(array) // масив двічі мутований через посилання: [ 'apple', 'orange' ]
// ---------------
const object = { val: 42, ref: {} }
const frozen = Object.freeze(object)
// frozen.val = 23 <- не спрацює, TypeError: Cannot assign to read only property...
frozen.ref.boom = 'woops' // а мутування вкладених даних за посиланням - спрацює
console.log(frozen) // { val: 42, ref: { boom: 'woops' }
// ---------------
const proxy = new Proxy(object, {
    set () { return true },
    deleteProperty () { return true }
})
proxy.val = 19 // зміна або видалення властивості не спрацює
delete proxy.val
proxy.newProp = 23 // так само, як і додавання нового
proxy.object.boom = 'woops' // але вкладені об'єкти все ще мутабельні
console.log(proxy) // { val: 42, ref: { boom: 'woops' } }
```

Рисунок 2.31 – Мутації вкладених об'єктів за посиланням

З копіюванням даних теж все не просто. У більшості випадків працює копіювання масивів та об'єктів вбудованими засобами JavaScript. На жаль, у цьому випадку створюється поверхнева копія, тому позбавлення від мутацій можливе тільки доти, доки відсутні інші вкладені об'єкти. Така ж історія з функціональними методами масивів – map та filter створюють поверхневу копію вихідного масиву. Тому для створення повноцінної копії потрібна вбудована функція глибокого копіювання, яка вимагатиме додаткових витрат [25]. Мутацію пов’язаних об'єкта та масиву зображено на рисунку 2.32.

```
const object = { val: 42, ref: {} }
const copy = {...object}
copy.val = 23
copy.ref.boom = 'woops'
console.log(object) // { val: 42, ref: { boom: 'woops' }
// ———————————————
const array = [null, 42, {}]
const copyArr = array.filter(Boolean)
copyArr[0] = 23
copyArr[1].boom = 'woops'
console.log(array) // [ null, 42, { boom: 'woops' } ]
console.log(copyArr) // [ 23, { boom: 'woops' }
```

Рисунок 2.32 – Мутації пов’язаних даних

### 2.4.2 Незмінні структури даних (persistent data structures)

Отже, з незмінністю у JavaScript все складно. Але можна обійти існуючі обмеження за допомогою спеціальних структур даних. Якщо взяти бібліотеку, яка реалізовує незмінні структури, то по-перше, буде набагато складніше випадково мутувати дані, тому що бібліотека щоразу самостійно створює копії. І по-друге, стануть доступними різного роду оптимізації для ефективнішого копіювання даних. Як, наприклад, копіювання під час запису, коли під час читання даних використовується загальна копія, а у разі зміни створюється новий об'єкт. Дві найпопулярніші бібліотеки цього напрямку у світі фронтенд-розробки – це Immutable.js та Immer.js. Immer заморожує всі об'єкти, які повертає виробництво, щоб захистити розробника від можливих ненавмисних мутацій. Приклад роботи наведено на рисунку 2.33.

```
import produce from 'immer';
const object = {ref: {data: {}}};
const immutable = produce(object, (draft) => {
    draft.ref.boom = 'woops';
});
console.log(object) // { ref: { data: {} }
console.log(immutable) // {ref: {data: {}, boom: 'woops'}
// Оптимізація через копіювання під час запису
// копія data не створювалася, т.к. об'єкт не змінювався
console.log(object.ref.data === immutable.ref.data) // true
```

Рисунок 2.33 – Immer.js в дії

Слід пам'ятати про мінливу природу типів надісланих даних, і точно знати, які методи мутують, а які ні. Іноді деструктуризації буде достатньо, але коли надійність та незмінність важливіша за швидкість чи простоту розробки, незмінні структури даних підійдуть набагато краще. На рисунках 2.34 та 2.35 наведено приклади створення редюсера Redux шляхом деструктуризації та з використанням бібліотеки immer.js.

```
const addTodo = ( state = initState, action ) => {
  switch ( action.type ) {
  case ADD_TODO: {
    return {
      ...state,
      todos: [...state.todos, action.todo]
    };
  }
  default: {
    return state;
  }
  }
};
```

Рисунок 2.34 – Створення стору з використанням деструктуризації

```
// або
import produce from 'immer';
const addTodoImmer = ( state = initState, action ) =>
  produce( state, ( draft ) => {
    switch ( action.type ) {
    case ADD_TODO: {
      draft.todos.push( action.todo );
      break;
    }
    default: {return;}
    }
  } );
```

Рисунок 2.35 – Створення стору з використанням бібліотеки Immer.js

### 2.4.3 Побічні ефекти та чисті функції

Побічними ефектами називається будь-яка взаємодія із зовнішнім світом через операції вводу/виводу та мутація даних. Робота з глобальними змінними – теж побічний ефект. Це може бути логування, запис у файл, запит на сервер, тощо.

Функції без побічних ефектів, які залежать тільки від вхідних параметрів і для тих самих аргументів завжди повертають той самий результат, називають чистими. Якщо функція звертається до глобальної змінної або отримує дані через операцію зчитування даних ззовні, вона втрачає свою чистоту. Вони можуть викликати інші чисті функції, але якщо всередині ланцюжка викликів потрапить хоча б одна функція, яка не відповідає умовам, чистим перестає бути весь ланцюжок.

Побічних ефектів не вдасться позбутися повністю, але їх можна винести за межі функції, зробивши саму функцію чистою. Тоді вона прийматиме дані через параметри. Мутація даних усередині функції - ще один різновид побічних ефектів. Функція, яка мутує дані, ніби залишає слід у вигляді змін після виклику. Складність у тому, що багато вбудованих функцій JS за замовчанням мутують дані. Якщо про це забути, можна випадково залишити після виклику функції слід із побічних ефектів. Приклади побічних ефектів наведено у рисунку 2.36.

```
function impureFunction() {
  console.log( 'side effects' ); // логування
  window.addEventListener( 'scroll', () => {} );  // установка обробника
  fetch( '/analytics/pixel' ).then().catch( () => {  // запит на сервер
    app.state.hasError = true; // глобальна змінна
  } );
  if ( NODE_ENV === 'development' ) { // глобальна змінна
    const { value } = document.querySelector( '.email' );  // читання даних з DOM
    document.getElementById( 'menu' ).hidden = true; // модифікація DOM
    const id = localStorage.getItem( 'sessionId' );  // звернення до локального сховища
    localStorage.setItem( 'status', 'ok' );  // запис у локальне сховище
    const text = fs.readFileSync( 'file.txt', 'utf8' ); // читання з файлу
    fs.writeFileSync( 'log.txt', `${new Date}\n`, 'utf8' ); // запис у файл
  }
}
function impureObj ( o ) {
  return Object.defineProperty( o, 'mark', {
    value: true,
    enumerable: true,
  } );
}
const object = {};
const marked = impure( object );
console.log( object ); // { mark: true }, Object.defineProperty мутувала вихідний об'єкт
```

Рисунок 2.36 – Приклади побічних ефектів

Зовнішні залежності можна замінити відносно до параметрів. Чисті функції завжди повертають той самий результат для тих самих параметрів. Як тільки з'являється непередбачуваність, функція втрачає чистоту. Простий приклад непередбачуваного результату – робота з випадковістю. Щоб зробити функцію чистою, достатньо винести невизначеність за межі функції та передати її як параметр, що показано у рисунку 2.37. Тоді функція завжди повертатиме той самий результат для тих самих параметрів.

```
function impure (min, max) {
    return Math.floor(Math.random() * (max - min + 1) + min)
}
impure(1, 10) // 4
impure(1, 10) // 2

function pure (min, max, random = Math.random()) {
    return Math.floor(random * (max - min + 1) + min)
}
pure (1, 10, 0.42) // 5
pure (1, 10, 0.42) // 5
```

Рисунок 2.37 – Приклади чистої функції у роботі з випадковими значеннями

Крім того, чисті функції мають прозорість посилань. Це ефект, який дозволяє замість виклику функції підставити результат роботи, як показано на рисунку 2.38.

```
const refTransparency = () => Math.pow(2, 5) + Math.sqrt(100)
refTransparency() // виклик функції
// або
Math.pow(2, 5) + Math.sqrt(100) // можна розкрити і зрозуміти результат
32 + 10 // 42
```

Рисунок 2.38 – Прозорість посилань у чистих функціях

Переваги чистих функцій: простіше розібратися, як влаштована функція, їх можна запросто кешувати, легко тестувати, легко розпаралелювати.

То чому все не написати на чистих функціях? Строго кажучи, чистота відносна. Наприклад, функція з математичним PI не чиста, тому що залежить від глобальної змінної. Але ж навряд чи комусь заманеться змінювати значення PI, тому не варто виводити чистоту за межі сенсу. Та якщо написати програму тільки з чистих функцій, то вийде занадто абсолютна чистота. Код виглядатиме ось так: «(() => {})()». Програма без побічних ефектів непотрібна, адже така програма нічого не робить.

## 2.5 Висновки розділу

У даному розділі проаналізовано розвиток та становлення мови JavaScript. Наведено особливості сучасного використання функціональної парадигми, оглянуто механізми та можливості зміни програмної парадигми при інкапсуляції ФП у JavaScript. Було доведено, що JavaScript – мультипарадигмова мова програмування, тобто універсальна мова, що дозволяє використовувати кілька парадигм та програмувати у різних стилях.

У JavaScript відсутні повністю незмінні структури даних, але є ознаки, що в майбутніх версіях специфікації ECMAScript очікується їх поява. Додавання const в ES6 є однією з таких ознак.

В останніх версіях JS з'явилися зручні та корисні підходи, такі як оператори нульового злиття, зведення у ступінь, опціональної послідовності чи конвеєрний оператор («**», «??», «?.» та «|>»). Наприклад, завдяки опціональним ланцюжкам (object?.param) розробникам відтепер значно рідше доведеться писати щось на кшталт «object && object.prop1 && object.prop1.prop2». Завдяки методу .bind, JavaScript реалізує часткове застосування з коробки. Дві основні можливості часткового застосування та карування: налаштування функцій для реалізації композиції та спеціалізація. Також JavaScript від ФП має анонімні функції та лямбда-синтаксис. Запис виду «x=>x*2» є валідним виразом JavaScript. Такий синтаксис значно полегшує роботу з функціями вищого порядку і замиканнями, та дозволяє працювати з чистими функціями.

Таким чином, мова JavaScript відповідає сучасним вимогам функціональної парадигми, та має багато корисних методів та засобів для досягнення високих показників якості коду, швидкодії додатку та зручності розробки ПЗ. Подальший розбір ФП у сучасних js-фреймворках є доцільною та актуальною задачею.

# РОЗДІЛ 3
# ФУНКЦІОНАЛЬНЕ ПРОГРАМУВАННЯ У REACT

## 3.1 Бібліотека React з точки зору функціонального програмування

React.js — відкрита JavaScript бібліотека для імплементації інтерфейсів користувача, яка створена для вирішення проблеми часткового оновлення вмісту веб-сторінки, з якими найчастіше стикалися в розробці односторінкових застосунків [26]. З появою React багато конкуруючих фреймворків швидко перейшли на React-way. Наприклад, в версії 1.5 AngularJS директиви стали компонентами, а у документацію було додано нормальний опис методів життєвого циклу. Після виходу React нові UI фреймворки почали переходити на односпрямований потік даних, використовувати інтерфейс як функцію стану. Односпрямоване зв'язування даних дозволило вирішити цілу категорію проблем із синхронізацією даних та продуктивністю.

### 3.1.1 Наявні функціональні особливості React

Оглянемо найважливіші підходи функціонального програмування, які представлено в React.

Функції можуть бути присвоєні іншим змінним, таким, як інші типи даних, параметри або значення що повертаються. Ця властивість є у функцій JS, а як було доведено, все, що підтримує JS, підтримує функціональне перетворення. Реалізація функцій зворотного виклику та замикань у js використовує цю можливість.

Декларативне оголошення функції, а потім виклик функції для обчислення. Кожна частина має значення що повертається, яке обчислюється безпосередньо.

Чиста функція без побічних ефектів. Тобто коли у функцію передаються однакові параметри, результат буде однаковим. Завдяки цьому функція може бути вільною від зовнішніх впливів і не торкатися зовнішнього середовища, тобто бути повністю незалежною ззовні. Таким чином, виклик функції не матиме різних проблем, викликаних змінами у зовнішньому середовищі.

Компонент React – це генерація функції, а компонент вищого порядку в React – це функція вищого порядку. Наприклад, функція connect з бібліотек redux [27] та react-router [28]. Фактично, це функція вищого порядку, яку розміщено поза додатком. Принцип той же, тобто дочірній компонент передається в батьківський, а повертається вже новий компонент. Логіка батьківського компоненту може бути повторно використана. Суть процесу рендерінгу сторінки React – це вкладений процес виклику функцій.

Компонент передається батьківським компонентом крізь props. Це дає контроль, але субкомпоненти не можуть бути змінені безпосередньо всередині props, тобто це повинен бути єдиний потік даних, який у чомусь схожий на характеристики чистої функції та не змінює зовнішній стан. Тому при розробці компонентів потрібно передавати операції з побічними ефектами на зовнішній контроль, такі компоненти незалежні і легко адаптуються.

Функціональне програмування має функцію карування, та виклик комбінації функцій. Багато методів викликів curry та compose також використовуються в React та Redux, наприклад: const middleware = store => next => action => {…}. З цієї точки зору react має характеристики функціонального програмування, а redux - це модуль що підключається, який суворо дотримується функціонального підходу.

### 3.1.2 Відомі проблеми, незручності та труднощі

React з'явився в 2013 році і весь час команда розробників намагається забезпечувати зворотну сумісність нових «фіч» так довго, як це можливо. І це одна з причин, через яку було розроблено величезну кодову базу на основі React. Але забезпечення зворотної сумісності має свою ціну: останнім часом документація та багато ресурсів спільноти можуть бути застарілими та вводити в оману. При пошуку рішень можна знайти занедбані npm-пакети зі старим синтаксисом. Через те, що старий синтаксис та класи не приваблюють нових учасників (contributors), такі пакети мають багато відкритих питань і низьку активність розробки.

Офіційна документація React все ще рекомендує використовувати componentDidMount і componentWillUnmount, а не useEffect. Команда розробників React працює над новою версією документації під назвою Beta docs протягом останніх років. Зрештою, довгий перехід на хуки досі не завершено, що призводить до фрагментації спільноти. Нові розробники щосили намагаються знайти свій шлях в екосистемі React, а старі намагаються йти у ногу із часом [29].

До неприємностей сьогодні відносять і належність реакту до спільноти Meta. Спочатку Facebook виглядав дуже круто, вони хотіли «зблизити людей» та розвивали спільноту. Але потім були скандали щодо участі Facebook у схемі маніпулювання натовпом, вони винайшли концепцію «фейкових новин» [30], почали збирати про своїх користувачів інформацію без їхньої згоди [31]. Звісно, діти не відповідають за вчинки батьків, але React досі залежить від Facebook, та якщо стан справ Meta стане зовсім поганим, React може постраждати разом з ними.

## 3.2 Класові та функціональні компоненти React

Існує думка, що функціональні компоненти React відрізняються від компонентів що базуються на класах, кращою продуктивністю. Але більшість бенчмарків, якими це перевіряють, мають недоліки, та навіть самі розробники React рекомендують робити висновки, ґрунтуючись на результатах таких перевірок, з великою обережністю. Продуктивність, насамперед, залежить від того, що відбувається в коді, а не від того, чи обрані для реалізації можливостей функціональні, або засновані на класах компоненти. Більшість досліджень показують, що різниця у продуктивності між двома видами компонентів незначна. Проте слід зазначити, що стратегії оптимізації, що застосовуються при роботі з ними, відрізняються.

Бенчмаркінг буде оглянуто далі, але спочатку потрібно розібратися у головних відмінностях між функціональними та класовими компонентами React. Це відмінності у ментальній моделі використання таких компонентів, і вони полягають в тому, що функціональні компоненти захоплюють вже відрендеровані значення.

### 3.2.1 Особливості та відмінності компонентів, заснованих на функціях та класах

На рисунку 3.1 зображено один компонент у функціональному та класовому вигляді. Він виводить кнопку, натискання на яку імітує, за допомогою функції setTimeout, виконання мережевого запиту, та виводить вікно повідомлення з підтвердженням виконання операції. Неважливо, чи використовуються тут стрілочні функції або оголошення функцій. Конструкція виду «function handleClick()» працюватиме так само. Наприклад, якщо у props.user зберігається значення «Sophie», то у вікні повідомлення, через три секунди, очікується виведення «Followed Sophie».

Може здатися, що два подібні фрагменти коду еквівалентні. Але різниця є, і часто вона може призвести до помилки. І в ході рефакторингу

коду можна помилково перетворити одне на інше, не замислюючись про можливі наслідки.

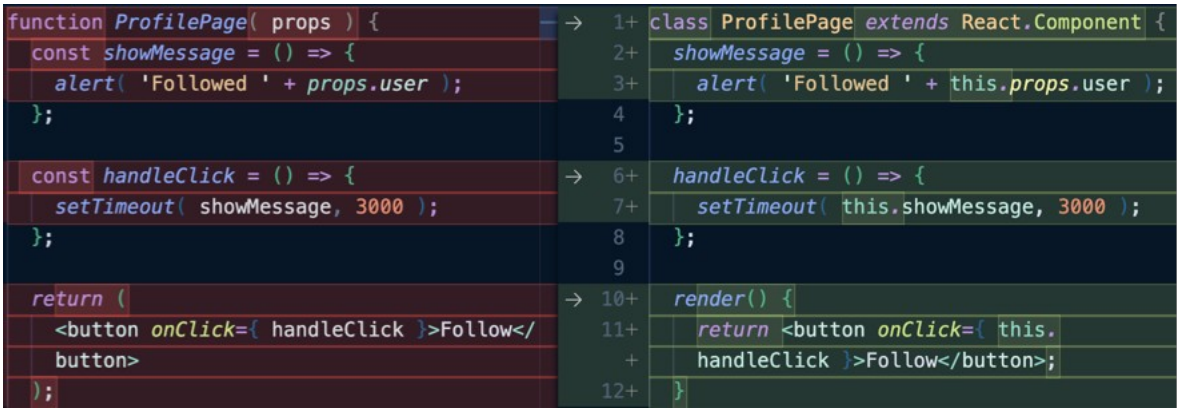



Рисунок 3.1 – ProfilePageFunction і ProfilePageClass

На рисунках 3.2 та 3.3 продемонстровано сторінку, на якій виводиться список, що дозволяє вибирати профілі користувачів, і дві кнопки Follow, що виводяться компонентами ProfilePageFunction і ProfilePageClass, функціональним, і заснованим на класі відносно. Якщо клацнути по кнопці Follow, а потім змінити вибраний профіль швидше, ніж пройдуть 3 секунди після натискання на кнопку, то компоненти поводитимуть себе по різному. При натисканні кнопки, сформованої функціональним компонентом, при поточному вибраному профілі «Sophie» і при перемиканні на профіль «Sunil», у вікні повідомлення буде виведено «Followed Sophie». Якщо зробити те саме з кнопкою, сформованою класовим компонентом, та перемкнути з профілю «Sunil» на «Sophie», буде також виведено «Followed Sophie».


Рисунок 3.2 – Особливості роботи компонента, заснованого на класі


Рисунок 3.3 – Використання функціонального компонента

У даному випадку правильною є поведінка функціонального компонента. Якщо користувач підписався на профіль, а потім перейшов до іншого профілю, то компонент не повинен сумніватися в тому, на чий профіль він підписався. Зочевидь, класова реалізація механізму, що розглядається, містить помилку [32].

Для того, щоб зрозуміти, чому компонент, заснований на класі, поводиться саме так, треба звернути увагу на метод showMessage з рисунку 3.1, а саме на праву, «класову» частину прикладу. Цей метод здійснює читання даних із this.props.user. Властивості React іммутабельні, тому вони не змінюються. Однак this є мутабельною сутністю, і, насправді, мета наявності this у класі полягає саме у можливості this змінюватися. Сама бібліотека React періодично виконує мутації this, що дозволяє працювати зі свіжими версіями методу render та методів життєвого циклу компонента. У результаті, якщо компонент виконує повторний рендерінг під час виконання запиту,

this.props зміниться. Після цього метод showMessage прочитає значення user із «найновішої» сутності props.

Це дозволяє зробити цікаве спостереження, що стосується інтерфейсів користувача. Якщо сказати, що інтерфейс користувача, концептуально, є функцією поточного стану програми, то обробники подій є частиною результатів рендерінгу — так само, як і наявні результати рендерінгу. Обробники подій «належать» до конкретної операції рендерінгу разом з конкретними властивостями та станом.

Однак планування таймауту, зворотній виклик (або «коллбек») якого читає this.props, порушує цей зв'язок. Коллбек showMessage не «прив'язаний» до жодної конкретної операції рендерингу, в результаті він «втрачає» правильні властивості. Читання даних this розриває цей зв'язок.

### 3.2.2 Рішення засобами функціональних та класових компонентів

Як вирішити цю проблему, якщо уявити, що функціональних компонентів у React немає? Потрібен якийсь механізм, що дозволяє «відновити» зв'язок між методом render з правильними властивостями та коллбеком showMessage, що виконує читання даних із властивостей. Цей механізм повинен розташовуватися там, де втрачається сутність props із правильними даними. Один із засобів зробити це полягає в тому, щоб заздалегідь прочитати this.props в обробнику події, а потім у явному вигляді передати те, що було прочитано, функції зворотного виклику, що використовується в setTimeout. Приклад зображено на рисунку 3.4.

```
class ProfilePage extends React.Component {
  showMessage = ( user ) => {
    alert( 'Followed ' + user );
  };

  handleClick = () => {
    const { user } = this.props;
    setTimeout( () => this.showMessage( user ), 3000 );
  };

  render() {
    return <button onClick={ this.handleClick }>Follow</button>;
  }
}
```

Рисунок 3.4 – Передача даних функції зворотного виклику

Такий підхід спрацює, але додаткові конструкції що використовуються, з часом можуть призвести до збільшення обсягу коду і до зростання ймовірності появи помилок. Адже зазвичай потрібне щось більше, ніж єдина властивість, та потрібно ще й працювати зі станом компоненту. Якщо метод showMessage викличе інший метод, і цей метод прочитає this.props.something або this.state.something, це знов викличе ту саму проблему. А для того, щоб її вирішити, довелося б передавати this.props і this.state у вигляді аргументів усім методам, що викликаються з showMessage. Якщо і справді так писати код, то це знищить усі зручності, які надає використання класових компонентів.

Аналогічно, вбудовування коду alert в handleClick не вирішує глобальну проблему. Потрібно структурувати код так, щоб це дозволило розділяти його на безліч методів. Але ще й так, щоб можна було читати властивості та стан, які відповідають операції рендерингу, пов'язаної з конкретним викликом.

Слід зауважити, що проблема полягає не в синтаксисі, що використовується, а саме в тому, що читання даних із this.props виконується занадто пізно. Тому прив'язка this за допомогою bind до методів в конструкторі теж не допоможе. Однак цю проблему можна вирішити завдяки

JavaScript-замиканням. Властивості React є іммутабельними, про що постійно нагадують самі ж розробники React. Це означає, що якщо «замкнути» у замиканні властивості або стан конкретної операції рендерінгу, то завжди можна розраховувати на те, що вони не змінюватимуться.

На рисунку 3.5 показано «захоплення» характеристик під час виклику методу render. При такому підході будь-який код, що знаходиться в методі render (включаючи showMessage), гарантовано буде бачити властивості, захоплені в ході одного конкретного виклику цього методу. У методі render можна описувати скільки завгодно допоміжних функцій і всі вони зможуть користуватися захопленими властивостями і станом. Таким чином замикання дозволило вирішити проблему.

```
class ProfilePage extends React.Component {
  render() {
    const props = this.props; // захоплюємо властивості
    // зверніть увагу, що ми знаходимося всередині методу render.
    // ці функції – не методи класу.
    const showMessage = () => {
      alert('Followed' + props.user);
    };
    const handleClick = () => {
      setTimeout(showMessage, 3000);
    };

    return <button onClick={handleClick}>Follow</button>;
  }
}
```

Рисунок 3.5 – Використання замикання у класовому компоненті

Але навіщо потрібен клас, якщо функції оголошують усередині методу render, а не як методи класу? Можна спростити цей код, позбавившись «оболонки» у вигляді класу, яка його оточує, та створити просту функцію. Тоді, як і попередньому прикладі, властивості виявляються захопленими у функції, оскільки React передає їх у вигляді аргумента. На відміну від this,

React ніколи не виконує мутацій об'єкта props. Для зручності можна ще й деструктурувати props в оголошенні функції, як зображено на рисунку 3.6.

```
function ProfilePage({ user }) { // деструктуризація props
  const showMessage = () => {
    alert('Followed ' + user);
  };

  const handleClick = () => {
    setTimeout(showMessage, 3000);
  };

  return (
    <button onClick={handleClick}>Follow</button>
  );
}
```

Рисунок 3.6 – Використання деструктуризації у функціональному компоненті

Коли батьківський компонент рендеритиме ProfilePage з іншими властивостями, React знову викличе функцію ProfilePage. Але обробник події, який вже викликано, «належить» попередньому виклику цієї функції, у цьому виклику використовується його власне значення user та його власний колбек showMessage, який це значення читає. Все це залишається недоторканим. Саме тому у функціональній версії прикладу з рисунку 3.3 вибір профілю після натискання на відповідну кнопку до виведення повідомлення вже нічого не змінює. Якщо перед натисканням кнопки було обрано профіль Sophie, то у вікні повідомлення буде виведено 'Followed Sophie'.

### 3.2.3 Заміна мутабельності this на мутабельність ref

Як було зазначено, головна різниця функціональних та класових компонентів в React полягає у тому, що функціональні компоненти захоплюють значення. При використанні хуків принцип «захоплення значень» поширюється і на стан. Якщо користувач надіслав якесь

повідомлення, компонент не повинен плутатися в тому, яке саме повідомлення було надіслано.

```
function MessageThread() {
  const [message, setMessage] = useState('');
  const showMessage = () => {
    alert('You said: ' + message);
  };
  const handleSendClick = () => {
    setTimeout(showMessage, 3000);
  };
  const handleMessageChange = (e) => {
    setMessage(e.target.value);
  };
  return (
    <>
      <input value={message} onChange={handleMessageChange} />
      <button onClick={handleSendClick}>Send</button>
    </>
  );
}
```

Рисунок 3.7 – Функціональний компонент MessageThread

Константа message функціонального компонента з рисунку 3.7 захоплює стан, який «належить» тому, що рендерить компонент, який дає браузеру обробник натискання кнопки. В результаті у змінну message зберігається те значення, яке було в полі введення в момент натискання по кнопці Send.

Функціональні компоненти в React захоплюють властивості і стан за замовчуванням. Але що робити, коли потрібно прочитати найсвіжіші дані з властивостей чи стану, що не належать конкретному виклику функції? У компонентах, заснованих на класах, можна було б звернутись до this.props або this.state, тому що this — мутабельна сутність. Її зміною займається React. У функціональних компонентах також можна працювати з мутабельними значеннями, які спільно використовуються всіма компонентами. Ці значення називаються ref.

Сутність ref – це «запасний вихід» у мутабельний імперативний світ. Ця ідея нагадує концепцію DOM refs, але набагато загальніша. Її можна порівняти зі скринькою, в яку програміст може щось покласти. Навіть зовні

конструкція this.latestMessage виглядає дзеркальним відображенням конструкції latestMessage.current. Вони є уявленням однієї концепції.

```
function MessageThread() {
  const [message, setMessage] = useState('');
  const latestMessage = useRef('');
  // можна зчитувати та записувати `latestMessage.current`.

  const showMessage = () => {
    alert('You said: ' + latestMessage.current);
  };
  const handleSendClick = () => {
    setTimeout(showMessage, 3000);
  };
  const handleMessageChange = (e) => {
    setMessage(e.target.value);
    latestMessage.current = e.target.value;
  };
}
```

Рисунок 3.8 – Використання мутабельної сутності ref

За замовчанням React не створює у функціональних компонентах сутності ref для значень властивостей чи стану. У багатьох випадках вони не потрібні, і їх автоматичне створення виявилося б марною тратою часу. Однак роботу з ними, якщо це потрібно, можна організувати самотужки, як продемонстровано на рисунку 3.8.

Якщо прочитати message в showMessage, то результатом буде повідомлення, яке було в полі на момент натискання на кнопку Send. Але якщо прочитати latestMessage.current, то можна отримати найсвіжіше значення навіть якщо користувач продовжує вводити текст в поле після натискання на кнопку Send. Значення ref – це спосіб «ухилення» від однаковості рендерингу, в деяких випадках це може бути дуже доречним.

```
function MessageThread() {
  const [message, setMessage] = useState('');

  const latestMessage = useRef('');
  useEffect(() => {
    latestMessage.current = message; // зберігаємо "найсвіжіше" значення
  });

  const showMessage = () => {
    alert('You said: ' + latestMessage.current);
  };
  const handleSendClick = () => {
    setTimeout(showMessage, 3000);
  };
  const handleMessageChange = (e) => {
    setMessage(e.target.value);
    latestMessage.current = e.target.value;
  };
}
```

Рисунок 3.9 – Використання useEffect для оновлення значення ref

Загалом варто уникати читання чи запису значень ref у процесі рендерингу через те, що ці значення мутабельні. Рендерінг повинен бути передбачуваним. Однак якщо потрібно отримати найсвіжіше значення чогось, що зберігається у властивостях або в стані, ручне оновлення значення ref можна автоматизувати, використовуючи ефект. Приклад наведено у рисунку 3.9.

В даному випадку присвоєння значення відбувається всередині ефекту, в результаті значення ref зміниться тільки після того, як оновиться DOM. Завдяки цьому мутація не порушить роботу методів, які покладаються на безперервність операцій рендерінгу.

Використання значення ref у такий спосіб потрібно нечасто. Зазвичай набагато кращою схемою стандартної поведінки системи є захоплення властивостей чи стану. Однак це може бути зручним під час роботи з імперативними API, на зразок тих, що використовують інтервали або підписки. Та загалом так можливо працювати з чим завгодно: з

властивостями, зі змінними, що зберігаються в стані, з усім об'єктом props або навіть з функцією.

## 3.3 React-hooks

Хук – глобальна функція, яка оголошена всередині React і викликається при кожному рендеру компонента. React відстежує виклики цієї функції і може змінити її поведінку або вирішити, що вона має повернути. Це ще один спосіб описувати логіку компонентів. Він дозволяє додати до функціональних компонентів деякі можливості, властиві тільки компонентам на класах. Насамперед це підтримка внутрішнього стану. Потім підтримка побічних ефектів, наприклад, мережевих запитів, або запитів до WebSocket, типу передплати, відписки від каналів, або запитів до іншого асинхронного чи синхронного API браузера [33].

### 3.3.1 Мотивація та ідея

Подібний компонент, що виконує ту саму функцію, але написаний на хуках, виглядає набагато компактніше, що продемонстровано на рисунку 3.11. Загалом, при портуванні з компонентів на класах на компоненти на хуках код зменшується приблизно в півтора рази.

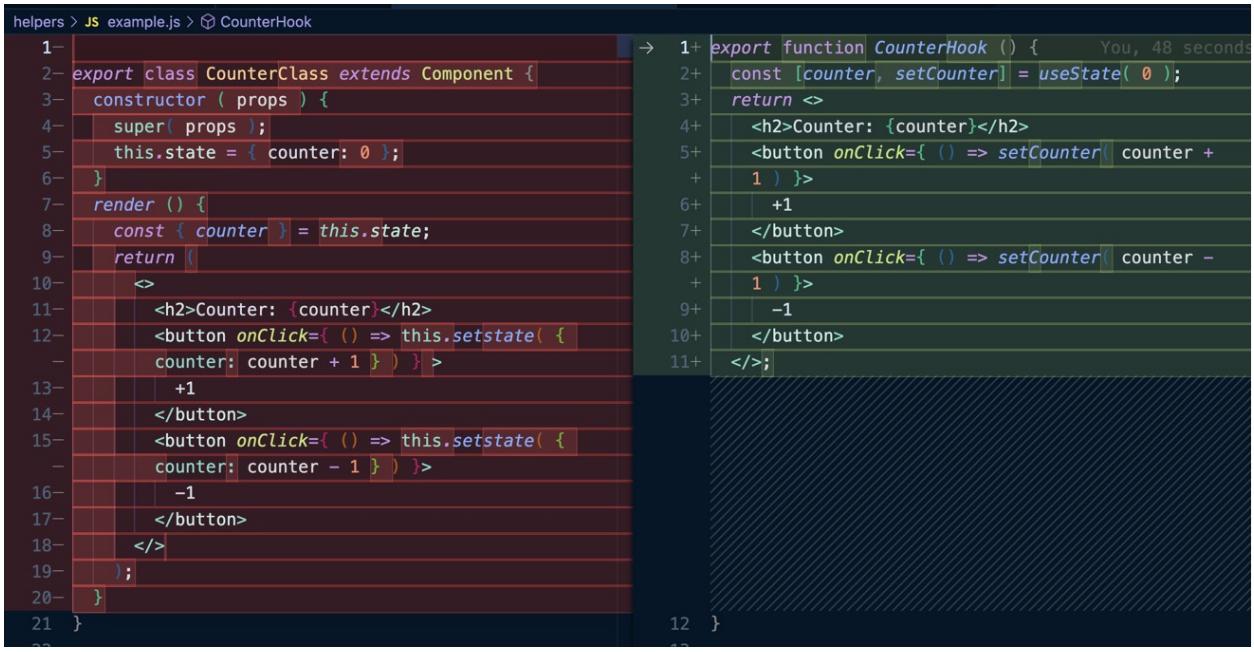



Рисунок 3.11 – Скорочення коду при переході компонента до функціонального підходу

Хуки практично всі дії з життєвим циклом пов'язують з хуком використання ефекту (useEffect), який дозволяє підписатися до монтування, оновлення props, і до розмонтування компонента. У класах для цього доводилося перевизначати кілька методів, типу componentDidMount, componentDidUpdate, і componentWillUnmount. Метод shouldComponentUpdate тепер можна замінити на React.memo [34]. У таблиці 3.1 відображено різницю між класовими та функціональними підходами з огляду життєвого циклу компонента.

Таблиця 3.1 - Життєвий цикл у класових та функціональних компонентах

| **Класи** | **Хуки** |
|---|---|
| Життєвий цикл | |
| componentDidMount | useEffect |
| comnponentDidUpdate | useEffect |

| | |
|---|---|
| componentWillUnmount | useEffect |
| shouldComponentUpdate | React.memo |

Різниця у тому, як обробляється стан, трохи менша. По-перше, у класів існує лише один об'єкт стану. Туди складається все, що завгодно. У хуках можна розбити логічний стан на якісь частини, якими зручно було б оперувати окремо. Метод setState() компонентів на класах дозволяє вказувати патч стейту, тим самим змінюючи одне поле стейту або одразу декілька. У хуках потрібно змінювати весь стейт цілком, і це добре. Це дозволяє розраховувати на те, що об'єкти ніколи не мутуватимуть, вони завжди нові. У таблиці 3.2 відображено різницю між класовими та функціональними підходами з огляду обробки стану.

Таблиця 3.2 - Стан у класових та функціональних компонентах

| **Класи** | **Хуки** |
|---|---|
| Стан | |
| один об'єкт стану | скільки завгодно змінних станів |
| можна міняти одне поле стану | змінюється тільки цілком |
| можна імперативно виконати код після зміни стану (callback) | зміна стану буде лише у наступному циклі рендеру |
| this.setState(state => patch) | setValue(v => newV); |

### 3.3.2 Обмеження та переваги

На використання хуків накладаються деякі обмеження, які відрізняють хуки від звичайних функцій. Насамперед їх не можна використовувати в компонентах на класах, тому що вони створені не для них, а для функціональних компонентів. Хуки не можна викликати всередині внутрішніх функцій, циклів, умов. Тільки на першому рівні вкладеності, всередині функцій компонента. Це обмеження накладає сам React, щоб

одержати можливість відстежувати, які хуки було викликано. І він їх складає у порядку десь «під капотом».

Коли змінюється якесь значення, неможливо підписатися в хуках на зміну цього значення безпосередньо, як можна було раніше другим аргументом у методі setState. Потрібно декларативно підписатися через зміни залежності у useEffect.

Якщо на проекті досить складна логіка і всередині хуків починають використовуватися інші хуки, то, швидше за все, це ознака, що варто винести хук, чи навіть кілька пов'язаних між собою хуків в окремий кастомний хук. І в ньому можна використовувати інші кастомні хуки, тим самим вибудовуючи ієрархію хуків, виділяючи туди загальну логіку.

Хуки надають деяких переваг при порівнянні з класами. Насамперед, використання кастомних хуків робить розподіл логіки набагато простішим. Раніше, використовуючи підхід з HOC, вкладена у компонент вищого порядку логіка була обгорткою над іншим компонентом. Тепер цю логіку закладають всередину хуків. Тим самим вкладеність дерева компонентів зменшується, і React простіше відстежувати зміни компонентів, перераховувати дерево віртуального DOM, тощо. Тим самим вирішується проблема так званого wrapper-hell.

Код, написаний із використанням хуків, набагато легше піддається мінімізації сучасними мінімізаторами типу Terser або старим UglifyJS. Не потрібно зберігати імена методів, не потрібно думати про прототипи. Після транспіляції зазвичай з'являлася купа прототипів, які потім оновлювались. Тепер цього не потрібно робити, тому набагато простіше мінімізувати код.

### 3.3.3 Реалізація найпоширених хуків

Згідно офіційної документації, головні хуки це useState, useEffect та useContext. Додаткові хуки: useMemo, useCallback, useReducer, useRef, useImperativeHandle, useLayoutEffect, useDebugValue, useDeferredValue,

useTransition, useId. Окремо винесено у категорію бібліотечних хуків (Library Hooks) useSyncExternalStore та useInsertionEffect [35].

По кожному з них можна було б написати окрему статтю, і таких статей вже дійсно написано чимало. Окрім того, огляд кожного випадку використання виходить за межі формату цієї роботи. Тому оглянемо реалізацію тільки головних та найпоширених з додаткових React-hooks, та розберемо їх використання на прикладах.

- UseState

Метод useState приймає один аргумент - значення за замовчанням. Повертає кортеж (tuple), в якому є саме значення і метод для його зміни. При створенні одразу декількох станів не потрібно думати про їх декомпозицію. Адже UseState розбиває стейт на логічні частини, щоб їх не змішувати в одному об'єкті, як у класах. І ці частини абсолютно повністю ізольовані, їх можна змінювати незалежно одна від одної. Приклад використання useState зображено на рисунку 3.12.

```jsx
import { useState } from 'react';

function Example() {
  // оголошення нової змінної стану, яку названо "count"
  const [count, setCount] = useState( 0 );

  return (
    <div>
      <p>You clicked {count} times</p>
      <button onClick={ () => setCount( count + 1 ) }>
        Click me
      </button>
    </div>
  );
}
function ExampleWithManyStates() {
  // оголошення декількох змінних стану
  const [age, setAge] = useState( 42 );
  const [fruit, setFruit] = useState( 'banana' );
  const [todos, setTodos] = useState( [{ text: 'Learn Hooks' }] );
  // ...
}
```

Рисунок 3.12 – Використання хуку useState

На відміну від методу setState, який можна знайти в класових компонентах, використання State не об'єднує об'єкти оновлення автоматично. Тобто, він не буде робити merge значень, а просто це значення оновить. За потреби можна повторити цю поведінку, комбінуючи форму функції оновлення з синтаксисом розширення об'єкта. Інший варіант - useReducer, який більше підходить для управління об'єктами стану, що містять кілька значень.

Лістинг 3.1 – Функція оновлення з синтаксисом розширення

```
const [state, setState] = useState({});
setState(prevState => {
  // Object.assign також працюватиме
```

```
  return {...prevState, ...updatedValues};
});
```

- UseEffect

Дуже важливий хук useEffect. Він дозволяє додавати побічні ефекти для компонентів, як альтернативу життєвому циклу. Наприклад, це може бути підписка на події або запити даних. Зазвичай для цього використовувалися методи componentDidMount та componentDidUpdate. Коли викликається useEffect, реакт виконує «побічний ефект» після оновлення змін у дереві DOM. Ефекти оголошуються всередині компонента, тому мають доступ до props та state. Слід зазначити, що useEffect виконується асинхронно, відразу після того, як відбувається зміна DOM. Це гарантує, що він буде виконаний після рендеру компонента, і може привести до наступного рендеру, якщо якісь значення зміняться.

На відміну від componentDidMount, що викликається один раз, useEffect викликається при кожному рендері. UseEffect, useCallback, useMemo, та деякі інші хуки другим аргументом приймають масив значень, які дозволяють сказати, що відстежити. Зміни в цьому масиві приводять до будь-якого ефекту.

Якщо не додати залежності, зміна props відбуватиметься кожен раз, і ця зміна призводитиме до ререндеру компонента. Як наслідок, ефект буде перезапрошено, і тим самим з’являється нескінченна низка запитів на сервер, і компонент сам себе постійно оновлює, оновлює, оновлює. Щоб це виправити, потрібно указати залежності, зміни в яких перезапускатимуть ефект. Якщо вказати порожній масив, то ефект після першого запуску більше не перезапускатиметься. На рисунку 3.13 зображено компонент вибору автора зі списку, і таблички з книжками цього автора. При зміні authorId буде викликано запит і завантажено новий список книжок.

```
function BookList () {
  const [authorId, setAuthorId] = useState( null );
  const [books, setBooks] = useState( null );
  const [message, setMessage] = useState( null );

  useEffect( () => {
    const subscription = messages.subscribe( setMessage );
    return () => subscription.unsubscribe(); // відписка (unsubscribe)
  }, [] ),

  useEffect( () => {
    fetch( `books${authorId ? '?author='+ { authorid } : ''}` )
      .then( ( resp ) => resp.json() )
      .then( setBooks );
  }, [authorId] ); // dependencies - масив залежностей
  return (
    <>
      <AuthorSelect value={ authorId } onChange={ setAuthorId } />
      <BookTable value= { books } />
    </>
  );
}
```

Рисунок 3.13 – Використання хуку useEffect

Із useEffect завжди можна повернути функцію очищення, яка використовується для видалення залишків роботи компонента. Функція очищення викликається не тільки перед розмонтуванням компонентів, але і перед виконанням наступного ефекту. Тому важливі події типу відписки компоненту від оновлень, які слід проводити тільки у разі розмонтування, потрібно ізолювати у окремий ефект з порожнім масивом залежностей. У наведеному прикладі при першому монтуванні компонент підписується на нові повідомлення, а при розмонтуванні проводить очистку та відписується від зайвих навантажень пам’яті.

- UseContext

UseContext дозволяє використовувати замість renderProps звичайне повернуте значення, до якого слід передавати контекст, який необхідно вилучити. На рисунку 3.14 зображено різницю між useContext та HOC.

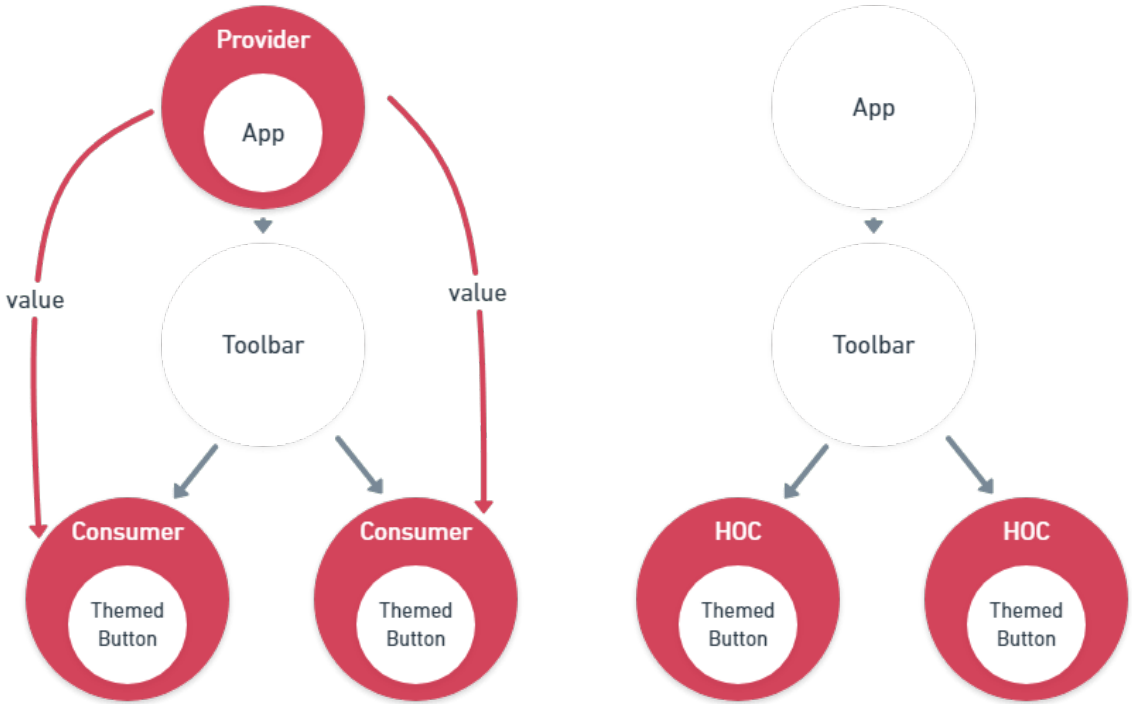



Рисунок 3.14 – UseContext vs HOC

Стан постачальника (Provider) надається споживачам (Consumer), а HOC мають власний стан. Можна зберегти батьківський компонент у Context.Provider і використовувати його всередині функцій Context.Consumer. UseContext змінює Context.Consumer і дозволяє отримувати дані за допомогою useContext. Таким чином можливо відмовитися від усіх HOC, які передавали контекст у props, і використовувати об’єкт просто у повернутому значенні.

Ідея появи цього хука, напевно, йде від бажання командою React позбутися Redux на користь власного використання контексту. Але в useContext поки що відсутня важлива особливість Redux - можливість реагувати на зміни лише частини контексту. Наведені у прикладі на рисунку 3.15 рядки коду не є еквівалентами з точки зору продуктивності. У випадку з

useSelector, повторний рендерінг відбудеться лише за зміни імені, з useContext – при зміні будь-яких даних користувача.

```
import React, { useContext, useState } from 'react';
import { useSelector } from 'react-redux';

const Func = () => {
  // Redux
  const nameFromRedux = useSelector( ( state ) => state.user.name );
  // контекст React
  const { name } = useContext( UserContext );
```

Рисунок 3.15 – Використання useContext та useSelector

Звісно, потрібно обережно користатися цим хуком. Адже величезний контекст ймовірно стане причиною проблем з продуктивністю. Тому для запобігання повторному рендерінгу, доводиться розділяти контекст на частини. Проте велика кількість провайдерів контексту нагадує той самий wrapper-hell, від якого не могли позбутися HOC. На рисунку 3.16 наведено приклад «provider-hell».

```
return (
  <AuthContext.Provider value={ authProvider }>
    <DataProviderContext.Provider value={ dataProvider }>
      <StoreContextProvider value={ store }>
        <QueryClientProvider client={ queryClient }>
          <AdminRouter history={ history } basename={ basename }>
            <I18nContextProvider value={ i18nProvider }>
              <NotificationContextProvider>
                <ResourceDefinitionContextProvider>
                  {children}
                </ResourceDefinitionContextProvider>
              </NotificationContextProvider>
            </I18nContextProvider>
          </AdminRouter>
        </QueryClientProvider>
      </StoreContextProvider>
    </DataProviderContext.Provider>
  </AuthContext.Provider>
);
```

Рисунок 3.16 – UseContext: provider-hell

- UseMemo

Один з існуючих засобів оптимізації React-програм на хуках це мемоізація. Мемоізацію можна розділити на внутрішню та зовнішню. Приклад наведено на рисунку 3.17.

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

function Example( { a, b } ) {
  const memoizedValue = useMemo(
    () => computeExpensiveValue( a, b ), [a, b],
  );
  // ...
}

function MyComponent ( props ) {
  /* рендер з використанням props */
}
function areEqual ( prevProps, nextProps ) {
  // повертає true, якщо nextProps рендерить той самий результат,
  // що і prevProps, інакше повертає false
}
export default React.memo( MyComponent, areEqual );
```

Рисунок 3.17 – Використання React.memo та useMemo

Зовнішня мемоізація – це React.memo, практично альтернатива класу React.PureComponent, який відслідковував зміну пропсів і змінював компоненти тільки тоді, коли змінилися пропси або стейт. Тут схожий принцип, але без стейту. Така мемоізація відслідковує зміни у пропсах, і якщо вони змінилися, відбувається ререндер. Якщо пропси не змінювалися, компонент не оновлюється, і на цьому заощаджуються ресурси. Внутрішній метод оптимізації - useMemo, використовується рідше. Метод дозволяє обчислювати якесь значення, і перераховувати його лише якщо вказані у залежностях значення змінилися.

- UseCallback

Цей хук можна назвати окремим випадком мемоізації для функцій. Він, перш за все, застосовується для того, щоб замемоізувати значення функцій-

обробників подій, які будуть передані в дочірні компоненти. Для використання потрібно описати якусь функцію, обгорнути її в useCallback, і вказати, від яких змінних вона залежить. Все це потрібно для запобігання рендеру зайвого разу. Приклад використання useCallback наведено на рисунку 3.18.

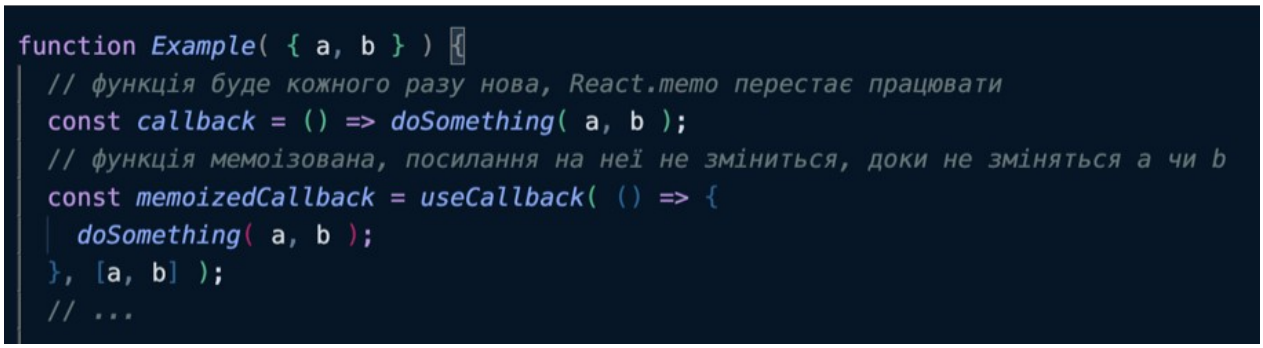



Рисунок 3.18 – Використання хуку useCallback

Більш наближений до реального використання приклад наведено на рисунку 3.19. Тут є сторінка зі списком хостів і фільтри, які дозволяють цей список хостів фільтрувати. Змінну maintenance додано для ілюстрації значення, що часто змінюється, та яке призводить до повторного ререндера.

В даному випадку проблема полягає в тому, що зміна всередині maintenance буде призводити до ререндера в тому числі і компонента фільтрів, тому що він підписується на зміну фільтрів. Туди також передається функція onChange, яка при кожному рендері буде нова. Тому якщо компонент HostFilters складний, наприклад, це купа dropdown, в які завантажено оброблені дані, і якщо він перерендерюватиметься зайвий раз, то продуктивність може знизитися. З'являться лаги, або просто знову нескінченна низка запитів. Тому обов'язково потрібно цей onChange обгорнути в useCallback.

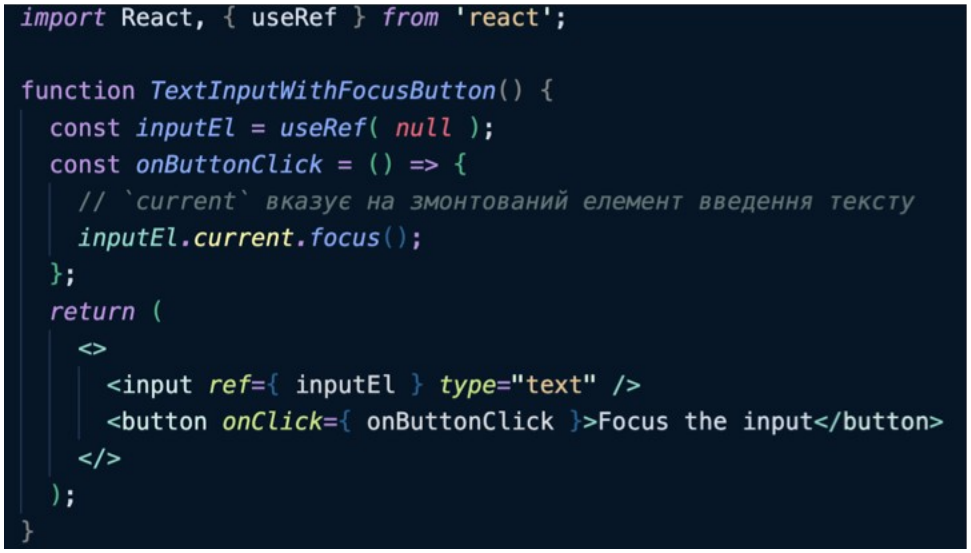
```
function Hosts () {
  const maintenance = useMaintenance ();
  const [hosts, setHosts] = useState ([]);
  const [filters, setFilters] = useState ({});

  useEffect ( () => {
    hostApi
      .getList(filters)
      .subscribe(resp => setHosts(resp.result) );
  }, [filters]);

  const onChange = useCallback( (event, f) => setFilters (f) , []) ;

  return (
    <>
      { maintenance !== null ? (
        <p>Maintenance duration is {maintenance}</p>
        ) : (
        null
      ) }
      <HostFilters value={filters} onChange={onChange} />
      <HostList value={hosts} />
    </>
  )
}
```

Рисунок 3.19 – Використання хуку useCallback

У даному випадку він ні від чого не залежить, тому в кінці вказано порожній масив залежностей.

- UseRef

UseRef – хук для роботи з сутністю ref, яку було оглянуто у підрозділі 3.2.3. Він повертає мутоване значення, де поле «.current» буде ініційовано першим аргументом, а сам об'єкт буде існувати доки існує компонент. З точки зору React, посилання призначені не тільки для вузлів DOM, але також є еквівалентом цього у функціональних компонентах. Або, іншими словами, ref посилання – це "стан, зміна якого не тягне за собою повторний рендерінг". Приклад використання useRef  наведено у рисунку 3.20.



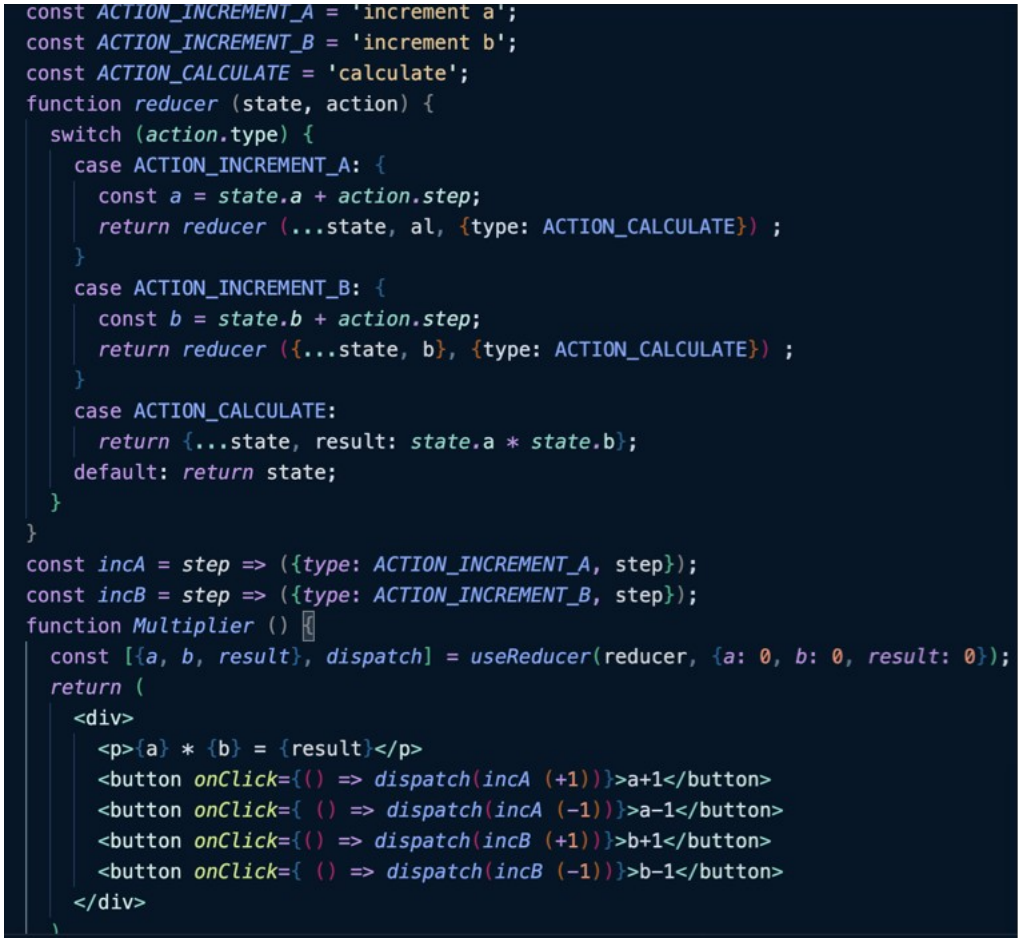
```
import React, { useRef } from 'react';

function TextInputWithFocusButton() {
  const inputEl = useRef( null );
  const onButtonClick = () => {
    // `current` вказує на змонтований елемент введення тексту
    inputEl.current.focus();
  };
  return (
    <>
      <input ref={ inputEl } type="text" />
      <button onClick={ onButtonClick }>Focus the input</button>
    </>
  );
}
```

Рисунок 3.20 – Використання хуку useRef

Звертатися безпосередньо до вузлів DOM вважається неправильним, адже за думкою команди React він, DOM, «брудний» (dirty) [36]. Тому потрібно використовувати ref-посилання. Але необхідно уважно слідкувати за розповсюдженням цих посилань. У більшості випадків, при використанні посилань компонентом, він передає їх нащадкам. Якщо нащадком є компонент React, він має перенаправити посилання до іншого компонента, і так до тих пір, поки один із компонентів, нарешті, не відрендерить відповідний елемент HTML (forward ref). Варто окремо зазначити, що forwardRef() не дозволяє створювати generic-компоненти у випадку використання TypeScript.

- UseReducer

Редюсер – це функція, яка приймає об’єкт стейту і об'єкт, зазвичай званий «action», який описує те, як цей стейт має змінитися. Хук передає параметри, а reducer вже вирішує, як зміниться стейт залежно від тих параметрів, і в результаті повертає новий об’єкт стейту з оновленнями.



```
const ACTION_INCREMENT_A = 'increment a';
const ACTION_INCREMENT_B = 'increment b';
const ACTION_CALCULATE = 'calculate';
function reducer (state, action) {
  switch (action.type) {
    case ACTION_INCREMENT_A: {
      const a = state.a + action.step;
      return reducer (...state, al, {type: ACTION_CALCULATE}) ;
    }
    case ACTION_INCREMENT_B: {
      const b = state.b + action.step;
      return reducer ({...state, b}, {type: ACTION_CALCULATE}) ;
    }
    case ACTION_CALCULATE:
      return {...state, result: state.a * state.b};
    default: return state;
  }
}
const incA = step => ({type: ACTION_INCREMENT_A, step});
const incB = step => ({type: ACTION_INCREMENT_B, step});
function Multiplier () {
  const [{a, b, result}, dispatch] = useReducer(reducer, {a: 0, b: 0, result: 0});
  return (
    <div>
      <p>{a} * {b} = {result}</p>
      <button onClick={() => dispatch(incA (+1))}>a+1</button>
      <button onClick={ () => dispatch(incA (-1))}>a-1</button>
      <button onClick={() => dispatch(incB (+1))}>b+1</button>
      <button onClick={ () => dispatch(incB (-1))}>b-1</button>
    </div>
```

Рисунок 3.21 – Використання хуку useReducer

На рисунку 3.21 useReducer приймає функцію reducer, і початкове значення стейту другим параметром. Повертає, так само як і useState, поточний стейт і функцію для його зміни, dispatch. Якщо передати в dispatch об'єкт-action, буде викликано зміну стейту.

Існує певна проблема при написанні коду на TypeScript і використанні строгої типізації. Залежно від типу, структура action може бути зовсім різною. Тому зробити так, щоб працювало автозавершення, та інші переваги статичної типізації, досить складно. Але можливо.

Насамперед, потрібно оголосити всі перерахування з усіма типами action. Звичайно, вони не повинні перетинатися, тому компілятор не дасть зробити однакові рядки. Для кожного типу action треба описати структуру, розширюючи його якимись полями і обов'язково вказуючи тип. Потім

створити UnionType «Action», де через вертикальну межу будуть перераховані всі типи даних що складаються, якими можуть бути action. Це перший крок.

```
import { useReducer } from 'react';

enum ActionType{ IncrementA, IncrementB, Calculate }
interface IActionIncrementA {step: number; type: ActionType. IncrementA; }
interface IActionIncrementB {step: number; type: ActionType.IncrementB;}
interface IActionCalculate {type: ActionType.Calculate;}
type Action = IActionIncrementA | IActionIncrementB | IActionCalculate;

const initialState = { a: 0, b: 0, result: 0 };
type State = Readonly<typeof initialState>;

const reducer = ( state: State, action: Action ) => {
  switch( action?.type ) {
  case ActionType.Calculate:
    return { ...state, result: state.a * state.b };
  case ActionType.IncrementA: {
    const a = state.a + action.step;
    return reducer( { ...state, a }, { type: ActionType.Calculate } );
  }
  case ActionType.IncrementB: {
    const b = state.b + action.step;
    return reducer( { ...state, b }, { type: ActionType.Calculate } );
  }
  default:
    return state;
  }
};
```

Рисунок 3.22 – Використання хуку useReducer з TypeScript

Другий крок – обчислення типу на основі значення. Наприклад, тут є initialState, просто об'єкт. І загалом, треба описувати структуру цього об'єкта окремим інтерфейсом. Але можна перекласти це завдання на TypeScript. Він вміє сам обчислювати значення. На рисунку 3.22 type State визначається тільки для читання, та обчислюється з initialState [37].

Далі ці типи легко використовувати в reducer. На вхід передаються State та Action, і відбувається перевірка switch по action.type. У TypeScript є така можливість при використанні UnionType: він може обчислити дозволений тип усередині case, якщо ці типи відрізняються значенням якогось поля, у даному випадку - type. І таким чином з’являється можливість

звертатись до полів action потрібного типу. Перш за все, це зручно. А ще це спрощує валідацію за типами і рефакторинг, та дозволяє користатися підказками автокомпліту.

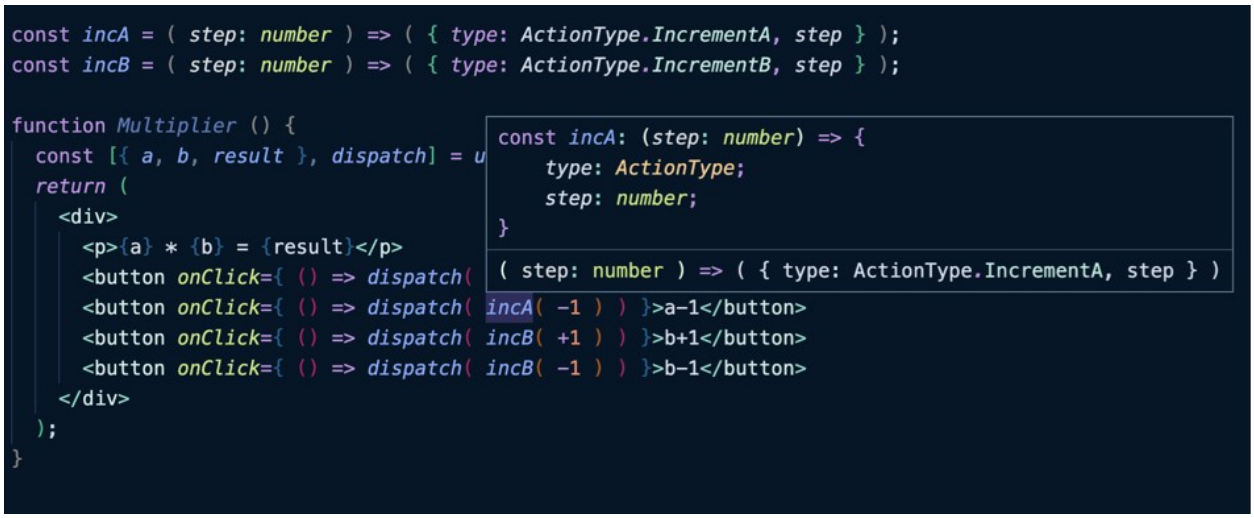



Рисунок 3.23 – Використання хуку useReducer з TypeScript

- UseCustomHook

Якщо потрібно перевикористовувати логіку компонентів із збереженням стану, зазвичай для цього використовують або HOC, або візуалізацію шаблонів пропсів, але вони створюють додатковий код додатка.

```
import { useState, useEffect } from 'react';
// створення
function useFriendStatus(friendID) {
  const [isOnline, setIsOnline] = useState(null);
  const handleStatusChange = (status) => setIsOnline(status.isOnline);
  useEffect(() => {
    ChatAPI.subscribeToFriendStatus(friendID, handleStatusChange);
    return () => ChatAPI.unsubscribeFromFriendStatus(friendID, handleStatusChange)
  });
  return isOnline;
}
// використання
function FriendStatus(props) {
  const isOnline = useFriendStatus(props.friend.id);
  if (isOnline === null) return 'Loading...';
  return isOnline ? 'Online' : 'Offline';
}
// або так
function FriendListItem(props) {
  const isOnline = useFriendStatus(props.friend.id);
  return (
    <li style={{ color: isOnline ? 'green' : 'black' }}>{props.friend.name}</li>
  );
}
```

Рисунок 3.24 – Створення та використання кастомного хуку

На рисунку 3.24 відображено кастомний хук, який можна викликати в різних компонентах, та варіанти його виклику. В обох випадках перевикористовується стан компонента, кожна функція useFriendStatus створює ізольований стан. Також варто відзначити, що початок цієї функції починається зі слова use, це говорить React про те, що це хук. Потрібно дотримуватися цього формату. Кастомні хуки можна писати для створення та відстежування анімації, підписки, таймерів та багато іншого.

## 3.4 Бенчмаркінг

Тепер, коли головні засоби використання функціональних компонентів та практичну реалізацію хуків розібрано, можна було б перейти до порівняння результатів швидкодії різних підходів. Про тестування та порівняння функціональних і класових компонентів React існує дуже багато інформації, статей та доповідей. Але є один приклад, важливий тим, що на нього відреагував та дав відповідь Ден Абрамов, один з головних розробників React, Redux, create-react-app та інших корисних програмних рішень екосистеми React.JS [38]. В ході даної роботи оглянемо лише цей окремий випадок, який відображує безпосередньо думку команди розробників React.

Автором статті було створено тестову програму, яка рендерить 10 000 екземплярів функціонального компонента, який має 3 значення стану, і ефект, який встановлює 3 значення стану один раз після першого рендеру. Основний компонент реєструє час, що минув від створення екземпляра кореневого компонента до моменту завершення відтворення 10 000 елементів. Для цього він також використовує ефект.

Лістинг 3.2 – частковий код файлу тесту з використанням React-hooks

```
const Benchmark = ({ start }) => {
  useEffect(() => {
    console.log(Date.now() - start);
  });
  return array.map((item, index) => <Component key={index} />);
};
render(<Benchmark start={Date.now()} />, document.getElementById('root'));
```

Лістинг 3.3 – частковий код файлу тесту з використанням React-HOC та власної бібліотеки автора «reactorlib»

```
const Benchmark = compose(
  withEffect(({ start }) => {
    console.log(Date.now() - start);
  })
)(_Benchmark);
render(<Benchmark start={Date.now()} />, document.getElementById('root'));
```

Тести проводилися на 12-дюймовому Macbook початку 2015 року (1,1 ГГц Core M, 8 ГБ оперативної пам’яті) під керуванням MacOS Sierra 10.12.3 і Chrome 71. Обидва варіанти мали версію React 16.8.1. Автор тесту, Арнел Енеро, зафіксував результати 10 тестових запусків на MacOS Chrome, і зауважив, що усі з них показали явного переможця [39]. Результати тесту зображено у таблиці 3.3.

Таблиця 3.3 – Час рендерінгу у мілісекундах

| Run# | Hooks | HOCs |
|---|---|---|
| 1 | 2197 | 1440 |
| 2 | 2302 | 1757 |
| 3 | 2749 | 1407 |

| | | |
|---|---|---|
| 4 | 2243 | 1309 |
| 5 | 2167 | 1644 |
| 6 | 2219 | 1516 |
| 7 | 2322 | 1673 |
| 8 | 2268 | 1630 |
| 9 | 2164 | 1446 |
| 10 | 2071 | 1597 |

Але Ден Абрамов відповів на це цілою статтею з іншими тестами, які доводять, що цей контрольний показник має багато важливих недоліків. По-перше, React у даному випадку працював у режимі розробки. Режим розробки має додаткові витрати, щоб забезпечити кращі попередження, але це робить його непридатним для тестів, оскільки результати нічого не кажуть про реальну продуктивність. Тому на сторінці документації щодо оптимізації продуктивності React першою порадою є використання виробничої збірки [40].

Після запуску обох тестів у робочому режимі порівняльні результати змінюються чимало, що показано у таблиці 3.4. І все ж ці результати все ще невірні, оскільки ці два фрагменти не виконують ідентичного порівняльного тесту. Версія Hooks оновлює стан і вимірює час у useEffect(). Як пояснюється у офіційній документації з Hooks-API, useEffect дозволяє браузеру відрендерити екран перед запуском ефектів. На думку команди розробників, з точки зору користувача, useEffect зазвичай забезпечує кращий досвід, оскільки початковий рендер не чекає, поки ефекти запустяться.

Таблиця 3.4 – Тест у робочому режимі React

| Hooks | HOCs |
|---|---|
| 175 | 156 |
| 171 | 164 |
| 169 | 154 |

| | |
|---|---|
| 181 | 138 |
| 167 | 159 |
| 153 | 194 |
| 151 | 152 |
| 155 | 152 |
| 147 | 163 |
| 160 | 162 |

Загалом, ефекти у програмах не змінюють стан. Зазвичай марнотратно чекати, поки всі вони виконаються, перш ніж почати рендерінг. Фактично, це поширена проблема продуктивності в класах React. Ось чому useEffect не блокує браузер від рендеру за замовчуванням.

У результаті очікування першого рендеру версія useEffect цього тесту виконує код вимірювання пізніше, ніж версія класу. Тому і виглядає «гірше». Аналізуючи вихідний код реалізації тесту, можна побачити, що HOC у цьому прикладі встановлює стан і реєструє час у методі життєвого циклу componentDidMount, який запускається синхронно. Це означає, що він спрацьовує раніше, але користувач фактично ще нічого не бачить, оскільки фрейм, власне, структура додатку, ще не була відбудована. Абрамов називає це порівнянням яблук з апельсинами. Щоб порівняти відносні речі, він пропонує спробувати useLayoutEffect(), який працює на етапі макета, подібно до componentDidMount(). Результати такого тесту відображено у таблиці 3.5. І тут вже виявляється, що версія тесту з Hooks навіть трохи швидша після того, як тести було виправлено для вимірювання одного й того ж.

Таблиця 3.5 – Результати тесту з використанням useLayoutEffect

| Hooks | HOCs |
|---|---|
| 121 | 156 |
| 106 | 170 |
| 111 | 157 |

| | |
|---|---|
| 141 | 152 |
| 111 | 156 |
| 121 | 158 |
| 105 | 170 |
| 111 | 162 |
| 108 | 166 |
| 108 | 157 |

Чи означає це, що useLayoutEffect швидший за useEffect? Абсолютно ні, це неправильний спосіб думати про це. Це лише означає, що useLayoutEffect або componentDidMount запускаються раніше. Але якщо конкретний варіант використання не пов’язаний із компонуванням, наприклад, розміщення спливаючої підказки, зазвичай бажано дозволити спочатку рендерінг екрану, а потім запуск ефектів. Це саме те, що useEffect дозволяє робити.

Якщо зберегти оновлення стану в useEffect, але вимірювати час у useLayoutEffect чи componentDidMount, можна отримати ще кращі цифри. Результати такого порівняння приведено у таблиці 3.6. Але це нежиттєздатний випадок, бо вони не виконують рівноцінну роботу. Адже, версія useEffect виконує менше роботи перед рендером, тому що спочатку показується початковий результат візуалізації, а потім запускаються ефекти, які змінюють стан. Це означає, що під час оновлення на екрані з’являється мерехтіння. Тому останній тест не є реалістичним інтерфейсом користувача, і знову неможливо сказати однозначно, яке рішення буде найкращим. Хуки дозволяють розробникам обирати самостійно, чи блокувати ефекти рендерінгу чи ні. Та самі розробники і повинні вирішувати, що є кращим для конкретного випадку використання.

Таблиця 3.6 – Порівняння useEffect та componentDidMount

| Hooks | HOCs |
|---|---|
| 106 | 174 |
| 104 | 198 |
| 88 | 149 |
| 88 | 154 |
| 88 | 149 |
| 89 | 163 |
| 85 | 162 |
| 90 | 157 |
| 89 | 164 |
| 91 | 153 |

## 3.5 Висновки розділу

Використання функціональної парадигми надає React таких корисних характеристик:

- швидкість розробки висока, а функції, що часто використовуються, можуть постійно повторно використовувати логіку. Функція є чорною скринькою ззовні і може використовуватися безпосередньо без побічних ефектів;
- можна легко зрозуміти корисність функції через ім'я, не знаючи внутрішньої реалізації;
- код більш зрозумілий, код, що фактично викликається, дуже простий, вся складна логіка інкапсульована в функції;
- паралельна обробка даних легка у виконанні, оскільки метод є чистою функцією і не впливає на зовнішні змінні, а послідовність обробки може бути організована довільно.

Хуки вирішують декілька проблем, але всі їх можна звести до передачі посилань. Краще ментальне правило, яке застосовується до хуків, яке зустрічається в інтернеті, полягає в тому, що програмувати треба так, ніби будь-яке значення може змінитися в будь-який час [41]. Та взагалі, можна

сприймати хуки як прості функції. Правил, що вимагаються при використанні хуків, всього два:

1) Для того, щоб реакт зміг контролювати порядок виконання хуків, їх слід виконувати тільки у верхній частині ієрархії функції. Не можна викликати всередині умов, циклів, вкладених функцій тощо. Якщо потрібно, можна визначити кастомний хук і викликати з нього.

2) Не можна використовувати хуки у компонентах на класах, тільки в React-функціях, в кастомних хуках, або саме у функціональних компонентах.

Для дотримання цих правил існує окремий лінтер, що повідомить про помилку, коли відбувається виклик хуків в компонентах класу або в циклах та умовах.

До переваг використання React-hooks відносять наступне:

- легше ділитися логікою стану (кастомні хуки);
- рятування від HOC і wrapper-hell;
- ефективніша мінімізація;
- не потрібно зберігати імена методів;
- не потрібно працювати з прототипами класів;
- не потрібно пам'ятати про this;
- спрощення логіки, пов'язаної із життєвим циклом;
- гнучкіша можливість оптимізації за рахунок мемоізації.

Зростання популярності JavaScript призвело до підвищення інтересу до концепцій функціонального програмування. Дослідні розробники в галузі функціонального програмування згодом почали працювати над середовищами Single Web Application - SPA, і в результаті з’явилися Redux, React, MobX та інші бібліотеки, які наразі використовуються мільйонами людей.

Починаючи з версії 7.1.0, React-Redux підтримує API хуків та надає такі хуки як useDispatch та useSelector, а починаючи з версії 5.1, з'явилася

підтримка хуків і в React-Router. Екосистема React має усі можливості для використання функціональної парадигми при створенні веб-застосунків. Розробники офіційно обіцяють і далі підтримувати класи, але відверто кажуть, що саме хуки стануть головним способом написання React-компонентів.

Та якщо більшість коду додатків буде заснована на функціональних компонентах, це означає, що всім розробникам потрібно більше знати про оптимізацію коду, і про те, які значення можуть з часом змінюватися.

У підрозділі 3.2 було розглянуто один із неправильних патернів використання класових компонентів, та виявлено вирішення цієї проблеми за допомогою замикань. Було доведено, що замикання допомагають виправляти невеликі проблеми, які складно помітити. Вони також спрощують написання коду, який правильно працює в паралельному режимі. Це можливо завдяки тому, що всередині компонента «замикаються» правильні властивості та стан, з якими цей компонент було відрендерено. Зазвичай, проблема «застарілих замикань» відбувається через помилкове припущення про те, що «функції не змінюються», або про те, що «властивості завжди залишаються однаковими». В даному розділі було доведено, що це не так.

У підрозділі 3.3 було виконано огляд та аналіз особливостей використання найпоширених хуків. Функції «захоплюють» свої властивості та стан, і розуміння того, про які функції йде мова, це дуже важливо. Це не помилка, а особливість функціональних компонентів. Функції не слід виключати з масиву залежностей для useEffect або useCalback.

UseEffect – це уніфікація обробки подій монтування, розмонтування та оновлення в одному інтерфейсі. Але потрібно пильно стежити за залежностями, та бути впевненими, що усі реактивні змінні включено до масиву залежностей. Нерідко однією із залежностей може бути створена функція. Оскільки React не бачить різниці між змінною та функцією, тепер запобігати повторному рендерингу доводиться за допомогою useCallback(),

який також вимагає наявності масиву залежностей. Через необхідність керувати залежностями простий компонент з кількома обробниками подій та колбеками життєвого циклу може стати купою локшинного коду. Але це не мінус useEffect() чи useCallback(). Скоріше, такий код свідчить про необхідність розбиття цієї функції, та можливо навіть перебудови логіки компонента.

UseRef – це альтернатива React.createRef, що схоже на useState, але зміни в ref не призводять до відтворення. UseImperativeHandle дозволяє об'явити в компоненті певні публічні методи. UseContext — корисній хук, що дозволяє взяти поточне значення з контексту, якщо провайдер вище за рівнем ієрархії визначив це значення.

У підрозділі 3.4 було оглянуто та розібрано один окремий випадок тестування та порівняння функціональних та класових компонентів. Проте цей приклад відображає ставлення до таких тестів та їх результатів команди розробників React в обличчі Дена Абрамова. Цього достатньо, щоб зрозуміти як взагалі проводити порівняння, та на які проблеми слід звернути увагу при проведенні таких тестів.

Іноді може здаватися, що хуки більш швидкі за компоненти вищого порядку, та забезпечують необхідні параметри за замовчуванням. Та бувають випадки, коли один підхід перемагає інший, і це залежить від багатьох речей. Еталонний тест, який рендерить тисячі текстових вузлів і потім їх одноразово оновлює, насправді не відображає проблем продуктивності, які виникають у реальних програмах. Він також не надає достатнього контексту про варіант використання, щоб розглянути різні компроміси.

Практичне використання функціонального підходу при розробці програмного забезпечення в екосистемі React визнано доцільною та актуальною задачею.

# РОЗДІЛ 4
# ПРОГРАМНИЙ МОДУЛЬ

## 4.1 Види рендерінгу додатків SSR, SSG, CSR

Раніше, у часи становлення Інтернету, нічого не генерувалося динамічно, і були лише статичні сторінки. Звичайні, заздалегідь створені, статичні HTML документи надсилалися клієнту. Коли користувач заходив на веб-сайт, на сервер надсилався простий HTTP запит, а у відповідь надходила розмітка, яка відображалася в браузері. Потім з'явилася можливість використовувати динамічний рендерінг і будувати шаблони для розмітки. Ці шаблони дозволяли заповнювати HTML інформацією, що надсилається клієнту. Кожен запит HTTP проходив через серверну частину веб-сайту, наприклад PHP, і збирав необхідні дані. Завдяки цьому з'явилася можливість додавати такі елементи, як імена користувачів, поточні дати, дані з бази даних, та інші [42].

Це був початковий Server Side Rendering. Клієнт (браузер) запитував у сервера необхідний для відображення HTML, який був згенерований саме на сервері, і далі відображався у браузері.

З появою технології AJAX з'явилася можливість вимагати дані асинхронно, без перезавантаження всієї сторінки. З точки зору UX це було величезним поліпшенням, адже вирішувало проблему мерехтіння сторінок. Таким чином галузь веб-розробки почала рухатися в бік Client Side Rendering. CSR – рендерінг на клієнті – це рендерінг програми у браузері за допомогою DOM.

Для спрощення роботи з фронтендом стали створюватися різні бібліотеки та фреймворки. З’явилися React, Angular і Vue, завдяки яким переважна більшість розробників познайомилися з повноцінним рендерінгом

на клієнті. Всі ці три технології використовують компонентний підхід і дозволяють розбивати розмітку на дрібні частини, що перевикористовуються. При такому підході генерація HTML відбувається саме на фронтенді, а бекенд використовується тільки для складних обчислень, роботи з базами даних, насичення фронтенду даними, тощо.

Логічним розвитком моделі CSR стала поява Single Page Application. SPA – це сайт чи додаток, що в якості оболонки для сторінок використовує єдиний HTML-документ, через що такі додатки і називають односторінковими. Завдяки можливості легко генерувати розмітку на стороні клієнта, SPA завоювали величезну популярність. Сервер відправляв клієнту практично порожній HTML, який містить лише базову розмітку, з порожнім  і один файл зі скриптом <script src='bundle.js'></script>, всередині якого написано весь код для генерації розмітки, стилів і логіки за допомогою JavaScript.

З часом, цей підхід призвів до цілої низки потенційних проблем. По-перше, виникла проблема з пошуковою оптимізацією. Оскільки пошукові роботи Google читають і індексують веб-сайти, необхідно віддавати інформацію про контент і розмітку з сервера, а при такому підході все генерується на клієнті. Тоді ще пошукові системи не були здатні обробити інформацію згенеровану таким чином. Все, що міг побачити робот, це порожній кореневий HTML-тег. Наразі ситуація дещо покращилася, тому що багато пошукових роботів навчилися виконувати Javascript, який необхідний для Client Side Рендерінгу. Проте результат такої індексації все ще залишає бажати кращого.

По-друге, потенційною проблемою є продуктивність. Оскільки для відображення сторінки в браузері потрібно виконання великої кількості JavaScript, то програма може працювати повільно, або з затримками. Особливо це помітно на старих мобільних пристроях.

Для вирішення цих проблем розробники знову повернулися до ідеї створення розмітки на сервері. Однак, на відміну від старого підходу, коли розмітка генерувалася на сервері за допомогою серверних мов програмування типу PHP, сьогодні SSR, як і CSR, використовує сучасні JavaScript бібліотеки на кшталт React. Різниця в тому, що програма генерує розмітку за допомогою React не на клієнтській стороні, а на серверній. Це рішення було спрямоване на виправлення існуючих проблем з продуктивністю та SEO. Щоразу, коли пошуковий робот запитує сторінку, він отримає необхідний контент. Тому для візуалізації більше не потрібно виконання JavaScript на клієнті. Тепер, коли в розробці згадують термін Server Side Rendering, то посилаються саме на цей, новий, сценарій.

### 4.1.1 Client Side Rendering

Відомо, що коли користувач відкриває веб-сторінку, браузер робить запит до серверу, та чекає на HTML у відповідь. Потім завантажується весь JavaScript, який було пролінковано у HTML. І тільки тоді компілятор, у даному випадку це React, починає все збирати. Поки виконуються ці три пункти, користувач бачитиме білий екран. Facebook, Gmail, Trello, Quora, Slack та багато інших великих компаній використовують Client Side Rendering. Схему роботи CSR відображено на рисунку 4.1.

Плюси CSR:

- менше навантаження на сервер;
- відмінно підходить для веб-застосунків;
- швидкий рендер після першого завантаження;
- швидка навігація.

Останні два пункти можна об’єднати. Завантаження бандлів та скриптів відбувається лише один раз. І після того, як це все завантажилося, не потрібно буде робити зайвих запитів, щоб перейти на іншу сторінку. Тобто навігація працюватиме набагато швидше, ніж при рендеру на стороні сервера.

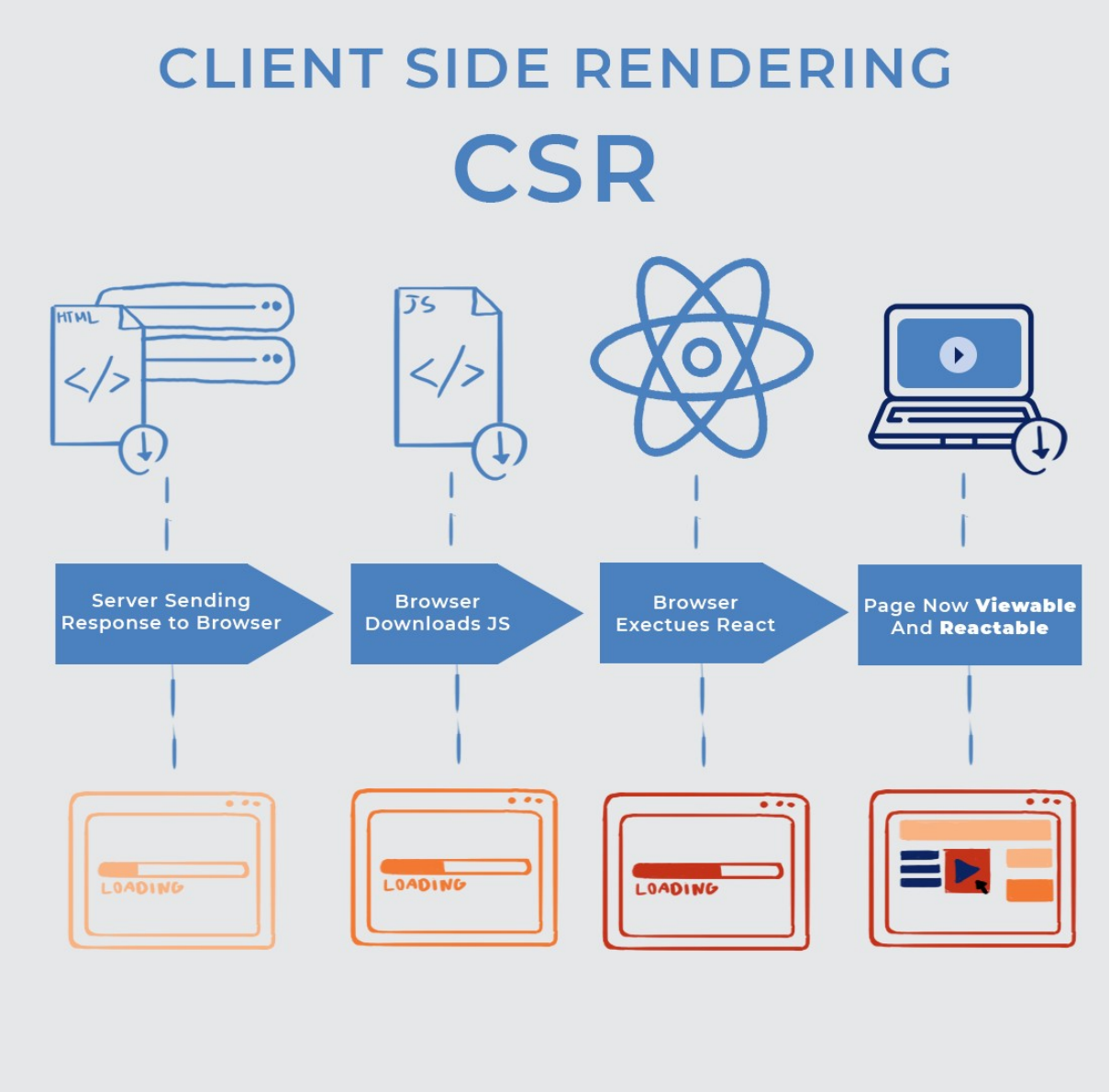



Рисунок 4.1 – Схема роботи Client Side Rendering

Мінуси CSR:

- повільне перше завантаження. «Повільне» - це досить відносний опис, але сьогодні навіть декілька секунд очікування користувачем можуть стати критичними;
- при клієнтському рендерінгу додатку необхідно додатково звертатись до API сервера для відображення, тому веб-скраперам (роботи, що витягують дані з сайту та надають їх веб-краулерам) потрібно запускати скрипти. Але не

всі пошукові систему вміють працювати з js, тому можуть прорахувати та проіндексовати сторінку неправильно;

- SEO – у разі неправильної індексації або наявності неправильної інформації у браузері, пошукові системи не зрозуміють сайт, і просто не виставлятимуть у топ пошуковика;
- завантаження сторінки, шаблону та стилів сильно навантажуючи техніку користувача.

### 4.1.2 Static Site Generation

SSG – генерація статичного сайту. Підхід «статичного» Інтернету, коли сервер віддавав простий HTML і CSS, переосмислений відповідно до сучасних вимог. Коли браузер запитує на розмітку, сервер в момент збірки додатку генерує веб-сайт як статичні файли і віддає готову сторінку. Різниця зі старим підходом у тому, що замість чистих HTML і CSS для створення додатків використовуються сучасні інструменти на кшталт React, Vue чи Angular.

Проблема у тому, що такий підхід не передбачає динамічну роботу з сервером. Тобто з SSG неможливо створити, наприклад, кошик інтернет-магазину, оскільки інформація, що відображається в кошику, повинна бути унікальна для кожного користувача. У такому випадку потрібно обирати SSR або CSR. Але статична генерація сайтів зручна, наприклад, для сторінки "Доставка", або картки товару. Тобто для інформації, яка ідентична для всіх користувачів.

Плюси SSG:

- SEO - можливість настроїти пошукову оптимізацію;
- більша продуктивність ніж у CSR, оскільки дані згенеровано під час збірки;
- дуже дешево, адже при використанні статичної генерації сайтів, сервер не потрібний. Достатньо завантажити код свого сайту на github і зібрати

його, наприклад, за допомогою Netlify. У такому разі доведеться заплатити лише за доменне ім'я;

- немає проблем з безпекою, адже у статичного сайту немає сервера.

Мінуси SSG:

- SSG складніше, ніж CMS. При початковому налаштуванні більшість CMS надає величезну кількість підказок для розгортання сайту. У разі роботи зі статичним генератором сайтів доведеться вивчати документацію;

- через те, що відобразити можна лише статичні дані, немає можливості створювати складні веб-додатки.

### 4.1.3 Server Side Rendering

SSR дозволяє отримати доступ до всіх необхідних даних для побудови сторінки на сервері. Повністю готова сторінка відправляється у браузер і відразу відображається. Це допомагає веб-застосункам завантажуватись за менший час і підвищує швидкість відгуку. Схему серверного рендерінгу зображено на рисунку 4.2.

Плюси SSR:

- SEO. Як і у випадку з SSG, пошуковий робот отримує всю необхідну інформацію;

- First Contentful Paint. Завдяки SSR сторінки сайту швидше стають доступними для взаємодії. Маючи продуктивний сервер, SSR може дати чудову оцінку First Contentful Paint, що покращує UX і SEO сторінки.

Мінуси SSR:

- уповільнюється час переходу між сторінками. Для початкового рендеру SSR швидше, ніж CSR, але при подальшій роботі з додатком доводиться рендерити дані двічі, один раз на сервері та один раз на клієнті;

- більш складна технологія. Написати програму тільки на React у зв'язці, наприклад, з Redux, значно простіше, ніж додати до цього стеку ще й

технологію SSR. Відповідно, збільшується кількість коду, складність розробки та з'являються додаткові бібліотеки;

- кешування. Для часткового закриття першого мінуса та на додаток до другого, додається більш складне кешування даних;

- вартість. Окрім підвищеної складності розробки, підходу SSR потрібен сервер, на відміну від CSR, і це робить продукт дорожче.

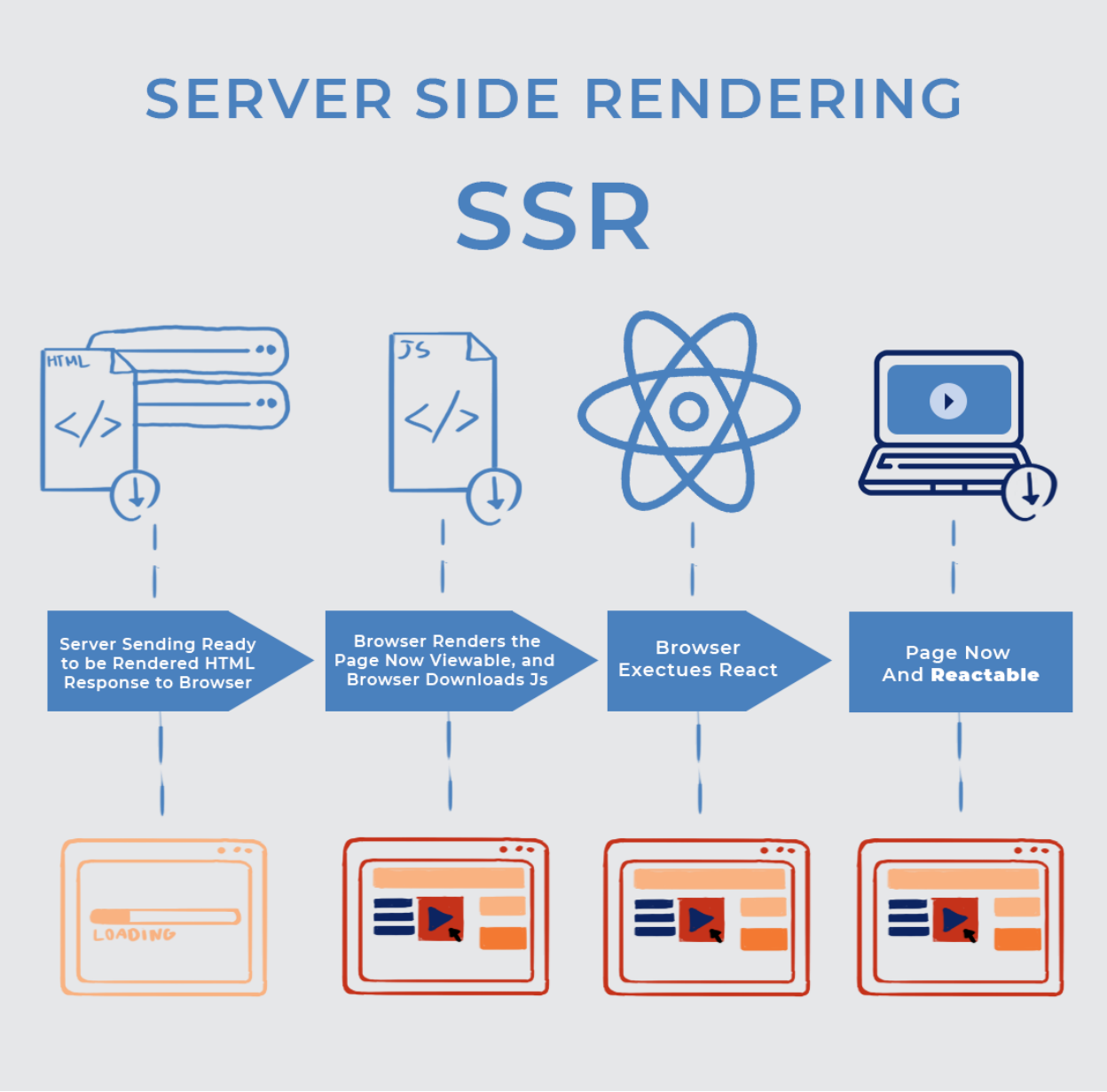



Рисунок 4.2 – Схема роботи Server Side Rendering

Але задача сьогодні полягає саме у тому, щоб розробити складний веб-додаток з динамічними даними, використовуючи сучасні інструменти, і не втративши SEO-оптимізації. За допомогою поєднання плюсів з кожного підходу необхідно створити «універсальний рендерінг». Потрібен фреймворк для React, який допоможе створювати додатки з підтримкою Typescript, у підходах SSG, SSR, CSR, з передзавантаженням роутів, і, бажано, з мінімальними налаштуваннями конфігурації. В більшості випадків мінімальний набір можливостей та властивостей сучасного додатка виглядатиме так:

- швидкі та плавні переходи між сторінками, як у CSR;
- можливість роботи з SEO;
- швидкий First Contentful Paint;
- робота з динамічними даними;
- зменшення витрат, пов'язаних із вмістом сервера.

## 4.2 Next.js

### 4.2.1 Огляд можливостей фреймворку

Фреймворк Next.js створений у 2016 році компанією Vercel (раніше Zeit) [43]. Основним його завданням є робота з Server Side Render-додатками, написаними на React. Її можна виконувати і самостійно, наприклад, за допомогою ReactDOMServer та Express.js, але це не оптимальний спосіб, тому що розробник у будь-якому випадку пише багато boilerplate-коду. Але Next.js більш оптимізований під такі завдання, та виводить розробку SSR-програм на наступний рівень і розбавляє її різними оптимізаціями.

Згідно офіційної документації, робота Next.js дотримується шести основних принципів:

1) Робота без налаштування. Використання файлової системи як API

2) Лише JavaScript. Все є функціями
3) Автоматичний Server Side Rendering та code-splitting
4) Механізм отримання даних визначається розробником
5) Передзавантаження для збільшення продуктивності
6) Простий деплой та розгортання

На сьогодні проект має багату аудиторію, в його розробку та оптимізацію зробив внесок Google, а технологією користуються великі компанії на кшталт Uber, Netflix, GitHub та інших. І навіть у самій документації React його включили як один із тул-чейнів для розробки.

Створити новий додаток на Next.js можна за допомогою готового тулкіту: npx create-next-app my-app. Npx дозволяє запускати виконувані npm-пакети без попереднього встановлення. Ця команда створить новий додаток у папці my-app та встановить усі необхідні залежності. Next.js "з коробки" підтримує eslint і CSS-модулі. Статичні файли можна складати в папку public, як і в звичайному React.

Окремо слід зауважити про папку pages. Кожен файл усередині цієї папки (крім папки api, файлів _app та _document) сприймається Next.js як окрема сторінка. Так Next.js автоматично конвертує файлову структуру у URL, включаючи динамічні параметри. На рисунку 4.3 зображено структуру папки pages та підписи динамічних параметрів, які буде конвертовано з наданих файлів.

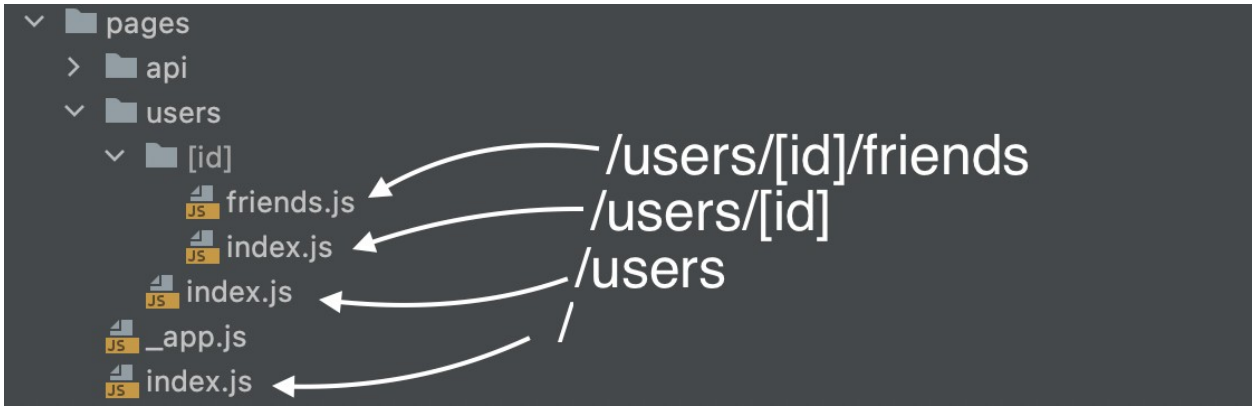


Рисунок 4.3 – Динамічна конвертація файлової структури

Для генерації вхідних параметрів props Next.js пропонує два варіанти: getServerSideProps та getStaticProps. GetServerSideProps буде викликано на кожен запит під час роботи програми. Всередині можна робити виклики до зовнішніх сервісів та API, щоб отримати останню актуальну інформацію. Залежно від того, заходить людина на сайт вперше або просто переміщається між сторінками, цей виклик буде зроблено або на сервері або на клієнті. GetStaticProps буде викликана один раз при збірці програми, вона підготує готові HTML-файли [44].

Наприклад, якщо є список статей, які рідко оновлюються, їх можна заздалегідь відрендерити за допомогою getStaticProps, і ці сторінки швидше завантажуватимуться, тому що вся інформація буде вже у них. Обидві функції потрібно експортувати саме з іменованим експортом, щоб Next.js їх побачив та використав: «export async function getServerSideProps(context) {…}».

Якщо на сторінці немає динамічного контенту, вона не довантажує дані з сервера і просто містить статичний контент, то Next.js оптимізує її та підготує готовий HTML-файл, який просто віддаватиметься клієнту. Через це у директорії pages не можна зберігати безпосередньо компоненти. Next.js сприйматиме їх як окремі сторінки і намагатиметься оптимізувати, що призведе до більшого часу складання. Тому pages не зберігають нічого, крім простих компонентів, що імпортують контент з іншої папки.

Одна з відносно нових можливостей Next.js – використання каталогу API. Це дозволяє описувати методи, які запускатимуться лише на сервері. Строго кажучи, це шар між фронтендом і зовнішніми сервісами, який можна легко реалізувати в проекті. URL для доступу до API будується за тим же принципом, як і URL для сторінок. Структура, зображена на рисунку 4.4, формує шлях api/banks/[id], до якого можна робити запити з фронтенда. На

виході очікується дефолтний експорт функції, яка оброблятиме запити та повертатиме відповіді. Всередині неї може бути все, що завгодно.

Так як цей метод виконується лише на сервері, можна безпечно звертатися до зовнішніх сервісів та змінних оточення, не показуючи зайву інформацію на клієнті. Але слід пам’ятати, що це лише маленький шар між фронтендом та різними API, тому повноцінний бекенд тут будувати не варто.

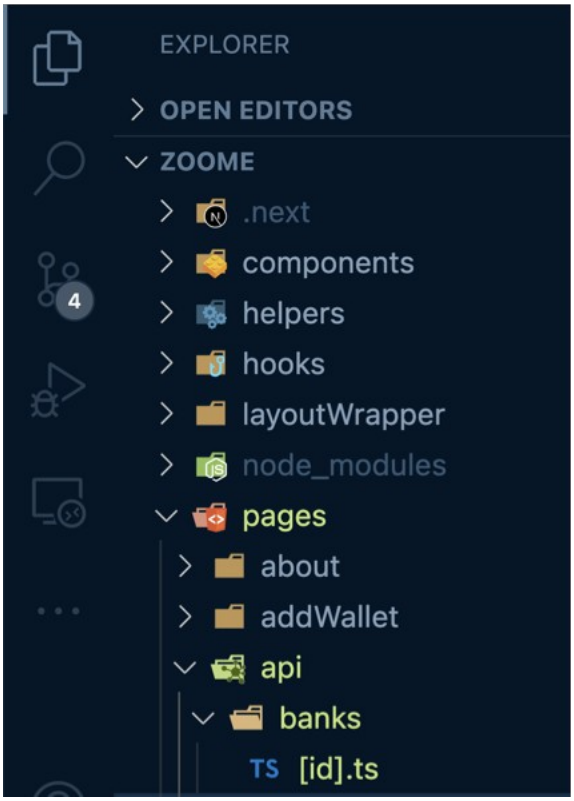



Рисунок 4.4 – Структура pages/api

Лістинг 4.1 – Код файлу pages/api/banks/[id].ts

```
const banks = {
   "1": { name: "MonoBank", description: "MonoBank", id: "..." },
   "2": { name: "PrivatBank", description: "Privat 24", id: "..." },
   "3": { name: " OschadBank", description: "OschadBank.ua", id: "..." },
}
```

```
export default function handler(req, res) {
   const { id } = req.query;
   res.status(200).json(banks[id]);
}
```

Файли _app та _document використовуються для кастомізації сторінок. Компонент _app потрібний для побудови загальної логіки між усіма сторінками. Наприклад, сюди можна підключити Redux, сервіси автентифікації, різні кеші та інше. Якщо всередині _app використовується getServerSideProp, то всі сторінки починають оброблятися в runtime - і це відключає статичну оптимізацію. NextJS одразу про це повідомить. Файл _document потрібен для редагування HTML-шаблону, який використовується для рендеру сторінок. Це може бути потрібно для підключення css-in-js або рендеру різних статичних блоків html, які завжди будуть у шаблоні сторінки.

Next.js надає функції та компоненти, потрібні для різних випадків. До найпоширених з них можна віднести:

- next/router – модуль для роботи з роутером Next.js. Потрібен для програмних редиректів отримання даних з URL на клієнта.
- next/link - компонент Link, який рендерить "правильні" посилання. У HTML-розмітці вони будуть звичайними тегами <a> і спрацюють, навіть якщо JS у користувача вимкнено. Якщо JS увімкнено, то на кліку відбудеться перехід за допомогою клієнтського рендеру та оптимізації Next.js.
- next/head — компонент Head, який дозволяє змінювати контент тега <head>без огляду на серверний/клієнтський рендер.
- next/script — компонент Scriptдля підключення зовнішніх скриптів клієнта (наприклад, метрик).

### 4.2.2 Порівняння з React

Головна відмінність Next.js від «чистого» React – у способі рендеру кінцевих веб-сторінок. Якщо React завантажує мінімальний HTML і найчастіше великий бандл JS (іноді розділений між сторінками на модулі), Next.js використовує Server Side Rendering — формування початкового HTML на сервері, використовуючи той самий React. У процесі він оптимізує бандли так, щоб не відправляти одразу на клієнтську сторону більше коду, ніж дійсно необхідно для конкретної сторінки, а довантажувати його пізніше. Порівняємо процес, що відбувається після запиту сторінки, покроково, для Next.js та React.

Next.js:

1) Браузер запитує сторінку з інформацією про товар, наприклад, /product/1.

2) Сервер отримує запит, завантажує необхідні дані про товар з іншого сервера, формує HTML на основі отриманих даних і необхідних компонентів React.

3) Браузер відразу отримує HTML з потрібною інформацією та показує його користувачеві, але JS для інтерактивності ще не завантажено.

4) У фоновому режимі відбувається дозавантаження JS, після чого вбудовується вже в наявний HTML-код – цей процес називається hydration.

React:

1) Браузер запитує сторінку з інформацією про товар, наприклад, /product/1.

2) Сервер повертає мінімальний HTML-файл, у якому прописаний імпорт JS-файлу, що використовує React.

3) Після завантаження HTML починається завантаження JS.

4) Тільки після завантаження JS-файлу відбувається створення необхідних DOM-елементів, завантаження даних із сервера та відображення корисного контенту.

Порівнявши обидва процеси, можна вивести такі ключові відмінності Next.js від React:

Плюси Next.js:

- При використанні Next.js браузер одразу отримує готовий HTML із необхідною інформацією, не чекаючи завантаження JS.
- Next.js робить запити до зовнішніх API на стороні сервера.

Мінуси Next.js:

- Next.js завантажує JS після HTML, до цього сторінка залишатиметься не інтерактивною. Але навіть так Next.js завантажить спочатку мінімально необхідний JS, а потім уже весь інший.
- Потрібно більше серверних ресурсів, оскільки Next.js формує HTML на стороні сервера перед надсиланням його клієнту.

Ще одним відносним недоліком Next.js іноді називають той факт, що це самостійний фреймворк, тобто він має певний метод і набір інструментів, яких потребує від розробників при створення додатків. Однак ці потреби не виходять за межі сучасних вимог у веб-розробці, проте можливості Next.js цілком задовольняють більшості проектів.

## 4.3 Приведення існуючого класового додатку у відповідність до вимог ФП

Виконаємо практичну реалізацію зміни програмної парадигми та інкапсуляцію ФП у класовий додаток. Для наочності в якості прикладу обрано фінансовий помічник Zoome з попередньої роботи [45]. Архітектуру, структуру, логіку додатка та процес створення класової імплементації описано в пояснювальній записці роботи. Оглянемо деякі головні, з точки зору роботи додатку, сторінки та зміну їх кодової бази. Це сторінки авторизації, категорій та гаманців.

### 4.3.1 Сторінка Login

Загалом форми авторизації – це стандартна і проста річ, коли мова йде про два-три поля. Але при створенні якоїсь «особливої» форми, - наприклад, покрокової, чи з динамічним підбором даних, чи з десятками полів, тощо, - будь-яке відхилення від прийнятих стандартів щодо поведінки форм змушує писати купу зайвого коду. Додати поле, додати обробник, додати стан, додати типізацію стану, - і так кожного разу для нової суттєвості. Та на щастя, функціональний підхід передбачає не тільки написання кастомних хуків, але і використання зручних рішень від розробників програмного забезпечення. Наданий бібліотекою react-hook-form хук useForm дозволяє швидко і зрозуміло обробляти будь-яку кількість полів форми. Створений компонент FormInput допомагає винести додаткову логіку оброки полів в окреме місце та інкапсулювати внутрішній стан. Такий підхід забезпечує безболісне масштабування форм за потреби. Досить тільки додати назву необхідного поля і власні ж предикати.

Поточні зміни версій компонента Login зображено на рисунку 4.5. Використання об’єкту history замінено роботою хука useRouter в купі з обгорткою withRouter. Як можна зрозуміти з назви, це дозволяє використовувати усі можливості роутінгу додатку лише за пару строчок коду. Окрім запровадженої авторизації за провайдерами, додано форму з окремою парою пошти та паролю.

Лістинг 4.2 – код файлу pages/login/index.tsx

```
const callSignIn = ( providerName: keyof typeof providers ) => () => {
  firebase.auth()
    .signInWithPopup( providers[ providerName ] )
    .then( ( result ) => {
        const token = result?.credential?.accessToken;
        const user = result.user;
        if ( token && user ) router.push( '/wallets' );
```

```
        } )
        .catch( function( error ) {
                const errorCode = error.code;
                const errorMessage = error.message;
                const email = error.email;
                const credential = error.credential;
                console.error( 'error', {
                        errorCode,
                        errorMessage,
                        email,
                        credential
                } );
        } );
};
```

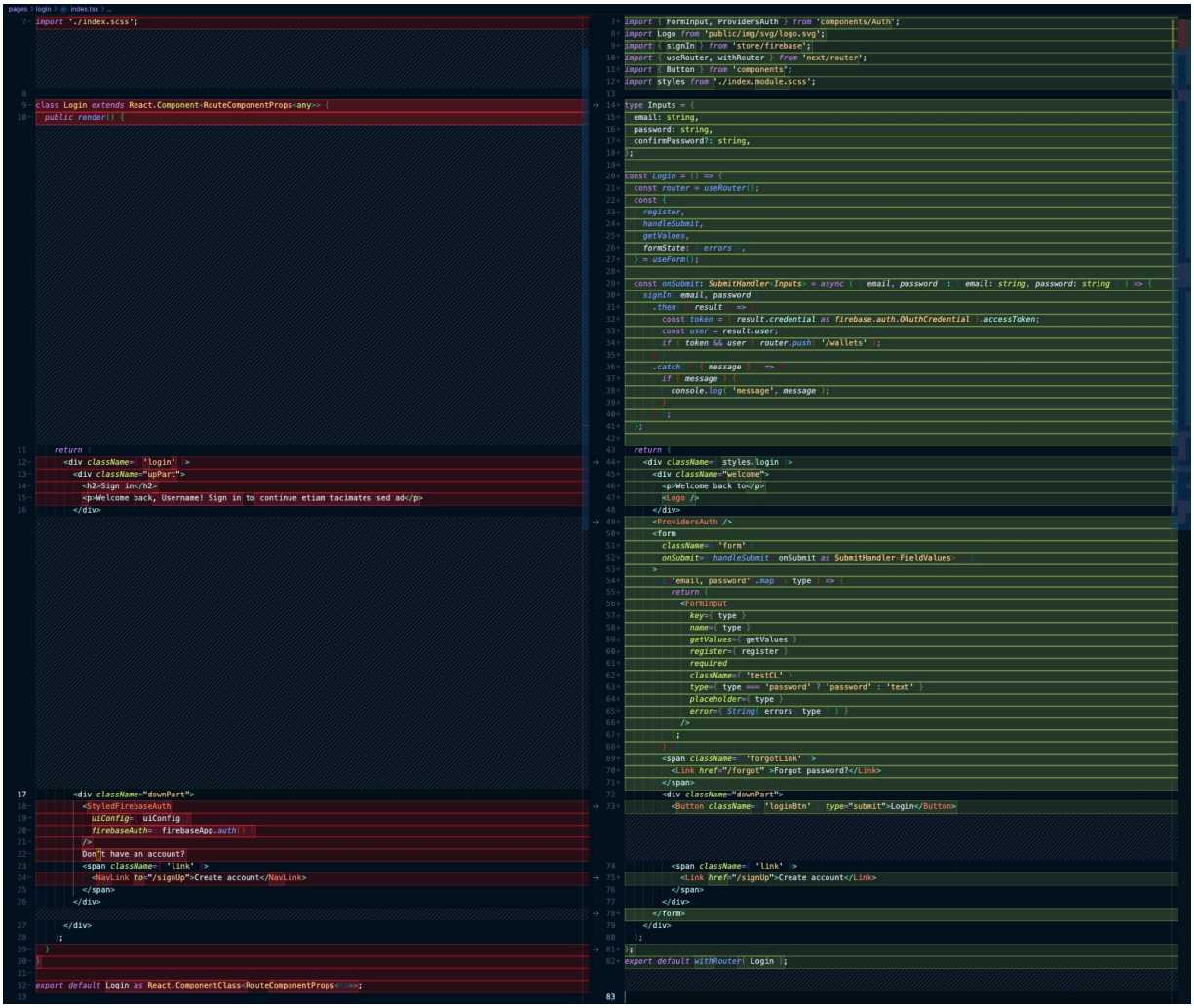


Рисунок 4.5 – Зміни компонента сторінки авторизації

### 4.3.2 Сторінка Categories

У компоненті категорій також було змінено деякі структури. Наприклад, стилі, згідно до вимог фреймворку Next.js, мають бути модульними, тому і викликаються в якості конструкта з властивостями. Для зручності відносні імпорти було замінено абсолютними, а деякі компоненти винесено на окремий рівень у папку common. Порівняння з класовою реалізацією відображено на рисунку 4.6.

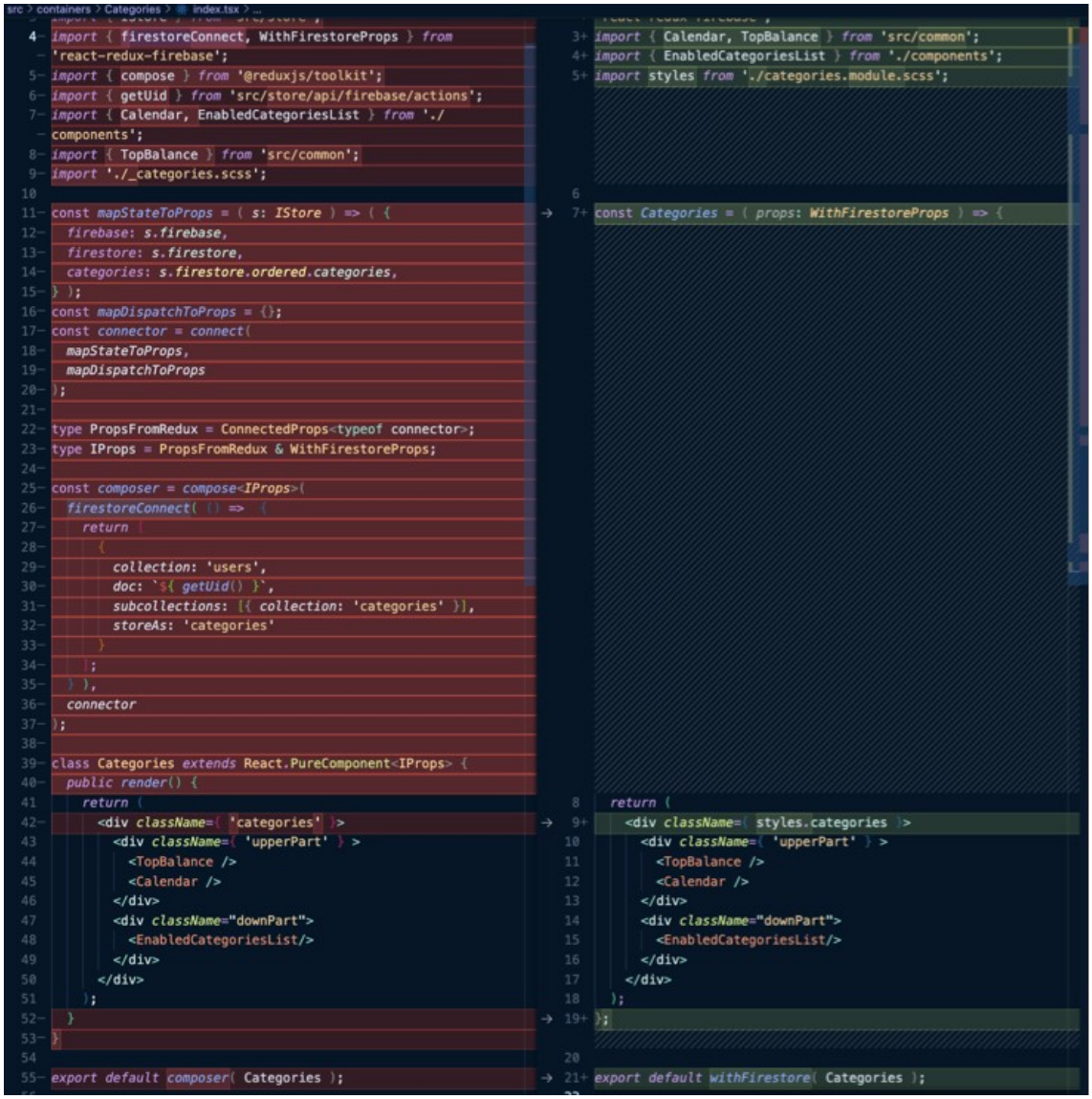


Рисунок 4.6 – Порівняння класової та функціональної реалізації компонента категорій

Можна помітити, що перехід з класового до функціонального стилю дозволив спростити код з 55 строк до 21. Це дуже багато. Але чи завжди можлива така економія та спрощення кодової бази? Нажаль, ні. Згідно до документації проекту Zoome, сторінки категорій та гаманців можуть бути відкриті за замовчанням в якості головної сторінки додатку. А як було доведено у підрозділі 4.1, для побудови сторінки на сервері Next надає SSR, який дозволяє отримати доступ до всіх необхідних даних. Тому оглянемо функціональну реакт-версію компоненту категорій, а перехід компоненту

гаманців з класового до функціонального підходу проведемо згідно до вимог фреймворку Next.js.

Обгортання класу у композицію через функцію firestoreConnect замінено ХОКом withFirestore, який робить те саме під капотом. Можна помітити схожість підходу з роутінгом з попереднього прикладу. Розробники Firestore додали такі хуки у нові версії саме для роботи з функціональними компонентами React. На рисунку 4.7 зображено компонент Categories.tsx у функціональному стилі та згенерований TypeScript опис роботи HOC withFirestore.

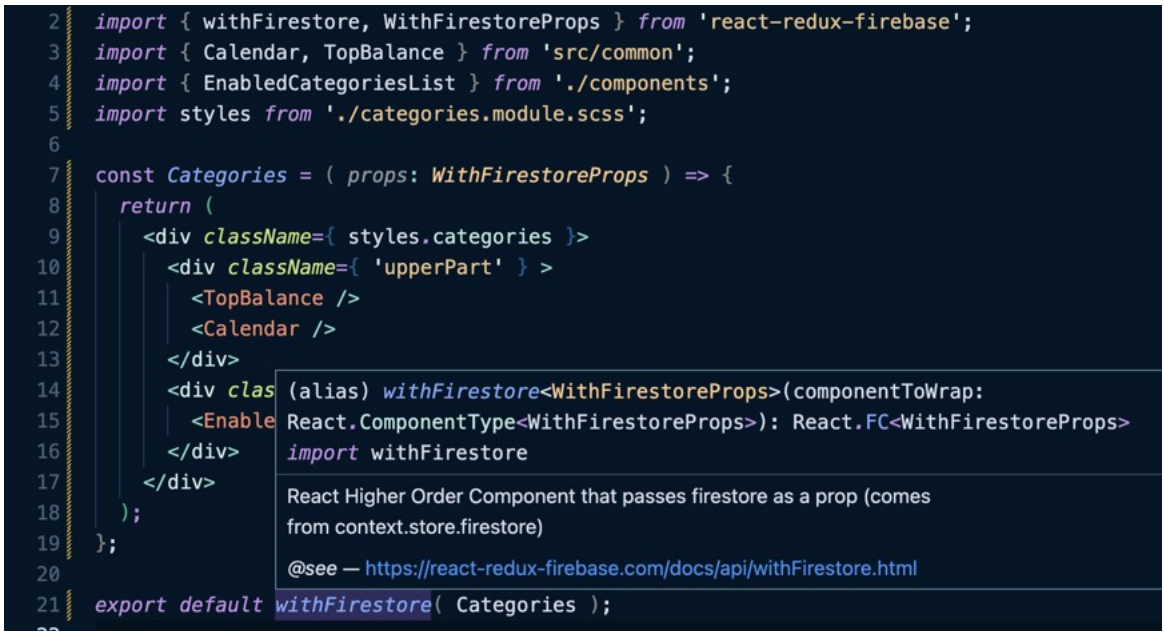
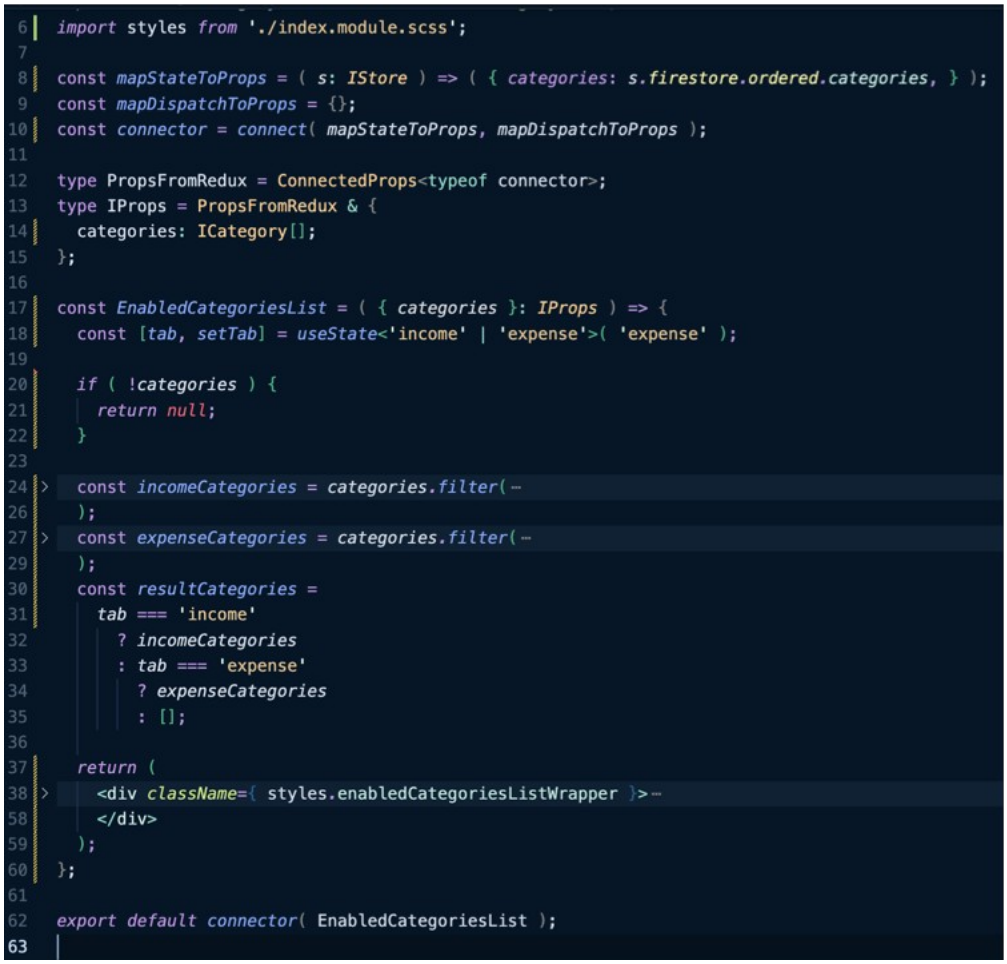



Рисунок 4.7 – Опис withFirestore

Компонент EnabledCategoriesList може працювати і далі у вигляді класу до тих часів, коли логіка додатку не потребує його оновлення та використання якихось хуків. Але перфекціоністам та шанувальникам консистентного коду ніхто не забороняє зробити цей перехід. Тоді код виглядатиме так, як показано на рисунку 4.8



```
import styles from './index.module.scss';

const mapStateToProps = ( s: IStore ) => ( { categories: s.firestore.ordered.categories, } );
const mapDispatchToProps = {};
const connector = connect( mapStateToProps, mapDispatchToProps );

type PropsFromRedux = ConnectedProps<typeof connector>;
type IProps = PropsFromRedux & {
  categories: ICategory[];
};

const EnabledCategoriesList = ( { categories }: IProps ) => {
  const [tab, setTab] = useState<'income' | 'expense'>( 'expense' );

  if ( !categories ) {
    return null;
  }

  const incomeCategories = categories.filter(…
  );
  const expenseCategories = categories.filter(…
  );
  const resultCategories =
    tab === 'income'
      ? incomeCategories
      : tab === 'expense'
        ? expenseCategories
        : [];

  return (
    <div className={ styles.enabledCategoriesListWrapper }>…
    </div>
  );
};

export default connector( EnabledCategoriesList );
```

Рисунок 4.8 – Компонент EnabledCategoriesList.tsx у функціональному стилі

Лістинг 4.3 - код файлу components/EnabledCategoriesList.tsx

```
const mapStateToProps = ( s: IStore ) => ( { categories:
s.firestore.ordered.categories, } );
const connector = connect( mapStateToProps, {});
type PropsFromRedux = ConnectedProps<typeof connector>;
type IProps = PropsFromRedux & {
  categories: ICategory[];
};

const EnabledCategoriesList = ( { categories }: IProps ) => {…};
export default connector( EnabledCategoriesList )
```

### 4.3.3 Сторінка Wallets

Як було доведено, Next.js надає можливості повернути збірку додатку як початкову сторінку з сервера. Для цього серверу потрібно сформулювати оточення та вхідні параметри для компонента. Тому передача даних з БД, в даному випадку це Firebase, має відбуватися не у функції compose(), а у функції getServerSideProps(). Класову та функціональну версії компонента гаманців зображено на рисунках 4.9 та 4.10.


Рисунок 4.9 – Класова реалізація компонента гаманців

```
  return (
    <LayoutWrapper>
      <div className={ styles.wallets }>
        <div className={ 'upperPart' } >
          <TopBalance wallets={ wallets } />
          <TypeSwitcher currentType={ currentType } updateType={ setCurrentType }/>
        </div>
        <div className="downPart">
          <ActiveTypeWallets wallets={ wallets } currentType={ currentType }/>
        </div>
      </div>
    </LayoutWrapper>
  );
};

export const getServerSideProps = async ( ctx: GetServerSidePropsContext ) => {
  try {
    const cookies = nookies.get( ctx );
    const token = await firebaseAdmin.auth().verifyIdToken( cookies.token );
    const { uid, email } = token;
    const wallets: IWallet[] = getArrayFromCollection(
      await firebaseClient.firestore().collection( 'users' ).doc( uid ).collection( 'wallets' ).get()
    ) as IWallet[];

    return {
      props: {
        message: `Your email is ${email} and your UID is ${uid}.`,
        wallets: wallets
      },
    };
  } catch ( err ) {
    return {
      redirect: {
        permanent: false,
        destination: '/login',
      },
      props: {} as never,
    };
  }
};

export default Wallets;
```

Рисунок 4.10 – Функціональна реалізація компонента гаманців

У функції getServerSideProps спочатку беруться cookies, з них витягується та верифікується токен, та за унікальним id користувача відбувається сбір необхідного набору даних з колекції Firebase. Тепер при першому запиті до серверу, якщо головною сторінкою позначено гаманці, Next проведе ці операції, згенерує початкову сторінку та відправить до клієнтської частини. Подальші зміни буде відпрацьовано на стороні клієнта, як у випадку стандартного SPA.

Коду скорочено вже не так багато, близько 10 строчок. Але завдяки Next реалізовано оптимізовану з точки зору SEO сторінку, яку краще будуть індексувати пошукові роботи. LayoutWrapper – це компонент вищого порядку, який додає до будь-якого переданого контенту хедер і футер, та налаштовує передачу метаданих у тег <Head/>. Код компонента зображено на рисунку 4.11.

```
15  const LayoutWrapper = ( { children, title, description, keywords, ...rest }: ILayoutProps ) => {
16
17    return (
18      <div className={ styles.wrapper }>
19        <Head>
20          <title>{title}</title>
21          <meta name="description" content={ description } />
22          <meta name="keywords" content={ keywords } />
23        </Head>
24        <Header classname={ styles.header }/>
25        <Sidebar classname={ styles.sidebar }/>
26        <div className={ styles.body }>
27          {children}
28        </div>
29        <Footer classname={ styles.footer } />
30      </div>
31    );
32  };
33
34  LayoutWrapper.defaultProps = {
35    title: 'Zoome',
36    description: 'Personal finansial assistant',
37    keywords: 'money, savings, assistant',
38  };
39  export default LayoutWrapper;
```

Рисунок 4.11 – Код LayoutWrapper HOC

Лістинг 4.4 – Код файлу pages/wallets/index.tsx

```
const Wallets = ({ wallets }: InferGetServerSidePropsType<typeof
getServerSideProps> ) => {…};

export const getServerSideProps = async ( ctx: GetServerSidePropsContext ) => {
  try {
    const cookies = nookies.get( ctx );
```

```
    const token = await firebaseAdmin.auth().verifyIdToken( cookies.token );
    const { uid, email } = token;
    const wallets: IWallet[] = getArrayFromCollection(
      await
firebaseClient.firestore().collection( 'users' ).doc( uid ).collection( 'wallets' ).get()
    ) as IWallet[];

    return {…};
  } catch ( err ) {…}
};

export default Wallets;
```

### 4.4 Висновки розділу

В даному розділі оглянуто існуючі підходи до генерації шаблонів додатків та сайтів від SSR до CSR, і знов до SSR. Розібрано плюси і мінуси статичної генерації сторінок, рендерінгу на клієнтській та серверній стороні. Виявлено перелік необхідний для сучасних додатків можливостей та властивостей, зібраний з плюсів різних підходів.

Проведено розбір роботи фреймворку Next.js та реалізовано перехід існуючого класового додатку у відповідність до вимог функціональної парадигми. Next.js – це заснований на React фреймворк, призначений для розробки веб-застосунків, що мають функціонал, виходячий за рамки SPA, тобто так званих односторінкових додатків. Як відомо, основним недоліком SPA є проблеми з індексацією сторінок таких програм пошуковими роботами, що негативно впливає на SEO. Останнім часом ситуація змінюється на краще. Вже існують спеціальні інструменти, що дозволяють перетворити React-SPA на багатосторінник шляхом попереднього рендерингу

програми на статичну розмітку, та вбудови метаінформації в head. Однак Next.js істотно спрощує процес розробки не тільки SPA, а й багатосторінкових та гібридних додатків.

Плюси Next.js:

- Завдяки вбудованому рендерінгу на стороні сервера, програми Next.js завантажуються значно швидше, ніж програми React;
- Підтримує функцію експорту статичних сайтів;
- Швидкий вхід для тих, хто вже працював із бібліотекою React.js;
- Автоматичний поділ коду для сторінок;
- Легко створювати внутрішні API-інтерфейси за допомогою вбудованих маршрутів API та створювати кінцеві точки API;
- Вбудована підтримка маршрутизації сторінок, CSS, JSX та TypeScript;
- Швидке додавання плагінів для налаштування Next.js відповідно до потреб сторінки;
- Висока популярність.

Використання функціональної парадигми та NextJS в якості фреймворка, що підтримує функціональний підхід, визнано доцільним. Було проведено перехід класового компонента на функціональний стиль в залежності від потреб оточення та використання компонента. На прикладі головних компонентів існуючого додатку фінансового помічника Zoome оглянуто переходи з класового стилю до функціонального для React та NextJS.

# ВИСНОВКИ

В ході даної роботи було проведено дослідження на задану тему та покроково виконано завдання. У першому розділі проаналізовано предметну область та проведено огляд найбільш популярних парадигм програмування: імперативну та декларативну. Було проведено порівняння функціональної парадигми з ООП з установленням спільних та різних властивостей.

Озброївшись інкапсуляцією, успадкуванням та поліморфізмом, ООП дарувало надію на спрощення процесу створення нових програм. Спільнота розробників ПЗ надовго визнала методологію ООП істиною в останній інстанції, і з появою Java ця практика лише закріпилася. Проте недоліки методу з часом ставали дедалі очевиднішими. Жорсткі ієрархії об'єктів та відсутність чітких характеристик класів помітно затримували еволюцію ООП. Поступово невизначеність поширилася на всі без винятку складові елементи програмування, отже, не обходиться без небажаної взаємосумісності і непередбачуваних побічних ефектів.

Функціональна парадигма програмування - це набір концепцій, правил та абстракцій, що визначають стиль функціонального програмування. Відповідно до них, функціональний підхід передбачає розбиття коду на функції та прагне до використання лише чистих функцій. Тобто цей стиль спрямований на більш зручний поділ частин програм. Це спрощує розуміння коду для людей, які його не писали. Чистота функції, коли її виклик просто повертає результат, а не пише і не посилає емейли на пошту, дуже допомагає в цьому. Чітко окреслені контракти та невеликі стейтлес модулі - найпростіші та зручніші в роботі з ними. Функціональний підхід лише розвиває цю ідею до логічної точки — всі функції мають бути чистими, і не залежати від будь-якого стану.

Виявлено, що дисципліни ФП та ООП не є взаємовиключними. Можна писати об'єктно-орієнтовані функціональні програми, і навпаки, принципи та

патерни ООП можуть використовуватись у функціональних програмах, якщо прийняти дисципліну «вказівників на функції». Не обов'язково робити весь сайт або додаток на ФП. У місцях з логікою можуть бути деякі комбіновані розрахунки, де ООП незручне. Доведено, що процес веб-розробки нині полягає в декомпозиції задачі на складові частини, створенні компонентів для вирішення цих складових частин та зворотній збірці в єдиному додатку. І щоб встигнути в стислі терміни надавати споживачам продукцію з майже необмеженими можливостями, треба робити це досить швидко. Набагато простіше впоратися з таким завданням, коли програми що розробляються можна умовно розділити на кілька чистих функцій, перевірити які не складає труднощів. У таких алгоритмах немає побічних ефектів та абстрактних формулювань, розрахованих на результати у глобальному масштабі.

У другому розділі проаналізовано розвиток та становлення мови JavaScript з огляду функціонального підходу. Наведено особливості сучасного використання функціональної парадигми та оглянуто механізми і можливості зміни програмної парадигми при інкапсуляції функціонального підходу у JavaScript. Було доведено, що JS – мультипарадигмова, тобто універсальна мова програмування, що дозволяє використовувати кілька парадигм та програмувати у різних стилях.

До списку речей, які притаманні деяким функціональним мовам, і яких умовно немає в JavaScript, частіше за все відносять чистоту та незмінність. У функціональних мовах іммутабельність зазвичай дана за замовчанням. Більшість ФП-мов використовують спеціальні структури даних, наприклад, префіксне дерево, з можливістю спільного використання даних. Тобто старий об'єкт і новий об'єкт зберігають посилання на ті самі дані, якщо вони не змінювалися.

Зазвичай зміна глобальних значень безпосередньо впливає на поточний стан програми, тоді як операції введення/виводу змінюють щось за межами програми. Але у веб-розробці все обертається довкола DOM. Це не тільки

доступи та зміна глобальних змінних, але ще й операції вводу/виводу. Виведено, що фронтенд можна сприймати як один суцільний побічний ефект і у екосистемі JavaScript код пишуть саме для побічних ефектів. Тому замість того, щоб повністю їх позбутися, потрібно зменшити їх кількість, ізолювати ті, що залишилися, в одному місці, а більшість функцій зробити чистими. Чисті функції – одна з найкорисніших і найзастосовніших методик ФП.

В третьому розділі було проведено огляд використання функціональної парадигми у роботі з бібліотекою React. Було порівняно функціональні компоненти React та компоненти, які базуються на класах з огляду практичного використання. Розібрано плюси та мінуси використання функціонального підходу при роботі за бібліотекою. Використання хуків дозволяє виділити логіку, яка керує побічним ефектом. Раніше цю логіку доводилося розбивати на різні методи життєвого циклу у компоненті. І оскільки з'явилися хуки мінімізації та React.memo, функціональні компоненти відтепер піддаються мемоізації, тобто компонент не буде перетворено або оновлено, якщо його пропси не змінилися. Також з'явився хук useMemo, який можна використовувати для того, щоб обчислювати якісь важкі значення, або інстанцувати якісь службові класи лише один раз.

За кілька останніх років React перетворився на чудовий швидкий клієнтський JS фреймворк. І хоча React це бібліотека, сучасна екосистема та велика кількість розширень та бібліотек дозволяють сприймати його саме як фреймворк. React, Redux та Immutable.js разом запропонували розробникам практичне вирішення питання поєднання функціонального програмування та JavaScript – розробку чистих редюсерів без побічних ефектів. Це дозволило змінювати звичний стан продукту для створення великомасштабних функціональних програм в рамках синтаксису ES6. Остання версія React надала спрощений синтаксис для чистих компонентів, інтеграція яких не викликає побічних ефектів. Популярність такого підходу зумовила поширене розповсюдження функціонального програмування.

У четвертому розділі роботи було оглянуто використання функціональної парадигми у роботі з сучасним фреймворком, та розібрано плюси і мінуси статичної генерації сторінок, рендерінгу на клієнтській та серверній сторонах. Велика частина уваги була приділена опису клієнтського та серверного підходів до генерації шаблонів додатків. Проаналізувавши переваги та недоліки, було зроблено висновок, що для створення якісного сучасного додатку, потрібно поєднати плюси кожного з підходів та позбавитись мінусів та знайдено рішення, яке надає таких можливостей - Next.js.

Основна перевага Next.js – вбудована підтримка SSR для підвищення продуктивності та SEO. Рендерінг на стороні сервера працює шляхом зміни потоку запитів React, так що всі компоненти, крім клієнта, відправляють свою інформацію на сервер. Next.js також дозволяє редагувати тег <head> сайту, чого неможливо зробити в React. Тег <head> є основною частиною метаданих веб-сторінки та сприяє підвищенню SEO-рейтингу сайту. Використання SSR також надає перевагу у SEO, що допомагає сайту займати більш високі позиції на сторінках результатів пошукових систем. SSR підвищує рейтинг веб-сайтів для SEO, тому що вони завантажуються швидше та більше контенту сайту можна сканувати за допомогою трекерів SEO.

Наразі навколо Next.js сформувалося велике та активне ком'юніті, а сам фреймворк активно підтримується та розвивається розробниками. Всі оновлення публікуються в блозі і детально описані в документації, поряд із повноцінним інтерактивним туторіалом. Останні оновлення мають адаптацію React 18, нові файлові конвенції, паралельні запити даних та багато іншого. Next.js активно використовують та підтримують великі компанії на кшталт Google, а Vercel пропонує зручну інфраструктуру для деплою додатків. Отже можна з упевненістю сказати, що Next.js буде і далі популярним у найближчому майбутньому.

Доведено, що функціональний підхід загалом та хуки зокрема надають досить великі можливості. Потрібно добре розуміти, як вони працюють, особливо те, як вони мемоізують значення. Адже з великими можливостями приходить відповідальність. Якщо неправильно щось замемоізувати, то програма зберігатиме неактуальні значення, або навпаки — ререндер відбуватиметься надто часто. Щоб не зіштовхнутися з ситуацією, коли якесь значення замкнулося всередині чи у нижніх рівнях ієрархії, треба чітко розуміти, що таке замикання, як вони працюють, як функції обробляють вільні змінні тощо. Без цього знання буде по-справжньому складно налагодити якусь проблему.

Написання «функціонального» коду дає можливість поглянути на проблему з іншого боку, де розробка рішення може бути більш ефективною. І це просто збільшує кількість засобів висловити ідеї програміста. Відповідно, функціональне програмування може змінити стиль написання коду на краще. Якщо існуюча кодова база вже не дає необхідного рівня управління складністю бізнес-області, код засмічено непотрібними залишками, а імплементація нового функціоналу призводить до рефакторингу та втрати часу, перехід на функціональне програмування може бути рішенням. А в областях, пов'язаних з великою кількістю обчислень або перетворень даних, паралельним чи асинхронним програмуванням, функціональне програмування надає значних переваг.

# ПЕРЕЛІК ПОСИЛАНЬ

1. Why functional programming matters, by John Hughes. 1989. – 128 с.

2. Хендерсон П. Функциональное программирование. Применение и реализация: Пер. с англ.—М.: Мир, 1983.—349 с.

3. Заяць В.М., Заяць М.М. З-411 Логічне і функціональне програмування. Системний підхід. - Рівне : НУВГП, 2018. - 422 с.

4. Кайл Симпсон. Функциональное программирование на JavaScript. Как улучшить код JavaScript-программ. – Диалектика, 2018. – 304 с.

5. Олександр Жеребцов, Антон Іжиков. «Використання функціонального програмування у JavaScript та фреймворках», 2022 International Conference on Innovative Solutions in Software Engineering (ICISSE), Vasyl Stefanyk Precarpathian National University, Ivano-Frankivsk, Ukraine, Nov. 29-30, 2022, pp. 162-169 с.

6. Іжиков А. І. «Застосування функціональної парадигми у веб-програмуванні», Збірник тез учасників XXIV науково-практичної студентської конференції Запорізького Інституту Економіки та Інформаційних Технологій. Секція 4 Інформаційних технологій. с. 72.

7. Парадигмы программирования: простое объяснение. [Електронний ресурс]. – Режим доступу: https://highload.today/paradigmy-programmirovaniya/#5 - 06.01.23

8. Declarative vs imperative programming: 5 key differences. [Електронний ресурс]. – Режим доступу: https://www.educative.io/blog/declarative-vs-imperative-programming - 06.01.23

9. What Is a Procedural Programming Language? [Електронний ресурс]. – Режим доступу: https://www.indeed.com/career-advice/career-development/procedural-programming-language - 06.01.23

10. 4 Principles of Object-Oriented Programming. [Електронний ресурс]. – Режим доступу: https://khalilstemmler.com/articles/object-oriented/programming/4-principles/ - 06.01.23

11. Difference Between Imperative and Declarative Programming. [Електронний ресурс]. – Режим доступу: https://www.geeksforgeeks.org/difference-between-imperative-and-declarative-programming/ - 06.01.23

12. Difference Between Functional and Logical Programming. [Електронний ресурс]. – Режим доступу: https://www.geeksforgeeks.org/difference-between-functional-and-logical-programming/ - 06.01.23

13. Персональна сторінка Brendan Eich. [Електронний ресурс]. – Режим доступу: https://brendaneich.com/ - 06.01.23.

14. JavaScript. [Електронний ресурс]. – Режим доступу: https://developer.mozilla.org/ru/docs/Web/JavaScript/ - 06.01.23.

15. 4 новейших операции JavaScript. [Електронний ресурс]. – Режим доступу: https://medium.com/nuances-of-programming/введение-4-новейших-операции-javascript-5d744c5fc336 - 06.01.23.

16. Функциональное программирование — это не то, что нам рассказывают. [Електронний ресурс]. – Режим доступу: https://habr.com/ru/post/479238/ - 06.01.23

17. Переклад документації Fantasy-Land. [Електронний ресурс]. – Режим доступу: https://medium.com/devschacht/спецификация-fantasy-land-bf81121b58cb - 06.01.23.

18. Сторінка документації Fantasy-Land. [Електронний ресурс]. – Режим доступу: https://github.com/fantasyland/fantasy-land - 06.01.23.

19. Understanding Monads With JavaScript. [Електронний ресурс]. – Режим доступу: http://igstan.ro/posts/2011-05-02-understanding-monads-with-javascript.html - 06.01.23.

20. TypeScript. [Електронний ресурс]. – Режим доступу: https://www.typescriptlang.org/ - 06.01.23.

21. Functional Programming In JavaScript. [Електронний ресурс]. – Режим доступу: https://medium.com/free-code-camp/functional-programming-in-js-with-practical-examples-part-2-429d2e8ccc9e#.xiuwfmjbj - 06.01.23.

22. Офіційна сторінка проекту JSDoc. [Електронний ресурс]. – Режим доступу: https://jsdoc.app/ - 06.01.23.

23. Tail call optimization in ECMAScript 6. [Електронний ресурс]. – Режим доступу: https://2ality.com/2015/06/tail-call-optimization.html - 06.01.23.

24. Замыкания в javascript. [Електронний ресурс]. – Режим доступу: https://htmlacademy.ru/blog/js/lets-learn-javascript-closures - 06.01.23.

25. Mostly adequate guide to FP (in javascript, translated). [Електронний ресурс]. – Режим доступу: https://github.com/MostlyAdequate/mostly-adequate-guide-ru - 06.01.23.

26. React. Офіційна документація. [Електронний ресурс]. – Режим доступу: https://reactjs.org - 06.01.23.

27. Redux. [Електронний ресурс]. – Режим доступу: https://react-redux.js.org/introduction/why-use-react-redux - 06.01.23.

28. React-Router. [Електронний ресурс]. – Режим доступу: https://reactrouter.com/en/main - 06.01.23.

29. Сторінка бета-документації проекту ReactJS. [Електронний ресурс]. – Режим доступу: https://beta.reactjs.org/ - 06.01.23.

30. How Facebook and Other Sites Manipulate Your Privacy Choices [Електронний ресурс]. – Режим доступу: https://www.wired.com/story/facebook-social-media-privacy-dark-patterns/ - 06.01.23.

31. The Morality Of A/B Testing. [Електронний ресурс]. – Режим доступу: https://techcrunch.com/2014/06/29/ethics-in-a-data-driven-world/ - 06.01.23.

32. How Are Function Components Different from Classes? [Електронний ресурс]. – Режим доступу: https://overreacted.io/how-are-function-components-different-from-classes/ - 06.01.23.

33. React hooks: get the current state, back to the future. [Електронний ресурс]. – Режим доступу: https://dev.to/scastiel/react-hooks-get-the-current-state-back-to-the-future-3op2 - 06.01.23.

34. React hooks — победа или поражение? [Електронний ресурс]. – Режим доступу: https://habr.com/ru/post/428317/ - 06.01.23.

35. Introducing Hooks. [Електронний ресурс]. – Режим доступу: https://reactjs.org/docs/hooks-intro.html - 06.01.23.

36. Розділ DOM на сайті Консорціуму Всесвітньої павутини (W3C) [Електронний ресурс]. – Режим доступу: https://dom.spec.whatwg.org/ - 06.01.23.

37. Типізація DefinitelyTyped. [Електронний ресурс]. – Режим доступу: https://github.com/DefinitelyTyped/DefinitelyTyped/ - 06.01.23.

38. This benchmark is indeed flawed. [Електронний ресурс]. – Режим доступу: https://medium.com/@dan_abramov/this-benchmark-is-indeed-flawed-c3d6b5b6f97f - 06.01.23.

39. Вихідний код методу withEffect бібліотеки reactorlib. [Електронний ресурс]. – Режим доступу: https://unpkg.com/@reactorlib/core@0.3.2/lib/effect/withEffect.js - 06.01.23.

40. Optimizing Performance. [Електронний ресурс]. – Режим доступу: https://reactjs.org/docs/optimizing-performance.html - 06.01.23.

41. Making Sense of React Hooks. [Електронний ресурс]. – Режим доступу: https://medium.com/@dan_abramov/making-sense-of-react-hooks-fdbde8803889 - 06.01.23.

42. What does CSR, SSR, SSG and ISR means and why should you care? [Електронний ресурс]. – Режим доступу: https://www.flavienbonvin.com/data-building-strategy-for-nextjs-app/ - 06.01.23.

43. Офіційна сторінка фреймворку Next.JS. [Електронний ресурс]. – Режим доступу: https://nextjs.org/ - 06.01.23.

44. SSR - SSG - ISR - CSR in Next.js-The Ultimate Guide. [Електронний ресурс]. – Режим доступу: https://dev.to/idrazhar/ssr-ssg-isr-csr-in-nextjs-the-ultimate-guide-256m - 06.01.23.

45. А. Іжиков. Бакалаврська дипломна робота «Розробка веб-застосунку «персональний фінансовий помічник» за допомогою технології прогресивних веб-додатків». Запоріжжя: ПрАТ «ПВНЗ «ЗІЕІТ». 2021 р. 145 с.

# ДОДАТОК А
# ВИХІДНИЙ ПРОГРАМНИЙ КОД

Повний код файлу тесту з використанням React-hooks:

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

const array = [];
for (let i = 0; i < 10000; i++) array[i] = true;

const Component = () => {
  const [a, setA] = useState('');
  const [b, setB] = useState('');
  const [c, setC] = useState('');
  useEffect(() => {
    setA('A');
    setB('B');
    setC('C');
  }, []);
  return <div>{a + b + c}</div>;
};

const Benchmark = ({ start }) => {
  useEffect(() => {
    console.log(Date.now() - start);
  });
  return array.map((item, index) => <Component key={index} />);
};
```

render(<Benchmark start={Date.now()} />, document.getElementById('root'));

Повний код файлу тесту з використанням React-HOC та бібліотеки reactorlib:

```
import React from 'react';
import { render } from 'react-dom';
import { compose, withState, withEffect } from '@reactorlib/core';

const array = [];
for (let i = 0; i < 10000; i++) array[i] = true;

const _Component = ({ a, b, c }) => {
  return <div>{a + b + c}</div>;
};

const Component = compose(
  withState({
    a: '',
    b: '',
    c: ''
  }),
  withEffect(({ setA, setB, setC }) => {
    setA('A');
    setB('B');
    setC('C');
  }, true)
)(_Component);
```

```
const _Benchmark = () => {
  return array.map((item, index) => <Component key={index} />);
};

const Benchmark = compose(
  withEffect(({ start }) => {
    console.log(Date.now() - start);
  })
)(_Benchmark);

render(<Benchmark start={Date.now()} />, document.getElementById('root'));
```

Файл «pages/login/index.tsx»:

```
import * as React from 'react';
import firebase from 'firebase/compat/app';
import 'firebase/compat/auth';
import 'firebase/compat/firestore';
import Facebook from 'public/img/svg/facebook1.svg';
import Google from 'public/img/svg/google1.svg';
import Twitter from 'public/img/svg/twitter1.svg';
import { useRouter } from 'next/router';
import { providers } from 'firebaseClient';
import styles from './index.module.scss';

export const ProvidersAuth = () => {
  const router = useRouter();

  const callSignIn = ( providerName: keyof typeof providers ) => () => {
```

```
  firebase.auth()
      .signInWithPopup( providers[ providerName ] )
      .then( ( result ) => {
          const token = result?.credential?.accessToken;
          const user = result.user;
          if ( token && user ) router.push( '/wallets' );
      } )
      .catch( function( error ) {
          const errorCode = error.code;
          const errorMessage = error.message;
          const email = error.email;
          const credential = error.credential;
          console.error( 'error', {
              errorCode,
              errorMessage,
              email,
              credential
          } );
      } );
  };

  return (
    <div className={ styles.providersAuth }>
      <span className="title">Log in with</span>
      <div className="providers w-100 d-flex justify-around">
        <Facebook className={ 'authProvider' }
onClick={ callSignIn( 'facebook' ) }/>
        <Google className={ 'authProvider' } onClick={ callSignIn( 'google' ) }/>
        <Twitter className={ 'authProvider' } onClick={ callSignIn( 'twitter' ) } />
```

```
      </div>
    </div>
  );
};
```

Файл «src/components/EnabledCategoriesList.tsx»:

```
import React, { useState } from 'react';
import { connect, ConnectedProps } from 'react-redux';
import { IStore } from 'store';
import { ICategory } from 'store/firebase';
import EnabledCategoryLine from './EnabledCategoryLine';
import styles from './index.module.scss';

const mapStateToProps = ( s: IStore ) => ( { categories:
s.firestore.ordered.categories, } );
const mapDispatchToProps = {};
const connector = connect( mapStateToProps, mapDispatchToProps );

type PropsFromRedux = ConnectedProps<typeof connector>;
type IProps = PropsFromRedux & {
  categories: ICategory[];
};

const EnabledCategoriesList = ( { categories }: IProps ) => {
  const [tab, setTab] = useState<'income' | 'expense'>( 'expense' );

  if ( !categories ) {
    return null;
```

```
}

const incomeCategories = categories.filter(
  ( category ) => category.type === 'income'
);
const expenseCategories = categories.filter(
  ( category ) => category.type === 'expense'
);
const resultCategories =
  tab === 'income'
    ? incomeCategories
    : tab === 'expense'
      ? expenseCategories
      : [];
return (
  <div className={ styles.enabledCategoriesListWrapper }>
    <div className="tabs">
      {['expense', 'income'].map( ( tabName ) => (
        <div
          key={ tabName }
          className={ `${tab === tabName ? 'selected' : ''} tab` }
          onClick={ () => setTab( tabName ) }
        >
          {tabName}
        </div>
      ) )}
    </div>
    <div className={ 'list' }>
      {resultCategories &&
```

```
        resultCategories
          .filter( ( cat ) => !cat.isExcluded )
          .map( ( category ) => (
            <EnabledCategoryLine key={ category.id } category={ category } />
          ) )}
      </div>
    </div>
  );
};

export default connector( EnabledCategoriesList );
```

Файл «pages/wallets/index.tsx»:

```
import React, { useState } from 'react';
import { getArrayFromCollection, IWallet, IWalletType } from 'store/firebase';
import nookies from 'nookies';
import ActiveTypeWallets from 'components/ActiveTypeWallets';
import TopBalance from 'components/TopBalance';
import TypeSwitcher from 'components/TypeSwitcher';
import { firebaseAdmin } from '../../firebaseAdmin';
import { firebaseClient } from '../../firebaseClient';
import { InferGetServerSidePropsType, GetServerSidePropsContext } from 'next';
import LayoutWrapper from 'layoutWrapper';
import styles from './index.module.scss';

const Wallets = ({ wallets }: InferGetServerSidePropsType<typeof
getServerSideProps> ) => {
  const [currentType, setCurrentType] = useState( 'all' as IWalletType );
```

```
  return (
    <LayoutWrapper>
      <div className={ styles.wallets }>
        <div className={ 'upperPart' } >
          <TopBalance wallets={ wallets }/>
          <TypeSwitcher currentType={ currentType }
updateType={ setCurrentType }/>
        </div>
        <div className="downPart">
          <ActiveTypeWallets wallets={ wallets } currentType={ currentType }/>
        </div>
      </div>
    </LayoutWrapper>
  );
};

export const getServerSideProps = async ( ctx: GetServerSidePropsContext ) => {
  try {
    const cookies = nookies.get( ctx );
    const token = await firebaseAdmin.auth().verifyIdToken( cookies.token );
    const { uid, email } = token;
    const wallets: IWallet[] = getArrayFromCollection(
      await
firebaseClient.firestore().collection( 'users' ).doc( uid ).collection( 'wallets' ).get()
    ) as IWallet[];

    return {
      props: {
```

```
        message: `Your email is ${email} and your UID is ${uid}.`,
        wallets: wallets
      },
    };
  } catch ( err ) {
    return {
      redirect: {
        permanent: false,
        destination: '/login',
      },
      props: {} as never,
    };
  }
};

export default Wallets;
```

Файл «src/helpers/index.ts»:

```
import { v4 } from 'uuid';

export const generateUuid = () => v4();

export const createArrayLine = ( arr: string[], numberOfItems: number ) => {
  if ( arr.length <= numberOfItems ) {
    return [arr];
  }
```

```
  const result = [];
  let tempArr = arr;
  for ( let i = tempArr.length; i > 0; i -= numberOfItems ) {
    const line = tempArr.slice( 0, numberOfItems );
    result.push( line );
    tempArr = tempArr.slice( numberOfItems );
  }

  return result;
};

export const getRandomColor = (colorsArray) => {
  return colorsArray[ Math.floor( Math.random() * colorsArray.length ) ];
};

export const getCurrencyByProperty = ( prop: string, value: string ) => {
  const result = Object.keys( currenciesObject ).map( ( shortName ) => {
    const obj = currenciesObject[ shortName as keyof typeof currenciesObject ];
    return obj;
  } ).filter( ( item ) => item !== undefined );
  return result;
};

export const getRandomCurrency = () => {
  const randomIndex = Math.floor( Math.random() * Object.keys( currenciesObject
).length );
  const resultIndex = Object.keys( currenciesObject )[ randomIndex ] as keyof
typeof currenciesObject;
```

```
  const result = currenciesObject[ resultIndex ];
  return result;
};

export const createParsedAmount = ( value: string ) =>  {
  return Math.abs( Number( `${value.slice( 0, -2 )}.${value.slice( -2 )}` ) );
};

export const numberWithCommas = ( x: number ) =>
x.toString().split( '.' ).join( ',' );
```

Файл «pages/_app.tsx»:

```
import type { AppProps } from 'next/app';
import React from 'react';
import { Provider } from 'react-redux';
import store from 'store';
import { AuthProvider } from 'hooks/useAuth';
import '../styles/globals.css';
import '../styles/firebaseui-styling.global.css';

function MyApp( { Component, pageProps }: AppProps ) {
  return (
    <Provider store={ store }>
      <AuthProvider>
        <Component { ...pageProps } />
      </AuthProvider>
    </Provider>
  );
```

```
}

export default MyApp;
```

Файл «store/types.ts»:

```
import {
  FirebaseReducer,
  FirestoreReducer,
  getFirebase,
} from "react-redux-firebase";
import { ThunkAction } from "redux-thunk";
import firebase from "firebase/compat/app";
import { generateUuid } from "helpers";
import { currenciesObject } from "constants";
import { IStore } from ".";


export declare type ICurrency = typeof currenciesObject[ keyof typeof
currenciesObject];

export interface IProfile {
  name: string;
  email: string;
  uid: string;
  settings: {
    defaultCurrncy: string;
  };
}
```

```
export interface ISchema {
  users: IProfile;
}

export type AppThunk<ReturnType = void> = ThunkAction<
  ReturnType,
  IStore,
  typeof firebase,
  AllActions
>;

export type InferValueTypes<T> = T extends { [key: string]: infer U }
  ? U
  : never;

const uuid = generateUuid();
export type UID = typeof uuid;

export declare type IIcon = {
  path: string;
  color: string;
};
```