ДЕРЖАВНИЙ УНІВЕРСИТЕТ ІНФОРМАЦІЙНО-КОМУНІКАЦІЙНИХ ТЕХНОЛОГІЙ
НАВЧАЛЬНО-НАУКОВИЙ ІНСТИТУТ ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ
КАФЕДРА ІНФОРМАЦІЙНИХ СИСТЕМ ТА ТЕХНОЛОГІЙ

КВАЛІФІКАЦІЙНА РОБОТА

на тему:

«СИСТЕМА ВЕБ-ЗАСТОСУНОК ДЛЯ ПОШУКУ ТА КАТАЛОГІЗАЦІЇ КІНОФІЛЬМІВ НА REACT»

на здобуття освітнього ступеня бакалавр
за спеціальності 126 Інформаційні системи та технології
(код,найменування спеціальності)
освітньо-професійної програми Інформаційні системи та технології
(назва)

Кваліфікаційна робота містить результати власних досліджень. Використання ідей, результатів і текстів інших авторів мають посилання на відповідне джерело

_____ Костянтин КОЛЕШНЯ
(підпис) *(ім'я, ПРІЗВИЩЕ здобувача)*

Виконав:
здобувач вищої освіти
група ІСД-42 Костянтин КОЛЕШНЯ
(ім'я, ПРІЗВИЩЕ)
Керівник Ольга ПОЛОНЕВИЧ
к.т.н. *(ім'я, ПРІЗВИЩЕ)*
Рецензент: _____
науковий ступінь, вчене звання *(ім'я, ПРІЗВИЩЕ)*

Київ 2025

ДЕРЖАВНИЙ УНІВЕРСИТЕТ
ІНФОРМАЦІЙНО-КОМУНІКАЦІЙНИХ ТЕХНОЛОГІЙ

Навчально-науковий інститут Інформаційних технологій

Кафедра Інформаційних систем та технологій
Ступінь вищої освіти бакалавр
Спеціальність 126 Інформаційні системи та технології
Освітньо-професійна програма Інформаційні системи та технології

ЗАТВЕРДЖУЮ

Завідувач кафедрою ІСТ
_____ Каміла СТОРЧАК

“____” _____2025 року

З А В Д А Н Н Я
НА КВАЛІФІКАЦІЙНУ РОБОТУ

Колешні Костянтину Дмитровичу
(прізвище, ім’я, по батькові здобувача)

1.Тема кваліфікаційної роботи: Система моніторингу та управління транспортною інфраструктурою на основі IoT
керівник кваліфікаційної роботи: Ольга ПОЛОНЕВИЧ к.т.н., доцент
(ім’я, ПРІЗВИЩЕ, науковий ступінь, вчене звання)
затверджені наказом Державного університету інформаційно-комунікаційних технологій від “24” лютого 2025 р. №56

2. Строк подання кваліфікаційної роботи «30» травня 2025 р.
3. Вихідні дані кваліфікаційної роботи:
1. Науково технічна література.
2. Аналіз технологічних підходів розробки веб-застосунків.
3. Веб-застосунок клієнт-серверної архітектури для пошуку фільмів.

4. Зміст розрахунково-пояснювальної записки (перелік питань, які потрібно розробити):
1. Огляд існуючих рішень для вибору технологій розробки веб-застосунку.
2. Порівняльний аналіз сучасних JavaScript-фреймворків
3. Обґрунтування вибору архітектури SPA та технологічного стеку
4. Розробка структури та архітектури веб-застосунку (клієнтська і серверна частина)

5.Перелік ілюстраційного матеріалу: *презентація*

6. Дата видачі завдання «24» лютого 2025 р.

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів кваліфікаційної роботи	Строк виконання етапів роботи	Примітка
1	Вступ	01.03.25	
2	Розділ 1. ОГЛЯД ІСНУЮЧИХ РІШЕНЬ ДЛЯ ВИБОРУ ТЕХНОЛОГІЙ РОЗРОБКИ ВЕБ-ЗАСТОСУНКУ	09.03.25	
3	Порівняльний аналіз фреймворків JavaScript	17.03.25	
4	Розділ 2. ПРОЕКТУВАННЯ ВЕБ-ЗАСТОСУНКУ	23.03.25	
5	Архітектурне проєктування веб-застосунку	30.04.25	
6	Вибір інструментів і технологій	05.04.25	
7	Розділ 3. РОЗРОБКА ВЕБ-ЗАСТОСУНКУ	07.04.25	
8	Висновки	29.04.25	
9	Оформлення та перевірка бакалаврської роботи	01.05.25	

Здобувачка вищої освіти _____ Костянтин КОЛЕШНЯ
(підпис) *(Ім'я, ПРІЗВИЩЕ)*

Керівник
кваліфікаційної роботи _____ Ольга ПОЛОНЕВИЧ
(підпис) *(Ім'я, ПРІЗВИЩЕ)*

РЕФЕРАТ

Текстова частина кваліфікаційної роботи на здобуття освітнього ступеня бакалавра: 71 стор., 30 рис., 2 табл., 20 джерел.

Мета роботи – створення веб-застосунку для зручного пошуку, перегляду та персональної каталогізації інформації про кінофільми, що використовує сучасні веб-технології.

Об'єкт дослідження - сучасна веб-розробка.

Предмет дослідження - клієнт-серверний веб-застосунок на React.

У роботі проаналізовано сучасні підходи до створення веб-застосунків, проведено порівняльний аналіз фреймворків (React, Angular, Vue.js), обґрунтовано вибір архітектурної моделі (SPA) та стеку технологій (React, Node.js, Express, REST API). Реалізовано функціонал пошуку, фільтрації та візуалізації фільмів із використанням зовнішнього джерела TMDb API, а також локальне збереження добірок за допомогою LocalStorage. Створено сервер-проксі для захисту API-ключа й кешування запитів.

Застосунок має компонентну структуру з реалізованою внутрішньою маршрутизацією та адаптивним інтерфейсом.

КЛЮЧОВІ СЛОВА: ВЕБ-ЗАСТОСУНОК, КАТАЛОГІЗАЦІЯ, REACT, NODE.JS, SPA, REST, LOCAL STORAGE, КЕШУВАННЯ, РОЗРОБКА.

ЗМІСТ

ВСТУП

У сучасному цифровому середовищі значну роль у сфері масових комунікацій відіграють стрімінгові сервіси, онлайн-бази даних та інформаційно-аналітичні платформи. Особливої актуальності набуває завдання впорядкування та забезпечення доступу до структурованої інформації про фільми, що охоплюють широку жанрову палітру, різноманіття творчих команд, країн-виробників та дат випуску. Кіноглядачі, критики й ентузіасти все частіше звертаються до платформ, які надають синтезовані дані про кінострічки — зокрема, описи, рецензії, рейтинги, відеоматеріали й візуальний супровід. Прикладами таких популярних рішень є IMDb, TMDB і Letterboxd, які поєднують довідкову функцію з можливостями соціальної взаємодії та рекомендаційного контенту.

Однак стрімке зростання обсягу функціональності подібних сервісів виявляє низку недоліків: складність освоєння інтерфейсу, перенасиченість можливостями, нав’язлива реклама, відсутність гнучкості налаштувань і персоналізації, обмеження можливостей без реєстрації або підписки. Це створює потребу в створенні легкого та інтуїтивно зрозумілого в користуванні веб-застосунку, який дозволить користувачам самостійно шукати, фіксувати та класифікувати інформацію про фільми без складної авторизації та без перевантаження зайвими функціям.

У такому контексті актуальною є ідея розробки власного клієнтського веб-застосунку, який може слугувати демонстраційним або навчальним прикладом, що ілюструє принципи побудови сучасної інформаційної системи. Реалізація подібного проекту дозволяє дослідити ключові аспекти веб-архітектури, способи обробки медіаданих, засоби пошуку та фільтрації, а також реалізувати механізми персональної каталогізації з використанням React.

Результатом дипломної роботи є односторінковий веб-застосунок для пошуку та збереження інформації про кінофільми, орієнтований на забезпечення зручного доступу до базових функцій: пошуку, фільтрації за жанрами, рейтингом і роком випуску.

Актуальність теми

Розробка даного веб-застосунку є актуальною через:

- швидке зростання обсягів цифрового медіаконтенту;

- тенденцію до створення SPA-рішень із акцентом на інтерфейс користувача;

- потребу у кастомізований рішеннях для глядачів, оглядачів і кіноманів;

- нестачу зручних сервісів без реєстрації, складних налаштувань і нав'язливої реклами.

Мета дослідження

Мета роботи полягає у створенні веб-застосунку для пошуку, перегляду та індивідуального впорядкування інформації про фільми із застосуванням бібліотеки React, що відповідає сучасним вимогам до зручності, адаптивності та доступності користувацького досвіду.

Завдання роботи

Для досягнення мети передбачено вирішення наступних завдань:

1. Провести аналіз предметної області та існуючі рішення;

2. Визначити вимоги кінцевого користувача до функціональності продукту;

3. Обґрунтувати вибір технічного інструментів, бібліотек і підходів до реалізації;

4. Побудувати логічну й візуальну структуру SPA-застосунку;

5. Реалізувати функціональність пошуку, фільтрації та збереження даних;

6. Провести тестування інтерфейсу та функціональності згідно з нефункціональним вимогам.

Об'єкт і предмет дослідження

Об'єкт дослідження — процеси розробки клієнтських веб-рішень для подання тематично структурованої інформації.

Предмет дослідження — методи побудови SPA-застосунків із використанням React, принципи UI/UX-дизайну, механізми фільтрації та локального зберігання даних на стороні клієнта.

Методи дослідження

Під час реалізації проєкту застосовано такі методи:

- аналіз літератури та аналогів;

- проєктування UI за компонентною моделлю;

- програмна реалізація інтерфейсу із застосуванням React;

- тестування на відповідність функціональним і користувацьким очікуванням.

Практичне значення

Результат проєкту — функціональний прототип веб-застосунку, який може бути використаний:

- як навчальний приклад реалізації SPA-застосунку із застосуванням сучасних технологій;;

- як основа для масштабованого інформаційного сервісу з розширеною функціональністю;

- як персональний інструмент для ведення цифрової фільмотеки;

- як шаблон для впровадження інформаційних систем у суміжних галузях.

Отримані навички і знання мають універсальний характер і придатні до використання в інших програмних розробках.

1. ОГЛЯД ІСНУЮЧИХ РІШЕНЬ ДЛЯ ВИБОРУ ТЕХНОЛОГІЙ РОЗРОБКИ ВЕБ-ЗАСТОСУНКУ

Веб-сторінки з’явилися одночасно з появою інтернету. За цей час вони пройшли довгий шлях трансформацій – від текстових документів до складних застосунків. Сьогодні такі застосунки глибоко інтегровані у повсякденне життя кожної людини та надають динамічний та адаптивний користувацький досвід. Цей розвиток був зумовлений інноваціями в технологіях та методах розробки.

Спочатку веб-сторінки були статичними і використовували HTML тільки для навігації. Після того як з’явився CGI (Common Gateway Interface), з’явилася можливість динамічно створювати наповнення сторінки, в залежності від дій користувача. Таким чином, сторінки вже адаптували інформацію у реальному часі через взаємодію з сервером.

Ще пізніше почали з’являтися технології типу AJAX (Asynchronous JavaScript and XML), які дозволили робити асинхронні запити до сервера, не перезавантажуючи сторінку повністю. що суттєво покращило користувацький досвід і стало передумовою для створення високопродуктивних веб-застосунків.

Подальший розвиток стандартів як HTML5, CSS3, та ES6 (версія JavaScript) дозволили надзвичайно розширити функціонал веб-застосунків. HTML5 впровадив семантичну розмітку, підключення до API, підтримку мультимедії та покращив взаємодію з користувачем, CSS3 ми завдячуємо можливість створення складних анімацій, візуальних ефектів та адаптивного дизайну, а ES6 JavaScript запропонував новий синтаксис JavaScript, куди увійшли класи та інші інструменти для реалізації більш складної логіки.

Сучасний стан розвитку веб-застосунків

У теперішній час розробка веб-додатків характеризується різноманіттю екосистем і фремворків. React, Angular і Vue.js займають провідні позиції у фронтенд середовищі через продуктивність, зручність використання та масштабованості. Також, для того щоб забезпечити цілісність усього стеку розробки і писати все на

JavaScript, серверна частина часто реалізується за допомогою Node.js. Таке рішення знижує складність розробки і покращує ефективність команди.

Суттєву трансформацію підходу до розробки спричинило використання архітектури мікросервісної архітектури та API. Розбиття програм на невеликі, автономні сервіси покращило управління проектами та саму розробку. Такий підхід дав можливість розробникам незалежно один від одного проектувати, тестувати, розгортати та масштабувати окремі компоненти, що дає змогу швидко впроваджувати потрібні зміни і скорочувати час виходу на ринок. Разом з цим, зі зростанням кількості взаємозв'язків між сервісами у мікросервісних архітектурах критично важливим аспектом стала безпека. Сучасні веб-застосунки включають в себе різні практики і заходи для уникнення вразливостей. Вже на ранніх етапах проектування передбачається використання HTTPS замість HTTP, реалізацію політики безпеки контенту (CSP) та правильну конфігурацію CORS. Для забезпечення конфіденційності, цілісності даних та стійкість застосунку проти атак, важливі регулярні оновлення та дотримання галузевих стандартів.

У межах вибору технологій для веб-розробки ключовим є питання ефективного поєднання інструментів, що забезпечують стабільну роботу інтерфейсу, швидкий обмін даними з сервером та зручну підтримку архітектури застосунку. Розробка веб-застосунків вже давно вийшла за межі простого використання HTML та CSS — нині вона охоплює цілу екосистему технологій як для клієнтської, так і серверної частини.

Зі зростанням складності вимог до інтерактивності, продуктивності та масштабованості сучасних застосунків, з'являється потреба у гнучких підходах до розробки. Зокрема, середовище веб-розробки сьогодні активно використовує такі концепції як модульність, повторне використання компонентів, розділення логіки між сервісами, використання RESTful API та інтеграцію з зовнішніми платформами. Зростає роль інструментів візуального проектування (Adobe XD), інтерфейсного прототипування, а також автоматизованого тестування — таких як Jest, які дозволяють пришвидшити цикл розробки та уникнути помилок ще на етапі проєктування.

Популяризація NoSQL-рішень на зразок MongoDB, а також використання асинхронного виконання коду в Node.js, відкрили нові горизонти для розробки високопродуктивних систем. На додаток, важливими стали питання безпеки — зокрема реалізація шифрування паролів, токенізація (JWT), налаштування CORS та впровадження політик CSP для захисту від типових атак, таких як XSS та CSRF.

Таким чином, на сьогодні для побудови веб-застосунків недостатньо обрати лише мову програмування — потрібне комплексне розуміння всієї інфраструктури, її взаємодії та можливостей масштабування. Саме тому нижче подано аналіз сучасних інструментів і технологій, які найчастіше застосовуються при створенні веб-продуктів різного рівня складності.

1.1. Аналіз сучасних веб-технологій та інструментів

Сучасна веб-розробка сформувалася як самостійна галузь інженерії програмного забезпечення, що об'єднує велику кількість взаємопов’язаних технічних напрямів. До них належать, зокрема, проєктування користувацьких інтерфейсів, розробка серверної логіки, забезпечення інформаційної безпеки, автоматизоване тестування, керування даними, а також інтеграція з хмарними інфраструктурами. Зміна архітектурних парадигм — зокрема, перехід від монолітних застосунків до компонентно-орієнтованих підходів — зумовила зростання популярності односторінкових застосунків (SPA), які забезпечують високу інтерактивність, зменшення затримок у взаємодії з користувачем і гнучку обробку даних на клієнтській стороні.

Для реалізації SPA найбільш поширеним є використання JavaScript-фреймворків, серед яких провідні позиції займають React, Vue та Angular. Особливої популярності набув React, що розробляється та підтримується компанією Meta. Його широке застосування пояснюється низкою факторів: наявністю розвиненої екосистеми (зокрема, React Router, Redux, Context API, Next.js), високим рівнем абстракції компонентів, ефективною реалізацією механізмів повторного рендерингу через віртуальний DOM, а також порогом входження, прийнятним як для новачків,

так і для досвідчених розробників. У поєднанні з CSS-фреймворками та бібліотеками стилів — такими як Material UI, Tailwind CSS, Styled Components — React дозволяє створювати адаптивні, масштабовані та доступні інтерфейси користувача з високою продуктивністю та низьким часом відгуку. Застосування React у великих програмних продуктах, таких як Netflix, Airbnb, Facebook, Instagram, Pinterest, підтверджує його ефективність і придатність для промислового використання.

Разом із клієнтською частиною будь-який веб-застосунок потребує повноцінної серверної логіки, яка включає обробку HTTP-запитів, зв'язок з API, управління сесіями, кешування та реалізацію заходів інформаційної безпеки. Одним із найбільш продуктивних рішень у цій сфері є Node.js — середовище виконання JavaScript, яке дозволяє реалізовувати серверну логіку з використанням тієї самої мови програмування, що й клієнтську частину. Це сприяє уніфікації технологічного стеку, полегшує командну взаємодію і прискорює розробку. У зв’язці з Express.js, Node.js дозволяє швидко реалізовувати REST API, маршрути, обробку запитів, автентифікацію з використанням JSON Web Tokens (JWT), керування середовищем виконання за допомогою .env-файлів. Для зменшення навантаження на зовнішні API, наприклад TMDB, застосовується кешування за допомогою бібліотек типу node-cache, що дозволяє зменшити кількість повторних запитів і підвищити продуктивність системи. Важливою особливістю Node.js є підтримка асинхронної моделі виконання коду, яка має вирішальне значення при роботі з базами даних, сторонніми API або зовнішніми сервісами.

Поза межами базових технологій, веб-розробка ґрунтується на використанні широкого спектру допоміжних інструментів і сервісів. Для керування залежностями та пакунками застосовуються npm і Yarn, які дозволяють ефективно встановлювати, оновлювати та контролювати версії бібліотек. Хмарні платформи, такі як Vercel, Netlify та Firebase, забезпечують автоматизоване розгортання проєктів із вбудованими механізмами безперервної інтеграції та доставки (CI/CD), інтеграцією з репозиторіями Git та масштабуванням в режимі реального часу. Актуальними архітектурними підходами є JAMstack (JavaScript, API, Markup), serverless-архітектури (наприклад, AWS Lambda або Vercel functions), headless CMS-системи

(зокрема, Strapi та Sanity), а також використання WebAssembly (WASM) для інтеграції високопродуктивного коду на C/C++ або Rust безпосередньо у веб-застосунках.

Значну роль у забезпеченні надійності та стабільності веб-застосунків відіграють інструменти тестування. Jest є поширеною бібліотекою для юніт-тестування в екосистемі React, яка дозволяє перевіряти функціональність окремих компонентів, логіку обробки даних, а також виняткові ситуації. Для повного моделювання взаємодії користувача з інтерфейсом використовуються інструменти енд-ту-енд тестування, такі як Cypress або Playwright. Їхнє впровадження є критично важливим для підтримки якості програмного продукту, особливо в умовах інтенсивної командної розробки та безперервної інтеграції. У великих проєктах наявність розгалуженої системи тестів сприяє своєчасному виявленню регресій, підвищує стабільність релізів та полегшує рефакторинг.

Питання інформаційної безпеки є принципово важливим у контексті сучасної веб-розробки. Односторінкові застосунки мають забезпечувати захист як на клієнтському, так і на серверному рівнях від типових загроз, таких як XSS (міжсайтове скриптування), CSRF (міжсайтове підроблення запитів), SQL-ін’єкції, перехоплення сесій. Для цього застосовуються відповідні технічні засоби — політики Content Security Policy (CSP), обмеження CORS, кодування та валідація вхідних даних, автентифікація на основі токенів, використання захищених HTTPS-з’єднань. Захист API-ключів є окремим напрямом, що включає їх зберігання у .env-файлах, опрацювання виключно на серверній стороні та недоступність у клієнтському середовищі. У межах даного проєкту це реалізовано за допомогою проксі-сервера на базі Express.js.

Загальні тенденції розвитку галузі свідчать про зростання складності інструментального ландшафту та підвищення вимог до кваліфікації розробників. Все більшої популярності набувають архітектури з розділенням інтерфейсного і серверного рівнів (decoupled), використання гібридних фреймворків (наприклад, Remix для React), багатомовних середовищ (зокрема, Astro для генерації сайтів із використанням різних технологій), а також застосування генеративного штучного інтелекту для автоматизованого заповнення коду (GitHub Copilot, TabNine). Зростає

необхідність у підтримці інтеграцій з ML-сервісами, а також реалізації обміну даними в реальному часі — за допомогою WebSockets або GraphQL Subscriptions. Водночас REST API зберігає позиції як універсальний, передбачуваний і добре підтримуваний інструмент взаємодії.

Розробка сучасних веб-застосунків потребує комплексного підходу, який охоплює інтеграцію великої кількості взаємозалежних технологій. Навіть реалізація мінімально життєздатного продукту (MVP) передбачає знання основ клієнтської та серверної архітектури, механізмів кешування, маршрутизації, обробки асинхронних запитів, забезпечення інформаційної безпеки, адаптивності та продуктивності. У межах розробки застосунку для пошуку та каталогізації фільмів застосування стеку React + Node.js + REST API + TMDB дозволяє досягти оптимального співвідношення між масштабованістю, швидкістю, функціональністю та зручністю використання, що визначає доцільність обраного технічного рішення.

Додатково варто відзначити інструменти забезпечення якості та підтримки чистоти коду. ESLint є стандартом для статичного аналізу JavaScript-коду, який дозволяє виявляти синтаксичні, логічні та стилістичні помилки ще до запуску програми. Prettier виконує автоматичне форматування коду відповідно до заданих правил. У поєднанні з Husky та lint-staged вони забезпечують автоматизацію перевірок на етапі комітів до Git-репозиторію, обмежуючи ці перевірки лише зміненими файлами. Це дозволяє підтримувати узгодженість кодової бази та зменшити кількість помилок на етапі інтеграції змін.

На завершення, слід підкреслити важливість використання TypeScript як надбудови над JavaScript. Його статична типізація дозволяє явно визначати типи змінних, функцій і об’єктів, що знижує кількість помилок у час виконання, полегшує навігацію в проєкті та підвищує стабільність розробки. Крім того, TypeScript забезпечує зворотну сумісність із JavaScript і дозволяє поступово інтегрувати типи без необхідності повної міграції. У результаті, TypeScript суттєво підвищує надійність, передбачуваність і масштабованість проєктів у сучасних умовах.

1.2. Архітектура клієнт-серверних застосунків

Клієнт-серверна архітектура — це класична модель організації обчислювальних систем, у якій чітко розділено функції між двома сторонами: клієнт відповідає за ініціювання запитів, а сервер — за їх обробку, збереження та повернення відповідей. У сучасному веб-просторі ця модель є універсальною основою для більшості цифрових рішень — від простих веб-сторінок до складних корпоративних систем, мобільних застосунків і хмарних сервісів.

Ключовою особливістю клієнт-серверної взаємодії є асиметрія ролей: клієнт формулює запит, а сервер обробляє його, виконуючи обчислення або операції з базою даних, після чого надсилає відповідь. Цей процес забезпечується стандартними протоколами, зокрема HTTP, FTP, SMTP, які реалізують модель “запит–відповідь”. Важливою складовою є також інтерпроцесна комунікація — механізм обміну даними між віддаленими процесами, що дозволяє підтримувати логічну цілісність і ефективність системи навіть за умов розподіленого середовища.

Найпростішим прикладом є дворівнева архітектура, де клієнт напряму звертається до сервера бази даних. Такий підхід реалізується без проміжного програмного шару й підходить для невеликих рішень з обмеженим числом користувачів. Він має переваги у простоті впровадження та мінімальних витратах, але водночас демонструє суттєві обмеження: недостатню масштабованість, ускладнене оновлення клієнтів і ризики безпеки, пов’язані з прямим доступом до бази даних зі сторони користувача.

Більш надійною вважається трирівнева архітектура. Вона складається з трьох логічно відокремлених рівнів: користувацького інтерфейсу, бізнес-логіки та системи керування базами даних. У цій моделі клієнт надсилає запит серверному застосунку, який виконує основну обробку, а вже потім звертається до бази даних. Така структура дозволяє централізовано змінювати логіку без необхідності оновлення клієнтів, забезпечує більший контроль над доступом до даних і дає змогу масштабувати компоненти окремо, відповідно до поточного навантаження.

Подальший розвиток цієї моделі привів до появи багаторівневих архітектур, у яких кількість логічних шарів може суттєво перевищувати три. Наприклад, можуть

бути виділені окремі компоненти, що відповідають за автентифікацію, кешування, маршрутизацію, аналітику або взаємодію з API-шлюзами. Такий підхід широко використовується у складних розподілених системах, зокрема у сфері електронної комерції, фінансових сервісах, банківській інфраструктурі, а також у хмарних платформах. Його перевагами є висока гнучкість, масштабованість і відмовостійкість, але водночас це вимагає складної системи адміністрування, постійного моніторингу, логування та тестування.

Центральним елементом трирівневих і багаторівневих архітектур є middleware — програмне забезпечення, що забезпечує взаємодію між клієнтською частиною та сервером. Саме middleware реалізує більшість ключових функцій: бізнес-логіку, обробку помилок, автентифікацію, кешування, маршрутизацію, балансування навантаження та інтеграцію з іншими сервісами. До найпоширеніших прикладів належать Node.js, Express.js, Apache Tomcat — платформи, що слугують своєрідним «містком» між користувачем і сервером даних.

Значну трансформацію клієнт-серверна модель зазнала з розвитком хмарних технологій. У цьому контексті ключовими є моделі SaaS (програмне забезпечення як послуга), PaaS (платформа як послуга) та IaaS (інфраструктура як послуга). Їх застосування дає змогу масштабувати ресурси відповідно до потреб, зменшити витрати на фізичне обладнання, автоматизувати процеси розгортання та централізовано управляти обслуговуванням систем. У випадку SaaS користувачі працюють із застосунком через веб-інтерфейс, уся логіка якого розміщена в хмарі. Модель PaaS надає розробникам готові середовища для створення та запуску програм, а IaaS забезпечує доступ до орендованих обчислювальних ресурсів, сховищ і мережевої інфраструктури.

Одним із перспективних напрямів еволюції клієнт-серверної взаємодії є впровадження мобільних агентів — автономних програмних компонентів, які можуть переміщуватися між вузлами мережі й виконувати певні обчислення без постійного підключення до сервера. Ці агенти поєднують риси клієнт-серверної та однорангової (peer-to-peer) моделей, зменшують трафік у мережі та підвищують стійкість систем

до збоїв, зберігаючи здатність до виконання навіть при тимчасовій втраті зв'язку або недоступності окремих вузлів.

Попри широкі можливості, клієнт-серверна архітектура має низку важливих викликів. Одним із головних є забезпечення інформаційної безпеки. Через використання загальнодоступних мереж існує ризик перехоплення даних, атак на автентифікацію, ін'єкцій SQL, XSS, CSRF та інших типових загроз. Також збільшується складність адміністрування: зі зростанням кількості рівнів у системі зростає і потреба в розгорнутих інструментах моніторингу, логування подій, аудиту доступу та контролю консистентності між сервісами.

У підсумку, клієнт-серверна архітектура, попри свою відносну простоту, залишається актуальним і універсальним рішенням для проєктування та розгортання сучасних програмних систем. Її гнучкість і здатність до масштабування роблять її придатною як для невеликих проєктів, так і для складних хмарних сервісів. Послідовна еволюція цієї моделі — від двох до багатьох логічних рівнів — свідчить про її адаптивність до змін технологічного ландшафту. Обґрунтований вибір архітектури, з урахуванням вимог до безпеки, продуктивності та витрат, дозволяє ефективно вирішувати задачі будь-якої складності в межах веб-розробки.

1.3. Порівняльний аналіз фреймворків JavaScript (React, Angular, Vue.js)

У сфері сучасної фронтенд-розробки веб-застосунків ключове місце посідають JavaScript-фреймворки, які визначають не лише структуру коду, а й продуктивність, масштабованість та зручність подальшої підтримки програмного продукту. Саме вибір фреймворку часто зумовлює архітектурну побудову системи, організацію командної роботи, швидкість розгортання та обсяг технічного боргу. Серед найпоширеніших рішень, що стали індустріальними стандартами, виокремлюють Angular, React і Vue.js. Кожен із них реалізує власну концепцію побудови інтерфейсів, має відмінні функціональні особливості, різну складність освоєння та специфічні сценарії використання.

Angular є повноцінним фреймворком, створеним з урахуванням архітектурних шаблонів MVC (Model–View–Controller) та MVVM (Model–View–ViewModel). Його призначення — реалізація великих, структурно організованих

SPA-рішень (Single Page Applications), де особливо важливими є масштабованість, модульність і суворе дотримання єдиних стандартів. Angular повністю базується на TypeScript, що забезпечує статичну типізацію, покращене автодоповнення в редакторах коду та підвищену передбачуваність поведінки під час розробки. Крім того, Angular надає вбудовану підтримку для ряду важливих задач — маршрутизації, валідації форм, обробки подій, анімації, реактивного програмування (через RxJS) та модульного тестування. Підхід «конвенція понад конфігурацію» (convention over configuration), закладений у фреймворк, зменшує кількість ручних налаштувань, однак потребує глибокого розуміння внутрішньої архітектури платформи вже на ранніх етапах роботи з нею.

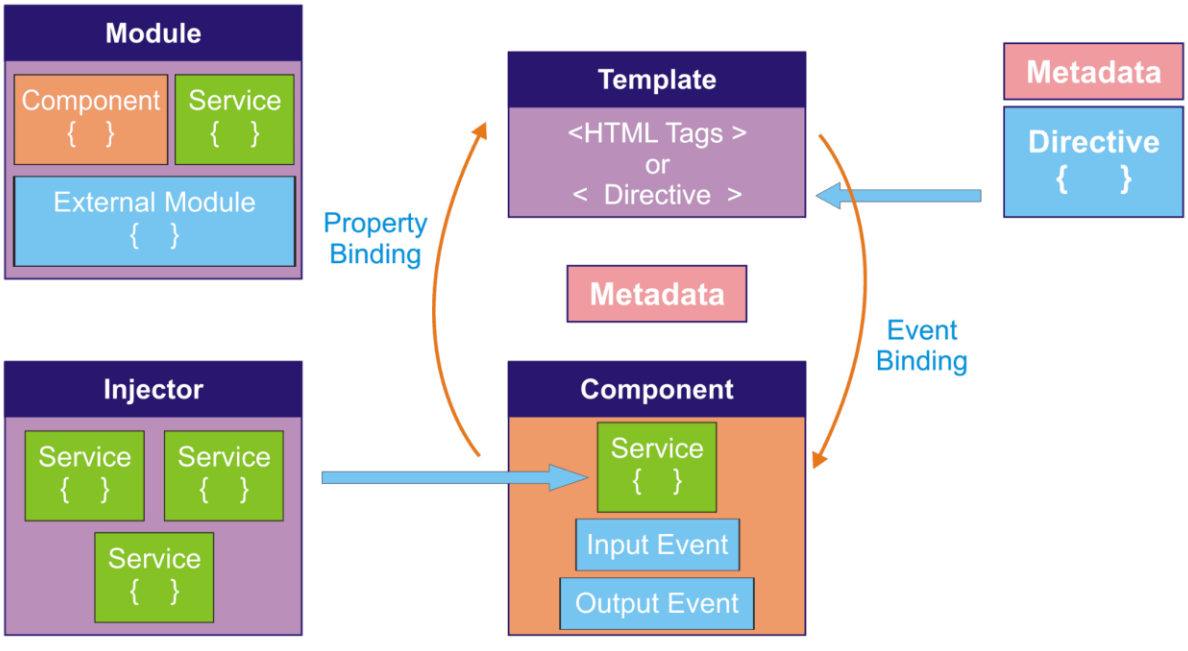

Рис. 1.1

React, на відміну від Angular, не є повноцінним фреймворком, а виступає як бібліотека, зосереджена виключно на побудові інтерфейсу користувача. Його основою є компонентно-орієнтований підхід, який дозволяє розробникам створювати UI як набір незалежних, повторно використовуваних компонентів. Ключовою технологічною перевагою React є використання віртуального DOM — абстракції, яка забезпечує ефективне оновлення інтерфейсу, мінімізуючи кількість перерендерів і підвищуючи продуктивність. Гнучкість React виявляється у тому, що він не нав'язує жорстких архітектурних рішень, залишаючи за розробником вибір супутніх

інструментів — для маршрутизації (React Router), керування станом (Redux, Zustand), стилізації (Styled Components, Emotion) тощо. React написаний на JavaScript, але має повну сумісність з TypeScript, що дозволяє будувати проєкти з високим рівнем безпеки типів та масштабованості. Широка екосистема додаткових бібліотек і активна спільнота підтримують його провідну позицію в галузі.

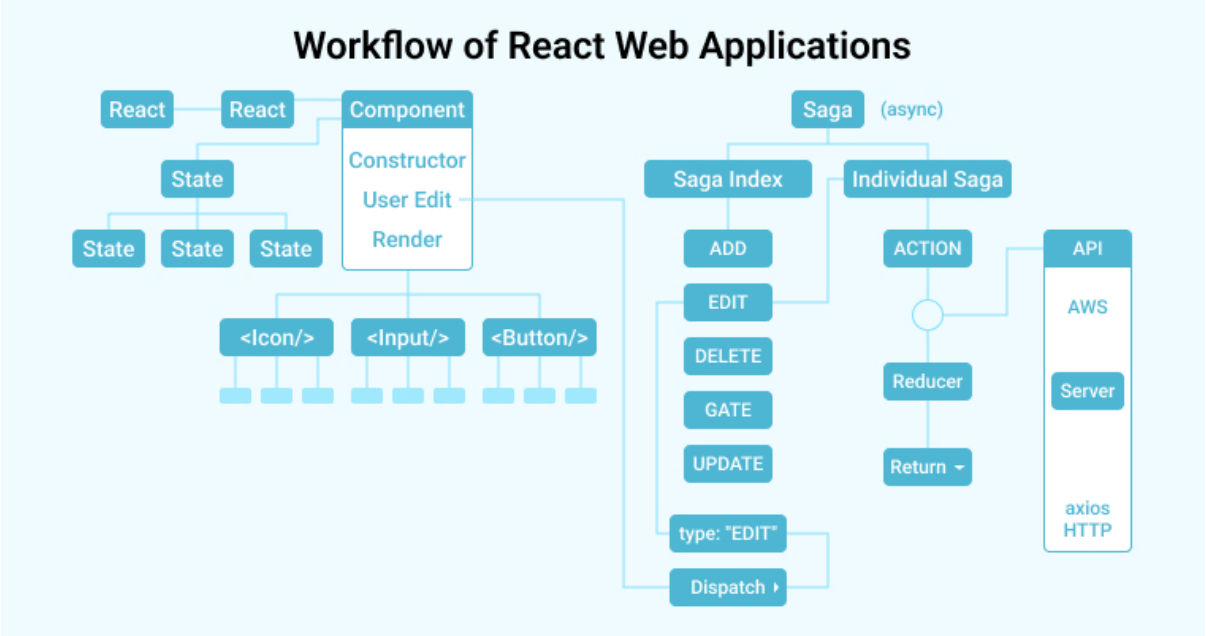

Рис. 1.2,

Фреймворк Vue.js поєднує деякі концептуальні підходи, притаманні як Angular, так і React, зберігаючи при цьому простоту, гнучкість і низький поріг входження. Його синтаксис наближений до класичного HTML-шаблонного стилю, що робить його зручним для швидкого освоєння, особливо розробниками з початковим досвідом. Vue підтримує двостороннє зв’язування даних, декларативне управління DOM і дозволяє використовувати як імперативний, так і декларативний стиль програмування. У базову екосистему входять Vue Router для маршрутизації, Vuex для централізованого керування станом, а також інструменти для модульного тестування, SSR та інтеграції з TypeScript. Однак попри свою зручність для невеликих і середніх проєктів, Vue менш масштабований у порівнянні з React і поступається йому в гнучкості, екосистемній підтримці, зрілості рішень для оптимізації продуктивності та інтеграції зі складними архітектурами.

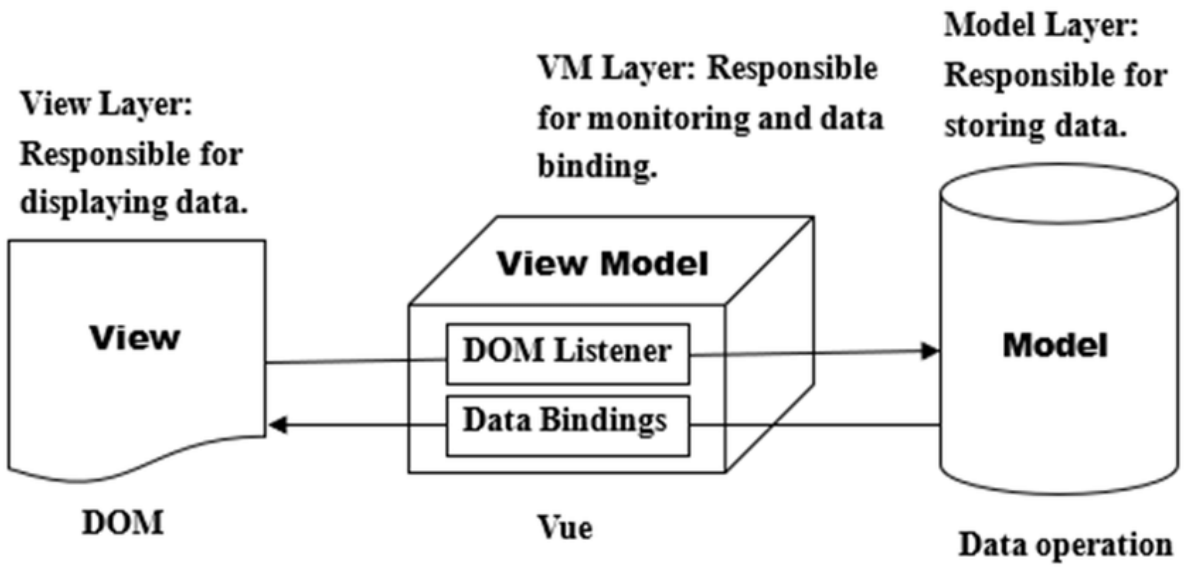

Рис. 1.3

З точки зору продуктивності виконання, React і Vue переважно демонструють вищу швидкість відображення інтерфейсу та обробки змін у DOM порівняно з Angular. Це зумовлено як компактнішою архітектурою, так і оптимізованим механізмом роботи з віртуальним DOM. Angular, натомість, виявляє свою силу у стабільності на складних проєктах із великою кількістю взаємозалежних компонентів. Завдяки статичному аналізу шаблонів, механізму зміни відстеження (Change Detection) і компіляції AOT (Ahead-of-Time), Angular може забезпечувати передбачувану продуктивність навіть у навантажених системах. У проєктах із динамічним UI та великою кількістю реактивних станів React демонструє високу ефективність, особливо в поєднанні з техніками мемоізації, Suspense, Server-Side Rendering (SSR) та інструментами оптимізації рендерингу.

Продуктивність розробки також варіюється залежно від обраного інструменту. Vue та React загалом вимагають менше зусиль на початкове освоєння, що робить їх зручними для стартапів, навчальних проєктів і невеликих команд. Швидке розгортання прототипів, гнучкість у виборі архітектурних рішень та простота документації — все це робить їх привабливими для розробників із різним рівнем досвіду. Angular, зі свого боку, орієнтований на складні, довготривалі корпоративні системи, де важливими є масштабування, формалізація структури коду, централізоване тестування та сувора типізація. Попри вищий поріг входження,

Angular надає все необхідне «з коробки» — що є критичним для команд із великою кількістю розробників.

Вибір фреймворку також залежить від типу та масштабу проєкту. Angular часто використовується для реалізації ERP-систем, корпоративних порталів, CRM, інтернет-банкінгу — там, де важлива стабільна структура, стандартизована взаємодія між модулями та тривалий життєвий цикл застосунку. React підходить для динамічних, інтерактивних платформ: стрімінгових сервісів, соціальних мереж, аналітичних панелей або кастомізованих маркетингових сайтів. Vue, своєю чергою, зручний як для невеликих односторінкових застосунків, так і для поступової інтеграції у вже існуючі проєкти — наприклад, у вигляді окремих віджетів або функціональних блоків.

Порівняльні дослідження показують, що React зазвичай демонструє найменший обсяг початкового коду і найкоротший час до MVP. Vue відзначається простотою налаштувань та доступною документацією, що робить його зручним для новачків і команд, які прагнуть швидко досягти функціонального результату. Angular, попри більший вхідний поріг, надає найповніший набір вбудованих можливостей, що особливо цінно в умовах корпоративного середовища зі складною бізнес-логікою, високими вимогами до тестування, підтримки та масштабування.

Узагальнюючи, можна зробити висновок, що жоден із розглянутих фреймворків не є абсолютним лідером у всіх категоріях. Кожен з них має свою нішу, свої сильні сторони та призначення. Раціональний вибір між Angular, React та Vue передбачає врахування багатьох чинників: технічних вимог проєкту, наявного досвіду команди, особливостей процесу розробки та стратегічного бачення розвитку системи. Обґрунтоване архітектурне рішення, ухвалене на початкових етапах, значною мірою визначає не лише ефективність розробки, але й витрати на подальшу підтримку, масштабування, модернізацію та інтеграцію із зовнішніми системами.

1.4. Обґрунтування вибору React

У процесі розробки сучасного веб-застосунку одним із ключових рішень є вибір технології, яка забезпечить збалансоване поєднання гнучкості, високої

продуктивності, швидкого старту та стабільної підтримки в довгостроковій перспективі. Це особливо важливо для динамічних інтерфейсів, орієнтованих на роботу з великими обсягами даних, де критичне значення має швидкість реакції, адаптивність до пристрою користувача та стабільність взаємодії. З урахуванням вимог до функціональності, масштабованості й підтримуваності, використання React як основного інструменту для реалізації клієнтської частини веб-застосунку є технічно вмотивованим і стратегічно доцільним.

React — це відкрита JavaScript-бібліотека, розроблена та підтримувана компанією Meta (раніше Facebook), яка надає розробникам засоби для створення високодинамічних, реактивних інтерфейсів. Основою його підходу є компонентна модель, що дозволяє будувати застосунок як набір незалежних, багаторазово використовуваних блоків. Такі компоненти можуть бути ізольовано протестовані, масштабовані та підтримані без ризику вплинути на решту застосунку. Цей принцип модульності покращує організацію структури проєкту й істотно полегшує як початкову розробку, так і довготривалу підтримку, що є особливо важливим у застосунках, де активно обробляються динамічні дані — наприклад, списки фільмів, рейтинги, жанри, рецензії.

Одним із визначальних технічних рішень у React є використання віртуального DOM — внутрішньої абстракції, що оновлюється в оперативній пам’яті та порівнюється з поточним станом реального DOM перед внесенням змін. Це дозволяє оптимізувати оновлення лише тих елементів, які дійсно змінилися, мінімізуючи кількість рендерингів і підвищуючи загальну продуктивність. Завдяки цьому React забезпечує чутливу та швидку взаємодію навіть у складних інтерфейсах із великою кількістю одночасних змін.

Окремої уваги заслуговує широка та добре підтримувана екосистема інструментів, що інтегруються з React без потреби в складних налаштуваннях. Серед них — React Router (маршрутизація), Redux та Context API (керування станом), Next.js (серверний рендеринг, генерація статичних сторінок, оптимізація SEO), Styled Components (стилізація з інкапсуляцією), а також численні бібліотеки для роботи з формами, графікою, інтеграцією з API тощо. Завдяки такій екосистемі React можна

ефективно застосовувати як для створення простих SPA, так і для побудови повнофункціональних веб-платформ із глибокою інтеграцією, великою кількістю станів і вимогами до оптимізації.

Ще однією суттєвою перевагою є низький поріг входження в екосистему React. Його синтаксис зрозумілий для розробників, знайомих з JavaScript, а принципи побудови компонентів логічні та інтуїтивно передбачувані. Це дає змогу швидко залучати нових членів до команди без значних витрат часу на навчання. Крім того, завдяки активній спільноті, великій кількості навчальних ресурсів, плагінів і відкритих рішень, React значно прискорює вирішення типових задач, пов’язаних із побудовою інтерфейсів.

Практичне підтвердження ефективності React можна знайти у кейсі Twitter Lite — прогресивного веб-застосунку, створеного як оптимізована альтернатива мобільному застосунку Twitter. Його основною метою було забезпечення повноцінного користувацького досвіду в умовах обмеженої пропускної здатності або нестабільного інтернет-з'єднання. Завдяки застосуванню React у зв’язці з Node.js розробникам вдалося досягти помітного зниження обсягу переданих даних, підвищення продуктивності та покращення швидкості завантаження. Як зазначено в офіційному блозі Twitter Engineering, застосунок став інтерактивним, забезпечивши зменшення середнього часу завантаження та скорочення затримок .

Ще один аспект, який сприяє популярності React, — його архітектурна нейтральність. На відміну від деяких фреймворків, React не накладає жорстких вимог щодо структури проєкту, що дозволяє розробникам адаптувати його до специфіки кожного окремого завдання. React однаково добре працює як у client-side, так і в server-side rendering (через Next.js), що критично важливо для продуктів, орієнтованих на SEO. Він також без проблем інтегрується з RESTful API, GraphQL, WebSocket, Firebase тощо — що відкриває широкі можливості для реалізації застосунків із різною логікою обміну даними.

Узагальнюючи, можна впевнено стверджувати, що React є технологією, яка повністю відповідає вимогам сучасної веб-розробки. Його гнучкість, висока продуктивність, широка підтримка спільноти, багата екосистема та наочні приклади

масштабного використання роблять його придатним як для реалізації MVP, так і для створення повноцінних промислових продуктів. У перспективі React дозволяє досягти оптимального балансу між швидкістю розробки, адаптивністю рішення, стабільною підтримкою та готовністю до масштабування — що робить його раціональним вибором для побудови сучасного, функціонального та надійного веб-застосунку.

1.5. Поняття SPA (Single Page Application)

Сучасна веб-розробка дедалі більше орієнтується на створення інтерактивних, динамічних і зручних у користуванні інтерфейсів, які наближаються за якістю до нативних мобільних або настільних застосунків. Однією з ключових архітектурних моделей, що дозволяє реалізувати такі характеристики, є Single Page Application (SPA), тобто «односторінковий застосунок». Це підхід, згідно з яким вся взаємодія користувача з веб-застосунком відбувається в межах однієї HTML-сторінки, а оновлення контенту здійснюється динамічно, без повного перезавантаження сторінки.

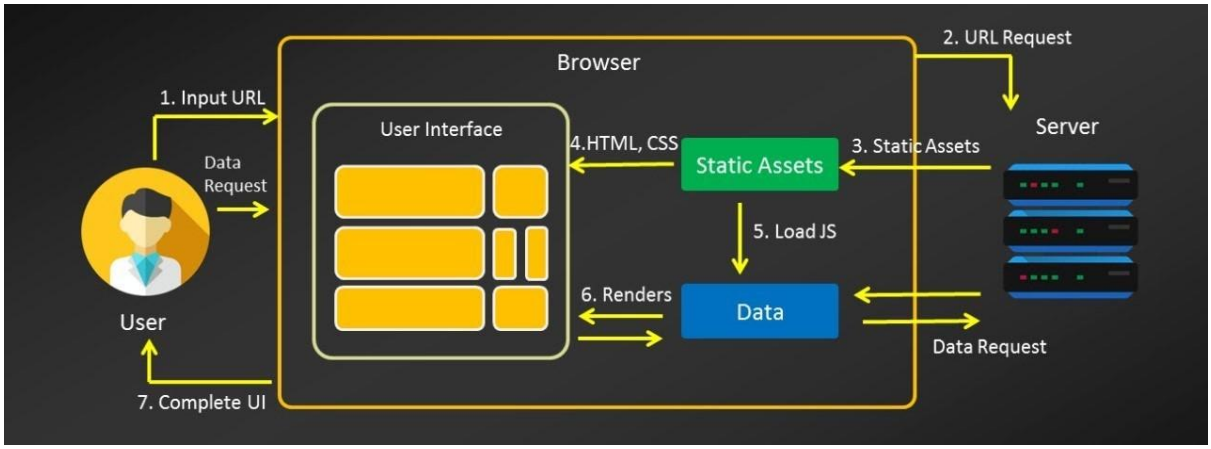

Рис. 1.4 Принцип роботи SPA,

Основна ідея SPA полягає в тому, що після початкового завантаження сторінки вся подальша передача даних здійснюється через асинхронні запити (зазвичай у форматі JSON), тоді як оновлення інтерфейсу виконується локально за допомогою JavaScript. Це дозволяє зменшити кількість запитів до сервера, уникнути повного рендерингу сторінки, знизити затримки у взаємодії та забезпечити більш

плавний користувацький досвід. Завдяки SPA інтерфейс реагує майже миттєво, що наближає взаємодію до рівня нативних застосунків.

Історичний розвиток SPA став можливим завдяки появі та широкому впровадженню JavaScript, а також технології AJAX, яка відкрила можливість здійснювати асинхронні запити до сервера без перезавантаження сторінки. Саме це стало точкою технологічного прориву: з’явилися застосунки нового типу — такі як Gmail, Facebook та LinkedIn — які могли оновлювати вміст «на льоту». У подальшому SPA-архітектура отримала поширення у низці популярних сервісів, включно з Netflix, Instagram, Trello, YouTube Studio, Spotify тощо, і наразі є стандартом у розробці складних клієнтських веб-застосунків.

Основні переваги SPA стосуються підвищеної швидкості роботи інтерфейсу, скороченого часу відгуку, покращеного UX, можливості реалізації офлайн-функцій та економного використання мережевих ресурсів. Вся логіка оновлення контенту обмежується єдиним контейнером (view), що дозволяє досягти високого рівня інтерактивності. Користувач сприймає таку систему не як веб-сторінку, а як повноцінну програму, що швидко реагує на введення, перемикається між розділами без затримок і зберігає стан між діями.

Архітектура SPA також має важливу організаційну перевагу — вона розділяє відповідальність між фронтенд- і бекенд-розробниками. Завдяки чіткому розмежуванню логіки інтерфейсу та бізнес-логіки сервера, обидві частини можуть розвиватися паралельно, використовуючи RESTful або GraphQL API як спільний канал зв’язку. Це прискорює розробку, спрощує тестування та підвищує масштабованість команди.

Найбільшого поширення SPA набули завдяки таким фреймворкам і бібліотекам, як React, Angular та Vue.js, які надають комплексні інструменти для побудови складних динамічних застосунків. Зокрема, React забезпечує ефективне управління станами компонентів, використання віртуального DOM для оптимізації рендерингу та підтримку гнучкої маршрутизації через React Router. У поєднанні з такими рішеннями, як Next.js, можлива реалізація SSR (server-side rendering) або SSG (static site generation), що значно покращує SEO та час першого завантаження.

Потрібно зазначити, що SPA не позбавлені недоліків. Одним із найбільш обговорюваних викликів є ускладнена оптимізація для пошукових систем. Оскільки більшість вмісту генерується на клієнтській стороні, стандартні пошукові боти можуть не мати змоги ефективно індексувати сторінки. Це може бути критичним для інформаційних ресурсів, де пошукова видимість є ключовою. Проте сучасні підходи, як-от SSR, SSG і hybrid-rendering (які реалізуються у Next.js), дозволяють вирішити ці обмеження, генеруючи HTML-контент на сервері або під час білду проєкту.

Додатковим аргументом на користь SPA є зменшення кількості HTTP-запитів і можливість збереження даних на клієнтській стороні, що є особливо корисним у мобільних умовах, коли з'єднання з мережею є нестабільним або повільним. У таких випадках SPA легко трансформується в PWA (Progressive Web App), яка поєднує функціональність веб-застосунку з можливістю офлайн-доступу, кешуванням ресурсів та інтеграцією з мобільною платформою користувача.

З технічної точки зору SPA використовують внутрішню маршрутизацію, при якій змінюється лише частина DOM, що відповідає за відображення конкретного розділу або компонента. Це дозволяє реалізувати глибоку навігацію, багатосторінкові форми, багаторівневі панелі керування без потреби оновлювати сторінку повністю. Такий механізм суттєво покращує взаємодію та дозволяє зменшити навантаження на сервер, оскільки логіка відображення контенту частково переноситься на клієнт.

У контексті використання React, архітектура SPA виглядає цілком природною. React забезпечує чіткий розподіл інтерфейсної логіки, дає змогу будувати компоненти будь-якої складності, а в поєднанні з React Router, Redux або Context API реалізовує навігацію, глобальний стан, кешування та реактивність. Завдяки підтримці SSR через Next.js, React-додатки можуть бути одночасно і SPA, і SEO-френдлі рішеннями. Це робить React-орієнтовані SPA одними з найкращих варіантів для реалізації інтерфейсів у сферах електронної комерції, онлайн-освіти, інформаційних панелей, особистих кабінетів, CRM-систем тощо.

Попри наявність деяких викликів — таких як безпека, управління початковим станом або потреба в оптимізації першого завантаження — SPA залишаються ключовим підходом у сучасній розробці. Вони особливо ефективні в контекстах, де

потрібна максимальна інтерактивність, складна навігація та персоналізація. Еволюція клієнтських інструментів і зростання обчислювальної потужності браузерів лише посилюють позиції SPA як архітектури, що формує нове покоління веб-досвіду.

Таким чином, модель Single Page Application стала однією з базових архітектурних концепцій сучасного вебу. Її практичне застосування доводить, що саме SPA, особливо у поєднанні з бібліотекою React, здатні забезпечити необхідний рівень продуктивності, гнучкості, адаптивності та зручності розробки, що робить її оптимальним вибором для побудови складних клієнтських інтерфейсів нового покоління.

2 ПРОЕКТУВАННЯ ВЕБ-ЗАСТОСУНКУ

2.1 ПОСТАНОВКА ЗАДАЧІ ТА ВИЗНАЧЕННЯ ФУНКЦІОНАЛУ

Проектування архітектури веб-застосунку для пошуку та каталогізації кінофільмів охоплює формалізоване визначення його структурної моделі, функціонального поділу відповідальності та логіки взаємодії між усіма складовими системи. Основна увага приділяється організації взаємодії між клієнтською та серверною частинами відповідно до принципів клієнт–серверної архітектури. Як базову архітектурну модель обрано концепцію одно сторінкового застосунку (SPA), яка забезпечує безперервну інтерактивність інтерфейсу, виключаючи необхідність повного перезавантаження сторінки при зміні вмісту. Це особливо важливо в контексті проектування застосунку, орієнтованого на високу частоту взаємодії з інтерфейсом, типову для пошукових сервісів.

2.2 ПРОЄКТУВАННЯ АРХІТЕКТУРИ ЗАСТОСУНКУ

Архітектурне проектування веб-застосунку для пошуку та каталогізації кінофільмів базується на поєднанні клієнт-серверної моделі з одно сторінковою архітектурою (SPA), що дозволяє забезпечити динамічність, швидкодію та масштабованість рішення. Основним завданням на цьому етапі було обґрунтоване вибудовування логічної структури системи, що дозволяє ефективно взаємодіяти з користувачем, зберігати дані, інтегрувати зовнішні джерела інформації та підтримувати гнучке розширення функціоналу у майбутньому.

Клієнтська частина системи реалізується засобами бібліотеки React.js, яка дозволяє компонувати інтерфейс на основі багаторазово використовуваних модулів (компонентів). Саме цей підхід забезпечує гнучкість побудови UI, спрощення супроводу, пришвидшення розробки та ефективну організацію внутрішньої логіки застосунку. Реалізація SPA-архітектури у поєднанні з React дає змогу відмовитися від повного перезавантаження сторінки при кожній зміні контенту, що значно покращує користувацький досвід.

З боку сервера використовується середовище Node.js у поєднанні з Express — це дозволяє ефективно обробляти HTTP-запити, захищати конфіденційні дані (зокрема API-ключ), а також кешувати часті запити для підвищення продуктивності. Сервер виконує роль посередника між фронтендом і зовнішнім API (The Movie

Database), надаючи централізований контроль над логікою запитів і можливість масштабування в майбутньому.

Архітектура побудована із урахуванням чіткого розподілу обов’язків: клієнт відповідає за взаємодію з користувачем та відображення інформації, сервер — за логіку обміну даними з API. Такий поділ не лише підвищує стабільність системи, а й дає змогу незалежно вдосконалювати обидві частини без порушення загальної функціональності.

Функціональна структура застосунку передбачає наявність основних модулів — сторінки з добірками, інтерфейс пошуку, екран з деталями фільму та особистий розділ користувача. Ці сторінки реалізовані як окремі компоненти React, які зв’язані через внутрішню маршрутизацію (React Router). Для управління глобальним станом, включно з локальними добірками та обраними фільмами, застосовується Context API, що забезпечує централізовану обробку даних без надмірної вкладеності пропсів.

Інтеграція концепції компонентної з ієрархічною структурою інтерфейсу дозволяє дотримуватись архітектурного шаблону MVC. У цьому контексті компоненти виступають як View, стан та локальні сховища — як Model, а обробники подій користувача реалізують функціонал Controller. Така структура сприяє прозорості та гнучкості в управлінні застосунком.

Загалом, запропонована архітектура демонструє відповідність сучасним вимогам до веб-застосунків: вона дозволяє масштабувати систему, легко додавати нові функції, підтримує автономність роботи та інтеграцію з API. Чіткий розподіл відповідальності і дотримання принципів модульності створюють надійну основу для реалізації першої робочої версії продукту (MVP) з урахуванням майбутніх оновлень і вдосконалень.

2.3. Вибір інструментів і технологій

2.3.1 Загальні принципи вибору

Розробка веб-застосунку для пошуку та каталогізації кінофільмів вимагає використання сучасного, гнучкого і масштабованого технологічного стеку. У зв'язку з потребою забезпечити швидку взаємодію з динамічними зовнішніми джерелами (зокрема API), побудову реактивного інтерфейсу користувача та підтримку

автономного збереження даних, було прийнято рішення використовувати єдину мову програмування — JavaScript — як на клієнтському, так і на серверному рівнях. Це забезпечує уніфіковану логіку, спрощує інтеграцію компонентів системи та знижує технічні ризики.

Ключовим архітектурним рішенням стала реалізація застосунку за моделлю SPA (Single Page Application), що дозволяє мінімізувати навантаження на сервер, прискорити час реакції інтерфейсу та забезпечити безперервність взаємодії з користувачем. Такий підхід є найбільш ефективним для динамічних інформаційно-довідкових систем.

2.3.2 Інструменти клієнтської частини (Frontend)

У реалізації клієнтської частини було обрано React — бібліотеку для побудови SPA, яка ґрунтується на компонентному підході. Вона дозволяє створювати повторно використовувані блоки інтерфейсу та забезпечує високу ефективність рендерингу завдяки використанню віртуального DOM. Для організації маршрутизації використовується React Router, що дозволяє користувачеві перемикатися між логічними розділами («Головна», «Деталі фільму», «Улюблене») без повного перезавантаження сторінки.

Для стилізації застосовується нативний CSS, що забезпечує повний контроль над виглядом інтерфейсу без додаткових залежностей від зовнішніх CSS-фреймворків. Такий підхід сприяє мінімізації розміру збірки та підвищенню продуктивності.

2.3.3 Інструменти серверної частини (Backend)

Для серверної логіки обрано середовище Node.js із фреймворком Express, що дозволяє ефективно обробляти запити та реалізовувати проксі-доступ до зовнішніх API. Серверна частина виконує функцію посередника між клієнтом і API The Movie Database (TMDB), захищаючи конфіденційні дані (зокрема API-ключі) та реалізуючи кешування запитів для підвищення швидкодії.

API TMDB обрано як основне джерело даних про фільми, оскільки воно надає широкий набір інформації: метадані, постери, жанри, трейлери, рейтинг тощо. Документація TMDB сприяє швидкій інтеграції та стабільній роботі сервісу.

Кешування реалізовано на рівні сервера з використанням відповідних інструментів, що знижує кількість повторних звернень до зовнішніх джерел і забезпечує стабільність навіть при високому навантаженні.

2.3.4 Оцінка альтернативних рішень

На етапі вибору стеку були проаналізовані альтернативні варіанти:

Технологія / Фреймворк	Недоліки у контексті проєкту
Vue.js	Потребує адаптації до обраної рхітектури
Angular	Надмірно складний для невеликого PA
Python + Django	Необхідність окремої мови для бекенду
Bootstrap	Недоцільний через просту структуру I
Next.js	Оптимізований під SSR, але не під SPA

Результати аналізу підтвердили доцільність обраного підходу: React як гнучка бібліотека для фронтенду, Express як мінімалістичний фреймворк для Node.js, і TMDB API як надійне джерело даних.

2.3.5 Взаємодія клієнта і сервера

Клієнт і сервер взаємодіють за принципами REST. Серверний компонент не лише забезпечує зв'язок із зовнішніми API, але й виконує функції захисту (ізоляція API-ключів), кешування популярних запитів, централізованої обробки помилок і контролю параметрів. Така модель дозволяє гнучке масштабувати застосунок, додаючи нові сервіси, функціональність або інтеграції без змін у клієнтській логіці.

Таким чином, обраний набір технологій є обґрунтованим з погляду технічної доцільності, простоти підтримки, продуктивності та гнучкості, необхідної для динамічного застосунку з перспективою масштабування.

2.4. Розробка структури компонентів застосунку

Реалізація веб-застосунку для пошуку, фільтрації та перегляду фільмів здійснюється на основі архітектури клієнт-сервер та з використанням API The Movie Database (TMDb). Архітектура системи базується на поєднанні сучасних технологій веб-розробки (React, Node.js, REST API) та принципах модульності й масштабованості. Особлива увага приділяється логічному розділенню компонентів і чіткій організації структури застосунку.

На основі об'єктно-орієнтованого підходу було створено UML-діаграму класів, яка ілюструє основні сутності системи, їхні атрибути, методи і взаємозв’язки між ними. Вона охоплює як клієнтські, так і серверні частини. Попри використання функціональних компонентів у React, у межах UML вони умовно представлені як класи для наочності логіки й функціональності.

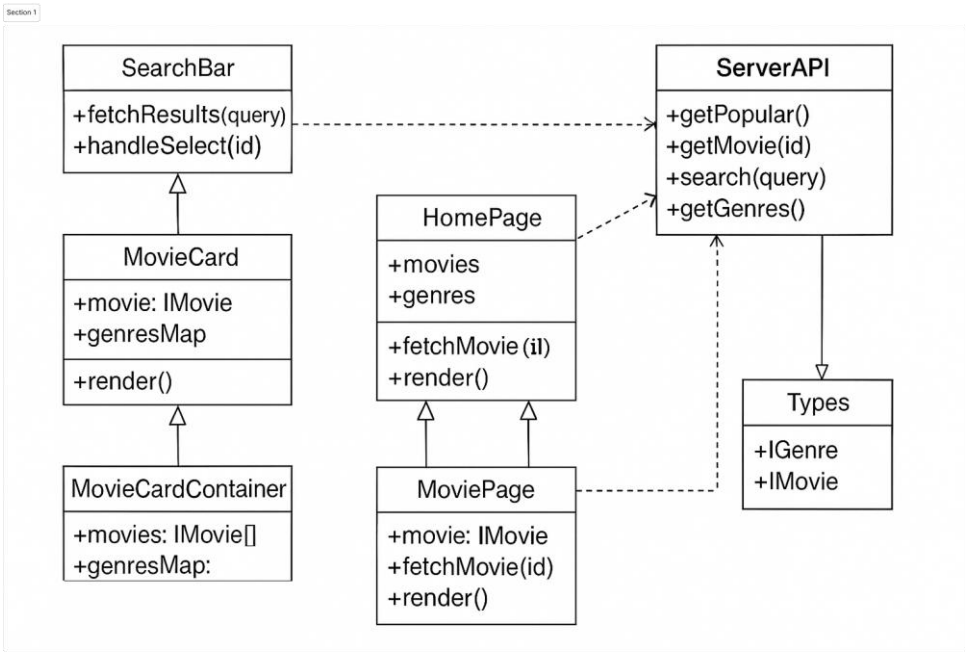

Рис. 3.1. UML-діаграма класів веб-застосунку для пошуку та перегляду фільмів.

Основні компоненти застосунку (див. рис. 3.1):

- SearchBar — реалізує механізм пошуку фільмів із функцією автодоповнення. Містить методи fetchResults(query) для запиту до /search і handleSelect(id) для переходу на сторінку конкретного фільму.
- MovieCard — представляє інтерфейсну картку фільму. Має атрибути movie: IMovie і genresMap, а також метод render() для візуалізації даних.

- MovieCardContainer — контейнерний компонент, що містить список об'єктів IMovie[] і genresMap, відповідає за фільтрацію і передачу даних до MovieCard.
- HomePage — головна сторінка, що відображає популярні фільми і жанри, реалізує метод fetchMovie(id) та render().
- MoviePage — сторінка для детальної інформації про окремий фільм. Має доступ до movie: IMovie і методи fetchMovie(id) та render().
- ServerAPI — умовний серверний клас, що містить методи getPopular(), getMovie(id), search(query) і getGenres() для взаємодії з API TMDb.
- Types — представляє інтерфейси IMovie та IGenre, що використовуються в усьому застосунку для типізації даних.

Всі взаємозв’язки між класами вказані у вигляді UML-діаграми: виклики API представлені пунктирними стрілками, а наслідування і використання — звичайними стрілками.

Структура файлової організації проєкту:

- components/ — містить компоненти загального призначення: MovieCard.tsx, MovieCardContainer.tsx;
- pages/ — реалізація SPA-маршрутів: HomePage.tsx, MoviePage.tsx;
- types/ — визначення типів даних: IMovie.ts, IGenre.ts;
- App.js — основний контейнер застосунку з конфігурацією роутів і глобального стану;
- index.js — вхідна точка програми.

Переваги реалізації:

- Компонентність — забезпечує повторне використання та легке тестування;
- Архітектурна чіткість — завдяки UML-діаграмі класів структура проєкту стає зрозумілою для всієї команди;
- Серверна абстракція — проксування запитів через ServerAPI підвищує безпеку й дозволяє легко змінювати джерело даних;

- Типізація з TypeScript — зменшує кількість помилок та полегшує розробку;
- Масштабованість і підтримуваність — структура дозволяє безболісно додавати нові компоненти або маршрути.

Таким чином, представлена реалізація веб-застосунку ґрунтується на сучасних інженерних практиках: від типізованої клієнтської логіки до централізованого серверного доступу до API, що в комплексі створює надійний, гнучкий і масштабований продукт.

3 РОЗРОБКА ВЕБ-ЗАСТОСУНКУ

3.1 Інструменти та засоби розробки

У процесі реалізації веб-застосунку для пошуку та каталогізації кінофільмів було створено комплексне середовище розробки, орієнтоване на ефективність, зручність і підтримку коду на всіх етапах життєвого циклу. Під час налаштування середовища враховувались специфіка SPA-архітектури, вимоги до продуктивності, а також потреба у швидкому тестуванні змін і повторному використанні компонентів інтерфейсу.

Для роботи з кодом було обрано IDE WebStorm, яка забезпечила повноцінну інтеграцію з технологіями React, TypeScript і Node.js. Серед її ключових переваг — автодоповнення, валідація синтаксису в реальному часі, зручна інтеграція з Git, підтримка шаблонів і фрагментів коду, а також вбудований термінал. Це дозволило значно пришвидшити розробку і зробити її структурованою на рівні всієї команди.

Клієнтська частина проекту була ініціалізована за допомогою create-react-app — інструменту, який автоматично налаштовує сучасне середовище для розробки SPA. Він включає підтримку JSX, Babel, Webpack, а також hot module replacement — функцію, що дозволяє оновлювати інтерфейс без повного перезавантаження. Це сприяє зменшенню часу циклів тестування й пришвидшує інтерактивну перевірку змін у коді.

На серверному рівні було використано Node.js разом із фреймворком Express для побудови API-проксі, через який відбувається взаємодія із зовнішнім сервісом TMDB. Для здійснення HTTP-запитів застосовано бібліотеку node-fetch, яка забезпечила асинхронну комунікацію та обробку зовнішніх відповідей. Реалізація кешування на стороні сервера відбулася з використанням бібліотеки node-cache — вона дозволила зберігати часто повторювані запити в оперативній пам’яті, що зменшило кількість звернень до TMDB і підвищило швидкодію застосунку.

Окрему увагу приділено організації безпечного зберігання конфігураційних параметрів. Для цього використано бібліотеку dotenv, яка дозволила відокремити конфіденційні дані (наприклад, API-ключі) у .env-файл, що не включається до

публічного репозиторію. Це відповідає сучасним стандартам безпеки і дозволяє легко перемикатись між середовищами (локальна розробка, тестування, продакшн).

Для керування залежностями використовується npm — стандартний пакетний менеджер для Node.js, який автоматизує встановлення, оновлення й документування зовнішніх бібліотек. Всі залежності фіксуються у файлі package.json, що дозволяє відтворити середовище розробки на будь-якому іншому пристрої без додаткових налаштувань.

Контроль якості коду забезпечували засоби статичного аналізу: ESLint для перевірки синтаксису та логіки, а також Prettier — для автоматичного форматування коду відповідно до заданого стилю. Ці інструменти допомогли уникнути людських помилок, знизити кількість конфліктів під час командної роботи та забезпечити єдині стандарти написання.

Загалом сформоване середовище розробки дозволило забезпечити швидку інтеграцію всіх частин проекту, стабільну роботу інтерфейсу, простоту масштабування й високий рівень керованості кодовою базою. Раціональне поєднання обраних інструментів створило технічну основу для якісної реалізації проекту з можливістю подальшого розвитку та підтримки.

Категорія	**Інструмент**	**Призначення**
IDE / едактор коду	WebStorm	Розробка, автодоповнення, Git-теграція, термінал
Ініціалізаці проєкту	create-act-app	Налаштування структури SPA, онфігурація Webpack, Babel, HMR
Фронтенд	React	Побудова компонентного терфейсу
	React outer	Внутрішня маршрутизація SPA
	CSS	Стилізація компонентів

Бекенд	Node.j + Express	Обробка HTTP-запитів, аршрутизація, проксі до TMDB API
	node-tch	Виконання зовнішніх HTTP-апитів
	node-ache	Кешування відповідей із овнішнього API
Конфігурац	dotenv	Зберігання змінних середовища API-ключів тощо)
Керування алежностями	npm	Інсталяція, оновлення та іксація зовнішніх бібліотек
Якість коду	ESLint	Статичний аналіз коду
	Prettie	Автоматичне форматування ідно зі стилем проєкту

Таблиця 3.1

3.2 Принцип побудови архітектури

У практичному втіленні архітектури веб-застосунку ключовим завданням стало не лише реалізувати спроєктовану логіку, а й перевірити її дієвість у реальних умовах. Якщо етап проєктування окреслював загальні принципи організації структури системи, то на цьому етапі йдеться про конкретні технічні механізми, які забезпечують стабільну роботу застосунку на всіх рівнях.

Клієнтська частина побудована за принципами компонентної моделі React, де кожна функціональна секція — пошук, деталізація фільмів, добірки — реалізована як окремий, ізольований компонент. Для кращої підтримуваності й зрозумілої логіки компоненти умовно поділено на два типи: контейнерні, що відповідають за обробку стану, запитів і логіки, та презентаційні, що займаються лише відображенням даних. Комунікація між ними реалізована через props або глобальний контекст за допомогою Context API, що дає змогу уникнути дублювання даних і зменшує кількість зайвих оновлень інтерфейсу.

Уся маршрутизація реалізована за допомогою бібліотеки React Router, з підтримкою динамічного імпорту компонентів. Завдяки використанню React.lazy у поєднанні з Suspense, реалізовано механізм лінивого завантаження (lazy loading) — тобто підвантаження лише тих компонентів, які потрібні в конкретний момент. Це не тільки скорочує час першого завантаження застосунку, а й покращує взаємодію на пристроях із повільним з'єднанням, зменшуючи навантаження на систему.

На серверному боці реалізовано багаторівневу логіку, побудовану за схемою «маршрути – контролери – сервіси», що є типовою для архітектури на основі Node.js з Express. Така модель дозволяє гнучко керувати запитами, розділяючи технічну реалізацію на незалежні модулі. Зовнішній API TMDB не викликається безпосередньо з клієнта — усі запити проходять через проксі-сервер, який виконує перевірку параметрів, обробку помилок, форматування результатів та кешування.

Для зменшення навантаження на API і пришвидшення відповіді використовується бібліотека node-cache. Кешування поширюється на часто повторювані запити — наприклад, списки популярних фільмів чи жанрів — і дозволяє зменшити затримки при відображенні одних і тих самих даних. У парі з локальним збереженням даних на клієнті через LocalStorage це створює багаторівневу систему кешування, яка забезпечує стабільну роботу навіть у разі обмеженого доступу до мережі.

Особливу увагу приділено розширюваності архітектури. Завдяки модульному підходу реалізація нових можливостей, таких як авторизація, реєстрація користувача, персональні добірки або синхронізація між пристроями, може бути здійснена без модифікації існуючого функціоналу. Достатньо лише додати новий middleware або контролер — без необхідності переписувати клієнтську частину чи порушувати вже налагоджені залежності. Це гарантує ізоляцію змін і підвищує надійність системи під час масштабування.

З логічного боку архітектура базується на адаптованій моделі MVC, що гармонійно вписується в концепцію React. Компоненти виступають у ролі View, стан (глобальний або локальний) виконує функції Model, а обробники подій і запитів

відповідають за Controller. Такий підхід забезпечує структурованість і прозорість у логіці, спрощує налагодження, тестування, і підвищує загальну керованість проектом.

Загалом архітектура проекту демонструє ефективне поєднання продуманої структури, технологічної гнучкості та практичної стійкості до змін. Вона адаптивна до розширення функціональності й відповідає сучасним вимогам до побудови масштабованих SPA-застосунків, забезпечуючи фундамент для подальшого розвитку продукту без загрози стабільності або складності супроводу.

3.3 Реалізація програмного веб-застосунку

Розглянемо детальну реалізацію клієнт-серверного веб-застосунку для пошуку, фільтрації та перегляду фільмів на основі API The Movie Database (TMDb). Архітектура системи поєднує сучасні веб-технології (React, Node.js, REST API) та демонструє розділення на логічні компоненти. Після побудови діаграми класів та опису основних логічних сутностей, доцільно перейти до розгляду безпосередньої реалізації програмного застосунку на рівні вихідного коду. Детальний аналіз структури проекту дозволяє прослідкувати, яким чином концептуальні компоненти, представлені на UML-діаграмі, реалізовані у вигляді функціональних модулів, типів та компонентів у середовищі JavaScript/TypeScript. Надалі буде розглянуто початкові файли клієнтської частини, які відповідають за запуск, навігацію, візуальне представлення фільмів та інтеграцію з серверною логікою. Такий поетапний розбір дозволяє провести відповідність між проектною моделлю системи та її практичною реалізацією на рівні коду.

Клієнтська частина:

Опис компонента index.js

Одним із базових компонентів клієнтської частини веб-застосунку є файл index.js, який виступає точкою входу (entry point) до всього інтерфейсу. Саме з цього модуля починається ініціалізація React-додатку, створення кореневого елементу DOM, підключення глобальних тем, базових стилів, а також запуск головного компонента — App.

На рис. 3.1 зображено фрагмент коду, що відповідає за створення кореневого вузла застосунку. Спочатку до проєкту імпортуються необхідні залежності: React, ReactDOM, глобальні стилі index.css, головний компонент App, а також інструменти з бібліотеки Material UI — зокрема, ThemeProvider, createTheme та CssBaseline. Останні відповідають за темізацію застосунку та скидання стилів браузера до узгодженого вигляду.

У центрі коду оголошується константа theme, яка створює об'єкт теми з палітрою у темному режимі (mode: 'dark'). Даний об’єкт передається до ThemeProvider, що обгортає кореневий компонент App і надає всім дочірнім елементам доступ до параметрів теми. Також використовується компонент CssBaseline, який забезпечує однаковий вигляд інтерфейсу у різних браузерах.

Сам рендеринг здійснюється за допомогою ReactDOM.createRoot(...) — сучасного способу створення кореневого вузла в React 18. Цей метод ініціалізує дерево компонентів у контейнері з ідентифікатором root, що знаходиться у HTML-файлі (public/index.html). На завершення, додатково підключається reportWebVitals() — функція, що теоретично дозволяє вести вимірювання продуктивності застосунку, хоча в межах цього проєкту вона не використовується активно.

Таким чином, файл index.js відповідає за запуск застосунку, застосування глобальних стилів і тем, а також за передачу управління головному маршрутизованому компоненту App.

```
import React from 'react';
import ReactDOM from 'react-dom/client';
import './index.css';
import App from './App';
import reportWebVitals from './reportWebVitals';
import { ThemeProvider, createTheme, CssBaseline } from '@mui/material';

const theme = createTheme( options: {
    palette: {
        mode: 'dark',
    },
});

const root = ReactDOM.createRoot(document.getElementById( elementId: 'root'));
root.render(
  <React.StrictMode>
      <ThemeProvider theme={theme}>
          <CssBaseline />
    <App />
      </ThemeProvider>
  </React.StrictMode>
);

// If you want to start measuring performance in your app, pass a function
// to log results (for example: reportWebVitals(console.log))
// or send to an analytics endpoint. Learn more: https://bit.ly/CRA-vitals
reportWebVitals();
```

Рис. 3.1 Код для index.js у клієнті застосунку

Опис компонента маршрутизації App.js(Сервер)

Компонент App, описаний у файлі App.js, є центральним маршрутизатором усього клієнтського інтерфейсу. Його головне завдання — визначення шляхів (routes) у межах односторінкового застосунку (SPA) та відображення відповідних сторінок залежно від маршруту.

Компонент побудовано з використанням бібліотеки react-router-dom, яка надає зручні засоби маршрутизації у React. У верхній частині файла App.js відбувається імпорт ключових елементів: BrowserRouter, Routes, Route, а також окремих сторінкових компонентів — HomePage та MoviePage.

BrowserRouter (псевдонім Router) виступає контейнером для системи навігації, забезпечуючи відстеження змін у адресному рядку браузера. Внутрішньо

використовується API history, який дозволяє перемикати компоненти без перезавантаження сторінки.

Далі, за допомогою конструкції Routes і Route, задаються два основні маршрути:

1. **/HomePage** — головна сторінка застосунку, яка відповідає за відображення популярних фільмів, жанрів, пошуку та фільтрації.
2. **/movie/:id MoviePage** — сторінка з детальною інформацією про конкретний фільм. Динамічна частина маршруту :id дозволяє передавати унікальний ідентифікатор фільму та отримувати дані з серверу відповідно до нього.

Таким чином, файл App.js виконує функцію маршрутизатора між ключовими логічними частинами клієнтської частини програми. Він ізольований від бізнес-логіки та зосереджений виключно на навігації, що відповідає принципам розділення відповідальності в проєктуванні інтерфейсів.

```jsx
import React from 'react';
import { BrowserRouter as Router, Routes, Route } from 'react-router-dom';
import HomePage from './pages/HomePage';
import MoviePage from './pages/MoviePage';

const App = () => {
    return (
        <Router>
            <Routes>
                <Route path="/" element={<HomePage />} />
                <Route path="/movie/:id" element={<MoviePage />} />
            </Routes>
        </Router>
    );
};

export default App;
```

Рис. 3.2

Опис компонента MovieCardContainer.tsx

Компонент MovieCardContainer виконує роль контейнера для виведення колекції фільмів у вигляді окремих карток (MovieCard) та забезпечує функціонал фільтрації за жанрами. Його структура відповідає вимогам до інтерфейсу

користувача, де на головній сторінці має відображатись перелік фільмів з можливістю їх сортування за жанровими категоріями.

На початку компонента оголошується стан selectedGenre, який зберігає ідентифікатор поточного вибраного жанру. За його допомогою реалізується фільтрація масиву movies, що передається як пропс. Якщо жоден жанр не вибрано, користувач бачить повний список; якщо жанр обраний — список обмежується лише тими фільмами, у яких масив genre_ids містить відповідний ідентифікатор.

Далі у компоненті виводяться кнопки жанрів, згенеровані з унікальних жанрових ідентифікаторів, що присутні в переданих фільмах. Для кожної кнопки використовується стилізований компонент Button з бібліотеки Material UI, що автоматично змінює вигляд при виборі жанру.

Основна частина компонента — візуальне відображення фільмів. Для кожного фільму, що відповідає умовам фільтрації, створюється елемент <MovieCard />, обгорнутий у компонент Link для забезпечення переходу до детальної сторінки фільму за маршрутом /movie/:id. Кожна картка має фіксовану ширину, що дозволяє гнучко компонувати їх у сітку за допомогою Flexbox (Box з flexWrap).

Таким чином, компонент MovieCardContainer реалізує динамічний інтерфейс зі зручною фільтрацією контенту, використовуючи принципи реактивного оновлення стану та компонентного відображення вмісту.

```tsx
import React, { useState } from 'react';
import { IMovie } from '../types/IMovie';
import { Box, Button, Stack, Typography } from '@mui/material';
import MovieCard from './MovieCard';
import { Link } from 'react-router-dom';

interface Props {
    movies: IMovie[];
    genresMap: Record<number, string>; // id => название
}

const MovieCardContainer: React.FC<Props> = ({ movies : IMovie[] , genresMap : Record<number, string> }) => {
    const [selectedGenre, setSelectedGenre] = useState<number | null>( initialState: null);

    const filteredMovies =
        selectedGenre === null ? movies : movies.filter((m : IMovie ) => m.genre_ids.includes(selectedGenre));

    const uniqueGenres = Array.from(
        new Set(movies.flatMap( callback: (movie : IMovie ) => movie.genre_ids))
    ).filter((id : number ) => genresMap[id]);
```

Рис. 3.3

```
    return (
        <Box sx={{ padding: 2 }}>
            <Typography variant="h6" gutterBottom>
                Фільтрація по жанрам:
            </Typography>
            <Stack direction="row" spacing={1} mb={2} flexWrap="wrap">
                <Button
                    variant={selectedGenre === null ? 'contained' : 'outlined'}
                    onClick={() => setSelectedGenre( value: null)}
                >
                    Усі
                </Button>
                {uniqueGenres.map((id : number ) => (
                    <Button
                        key={id}
                        variant={selectedGenre === id ? 'contained' : 'outlined'}
                        onClick={() => setSelectedGenre(id)}
                    >
                        {genresMap[id]}
                    </Button>
                ))}
            </Stack>
```

Рис. 3.4

```
            <Box
                sx={{
                    display: 'flex',
                    flexWrap: 'wrap',
                    gap: 3,
                    justifyContent: 'center',
                }}
            >
                {filteredMovies.map((movie : IMovie ) => (
                    <Box key={movie.id} sx={{ flex: '1 1 250px', maxWidth: 250 }}>
                        <Link to={`/movie/${movie.id}`} style={{ textDecoration: 'none' }}>
                        <MovieCard movie={movie} genresMap={genresMap} />
                        </Link>
                    </Box>
                ))}
            </Box>
        </Box>
    );
};

export default MovieCardContainer;
```

Рис. 3.5

Опис компонента MovieCard.tsx

Компонент MovieCard є візуальною одиницею, що відображає окремий фільм у вигляді картки з зображенням, назвою, рейтингом, роком виходу та жанрами. Він

отримує через пропси об’єкт movie типу IMovie та необов’язкову карту genresMap, що зіставляє числові genre_ids із текстовими назвами жанрів.

Основу компонента складає компонент Card з бібліотеки Material UI, що містить:

- CardMedia — компонент для відображення постера фільму, який завантажується з TMDb за прямим URL.
- CardContent — блок з текстовою інформацією:
  - назва фільму (movie.title);
  - рейтинг (movie.vote_average);
  - рік релізу, обчислений з поля release_date методом getFullYear().

Нижче відображається список жанрів у вигляді Chip-елементів (маленьких міток), які рендеряться лише у випадку, якщо відповідний жанр існує в переданій genresMap. Це дозволяє відображати жанри у зрозумілій формі, навіть якщо вони передаються як числові ID.

Картка має фіксовану ширину та висоту, що дозволяє забезпечити єдиний стиль для всієї колекції фільмів. Її можна легко стилізувати під темну або світлу тему інтерфейсу завдяки спадкуванню стилів від глобальної теми MUI.

Компонент MovieCard є універсальним і може бути повторно використаний в різних частинах застосунку — наприклад, у списках популярних фільмів, результатах пошуку чи персоналізованих добірках.

```
import React from 'react';
import { IMovie } from '../types/IMovie';
import { Card, CardContent, CardMedia, Typography, Chip, Stack } from '@mui/material';

const MovieCard: React.FC<{ movie: IMovie; genresMap?: Record<number, string> }> =
    ({ movie : IMovie , genresMap : Record<number, string>  = {} }) => {
    return (
        <Card sx={{ maxWidth: 250, height: '100%' }}>
            <CardMedia
                component="img"
                height="370"
                image={`https://image.tmdb.org/t/p/w500${movie.poster_path}`}
                alt={movie.title}
            />
```

Рис. 3.6

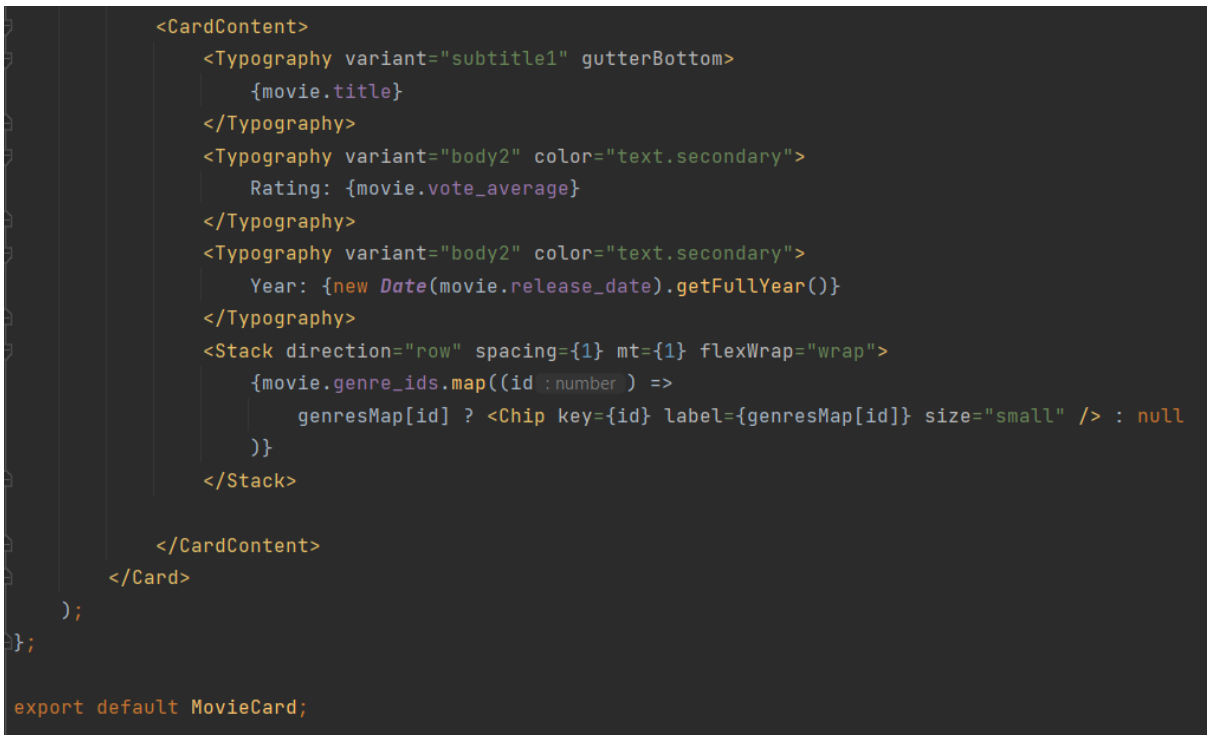

Рис. 3.7

Опис компонента SearchBar.tsx

Компонент SearchBar реалізує функцію пошуку фільмів за назвою у реальному часі з підтримкою автозаповнення та переходом до детальної сторінки обраного фільму.

Ключовим елементом є поле вводу (TextField), у якому користувач вводить запит. Значення цього поля зберігається у стані query. Після кожної зміни значення запиту викликається метод debouncedFetch, реалізований з використанням бібліотеки lodash/debounce. Такий підхід дозволяє зменшити навантаження на сервер, затримуючи запити до API до моменту, коли користувач завершить введення.

Функція fetchResults формує HTTP-запит до серверного маршруту /search, передаючи параметр query. Отриманий список фільмів зберігається у стані results.

Якщо результати пошуку не порожні, під полем вводу з’являється список знайдених фільмів (List), кожен з яких представлений елементом ListItemButton. При натисканні на конкретний фільм викликається функція handleSelect(id), яка перенаправляє користувача на сторінку фільму за допомогою useNavigate з react-router-dom, а також очищує запит та результати.

Компонент SearchBar оформлено за допомогою стилів Material UI — зокрема, Paper, List та ListItemText, що дозволяє досягти сучасного вигляду без додаткового CSS.

Цей компонент є інтерактивним, компактним, адаптованим до мобільних пристроїв та може бути розміщений у будь-якому місці інтерфейсу (наприклад, на головній сторінці або в шапці).

```
import React, { useState, useEffect } from 'react';
import { TextField, Paper, List, ListItemText, ListItemButton } from '@mui/material';
import { useNavigate } from 'react-router-dom';
import debounce from 'lodash/debounce';

const SearchBar = () => {
    const [query, setQuery] = useState( initialState: '');
    const [results, setResults] = useState( initialState: []);
    const navigate = useNavigate();
    const fetchResults = async (search: string) => {
        if (!search) return setResults( value: []);
        try {
            const res = await fetch( input: `http://localhost:4000/search?query=${encodeURIComponent(search)}`);
            const data = await res.json();
            setResults( value: data.results || []);
        } catch (err) {
            console.error(err);
        }
    };
    const debouncedFetch = debounce(fetchResults, 400);
    useEffect( effect: () => {
        debouncedFetch(query);
        return debouncedFetch.cancel;
    }, deps: [query]);

    const handleSelect = (id: number) => {
        navigate( to: `/movie/${id}`);
        setQuery( value: '');
        setResults( value: []);
    };
```


Рис. 3.8

```
    return (
        <div style={{ position: 'relative', maxWidth: 400, margin: '20px auto' }}>
            <TextField
                fullWidth
                label="Пошук фільму"
                variant="outlined"
                value={query}
                onChange={(e ... ) => setQuery(e.target.value)}
            />
            {results.length > 0 && (
                <Paper sx={{ position: 'absolute', top: '100%', left: 0, right: 0, zIndex: 10 }}>
                    <List>
                        {results.map((movie: any) => (
                            <ListItemButton
                                key={movie.id}
                                onClick={() => handleSelect(movie.id)}
                            >
                                <ListItemText primary={movie.title} />
                            </ListItemButton>
                        ))}
                    </List>
                </Paper>
            )}
        </div>
    );
};

export default SearchBar;
```

Рис. 3.9

Опис інтерфейсу IMovie.ts

Інтерфейс IMovie визначає структуру об'єкта фільму, який використовується на клієнтському боці для відображення та логіки взаємодії з фільмами. Цей інтерфейс було створено для спрощення типізації даних, що надходять з зовнішнього API TMDb, а також для забезпечення типобезпеки при розробці на TypeScript.

Інтерфейс включає такі поля:

- id: number — унікальний числовий ідентифікатор фільму.
- title: string — назва фільму.
- poster_path: string — відносний шлях до зображення постера фільму. Повна адреса формується на основі базового шляху TMDb.
- vote_average: number — середній рейтинг фільму за версією користувачів TMDb.
- overview: string — короткий опис фільму.
- release_date: string — дата виходу фільму (наприклад, "2023-05-17").

- genre_ids: number[] — масив числових ідентифікаторів жанрів, до яких належить фільм.

Цей інтерфейс активно використовується в компонентах MovieCard, MovieCardContainer, MoviePage, а також у функціях обробки результатів запитів на сервер. Тип IMovie дозволяє відслідковувати правильність переданих даних та зменшує ймовірність помилок під час компіляції та виконання програми.

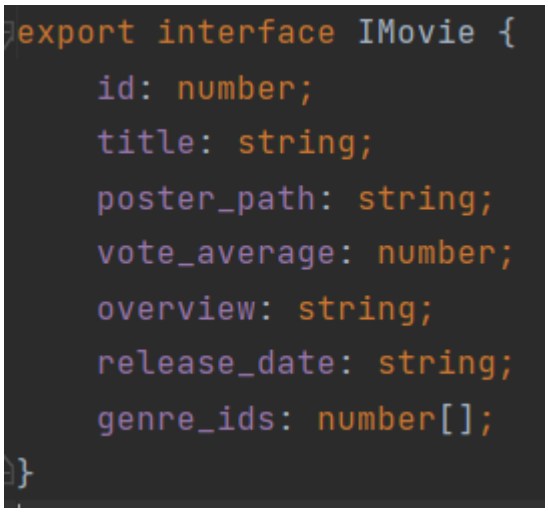

Рис. 3.10

Опис інтерфейсу IGenre.ts

Інтерфейс IGenre описує структуру об'єкта жанру фільму. Жанри надходять з TMDb API окремим списком і використовуються для формування мапи жанрів (genresMap), яка дозволяє відображати назви жанрів замість числових ID.

Інтерфейс містить два поля:

- id: number — числовий ідентифікатор жанру (наприклад, 28 для "Action").
- name: string — назва жанру англійською мовою (наприклад, "Comedy", "Drama" тощо).

Жанри зберігаються у масиві об'єктів IGenre[], який отримується через запит до маршруту /genres на сервері. Після отримання, масив конвертується у зручний формат мапи (Record<number, string>) для швидкого доступу до назв жанрів за їх ідентифікаторами.

Цей інтерфейс використовується переважно у таких компонентах, як MovieCardContainer (для фільтрації) та MovieCard (для відображення жанрів).

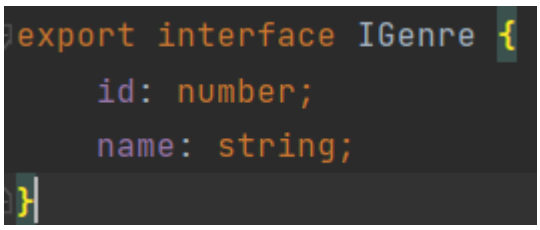

Рис. 3.11

Опис компонента HomePage.tsx

Компонент HomePage реалізує головну сторінку клієнтського інтерфейсу веб-застосунку. Його основна функція — відображення списку фільмів за замовчуванням (зокрема популярних), а також реалізація пошуку та фільтрації фільмів за жанрами. Структурно цей компонент виконує роль контейнера верхнього рівня, який об’єднує кілька інших дочірніх компонентів, зокрема SearchBar та MovieCardContainer.

У початковій частині оголошуються два стани:

- movies: IMovie[] — масив фільмів, отриманих з бекенду;
- genres: IGenre[] — масив жанрів, які також надходять із сервера.

Ці стани ініціалізуються у useEffect, що виконується один раз при монтуванні компонента. Всередині useEffect здійснюються два HTTP-запити:

1. до маршруту /popular, який повертає масив найпопулярніших фільмів;
2. до маршруту /genres, який повертає список жанрів з TMDb API.

Після отримання жанрів створюється структура genresMap типу Record<number, string>, яка дозволяє зіставити кожен числовий genre_id з відповідною текстовою назвою жанру. Це покращує читабельність інтерфейсу і дозволяє швидко здійснювати пошук за ключем у мапі.

У return JSX-розмітки послідовно підключаються два основні компоненти:

- <SearchBar /> — реалізує введення користувачем назви фільму та динамічне автозаповнення.
- <MovieCardContainer movies={movies} genresMap={genresMap} /> — відображає отримані фільми у вигляді сітки карток, дозволяючи також фільтрацію за жанрами.

Таким чином, HomePage.tsx є центральним компонентом користувацького інтерфейсу на головній сторінці, який агрегує основну логіку: запити до API, збереження стану, фільтрацію, пошук та візуалізацію даних.

```tsx
import React, { useEffect, useState } from 'react';
import MovieCardContainer from '../components/MovieCardContainer';
import { IMovie } from '../types/IMovie';
import { IGenre } from '../types/IGenre';
import SearchBar from '../components/SearchBar';

const HomePage = () => {
    const [movies, setMovies] = useState<IMovie[]>( initialState: []);
    const [genres, setGenres] = useState<IGenre[]>( initialState: []);

    useEffect( effect: () => {
        fetch( input: 'http://localhost:4000/popular') Promise<Response>
            .then((res : Response ) => res.json()) Promise<any>
            .then((data) => setMovies(data.results));

        fetch( input: 'http://localhost:4000/genres') Promise<Response>
            .then((res : Response ) => res.json()) Promise<any>
            .then((data) => setGenres(data.genres));
    }, deps: []);

    const genresMap: Record<number, string> = {};
    genres.forEach((g : IGenre ) => (genresMap[g.id] = g.name));
```

Рис. 3.12

```tsx
    return (
        <>
            <SearchBar />
            <MovieCardContainer movies={movies} genresMap={genresMap} />
        </>
    );
};

export default HomePage;
```

Рис. 3.13

Опис компонента MoviePage.tsx

Компонент MoviePage відповідає за відображення детальної інформації про один фільм. Він активується, коли користувач переходить за динамічним маршрутом /movie/:id, де :id — унікальний ідентифікатор фільму, переданий через URL.

Для зчитування параметру маршруту використовується хук useParams з бібліотеки react-router-dom. Отриманий id передається у виклик fetch(...) для звернення до бекенду за маршрутом /movie/:id, що у відповідь повертає повні дані про обраний фільм.

Відповідь записується у стан movie, який має тип IMovie | null. Поки дані не отримані, користувачу відображається повідомлення Loading.... Коли дані завантажено, відбувається рендеринг інтерфейсу з наступними елементами:

- Заголовок з назвою фільму (movie.title);
- Дата виходу та рейтинг (movie.release_date, movie.vote_average);
- Зображення постера, яке формується на основі поля poster_path з TMDb API;
- Короткий опис фільму (movie.overview).

Компонент MoviePage використовує стилі Material UI (Container, Typography, Box) для забезпечення адаптивної та читабельної верстки, яка підлаштовується під різні екрани. Постер зображення масштабовано за допомогою CSS-властивостей maxWidth та height: auto.

Таким чином, MoviePage.tsx реалізує повноцінну детальну сторінку, яка дає змогу користувачу переглянути розширену інформацію про обраний фільм, зберігаючи при цьому адаптивність та відповідність до сучасних стандартів UI.

```
import React, { useEffect, useState } from 'react';
import { useParams } from 'react-router-dom';
import { IMovie } from '../types/IMovie';
import { Container, Typography, Box } from '@mui/material';

const MoviePage = () => {
    const { id } = useParams<{ id: string }>();
    const [movie, setMovie] = useState<IMovie | null>( initialState: null);

    useEffect( effect: () => {
        fetch( input: `http://localhost:4000/movie/${id}`) Promise<Response>
            .then((res : Response ) => res.json()) Promise<any>
            .then((data) => setMovie(data)) Promise<void>
            .catch((err) => console.error(err));
    }, deps: [id]);

    if (!movie) return <Typography>Loading...</Typography>;
```

Рис. 3.14

```
    return (
        <Container sx={{ mt: 4 }}>
            <Typography variant="h4">{movie.title}</Typography>
            <Typography variant="subtitle1" color="text.secondary">
                Release date: {movie.release_date} — Rating: {movie.vote_average}
            </Typography>
            <Box sx={{ mt: 2 }}>
                <img
                    src={`https://image.tmdb.org/t/p/w500${movie.poster_path}`}
                    alt={movie.title}
                    style={{ maxWidth: '100%', height: 'auto' }}
                />
            </Box>
            <Typography variant="body1" sx={{ mt: 2 }}>
                {movie.overview}
            </Typography>
        </Container>
    );
};

export default MoviePage;
```

Рис. 3.15

Після розгляду клієнтської частини веб-застосунку, яка забезпечує візуалізацію даних, взаємодію з користувачем та маршрутизацію, доцільно перейти до опису серверної логіки. Серверна частина системи реалізована за допомогою середовища Node.js та бібліотеки Express.js і відповідає за обробку запитів від клієнта,

комунікацію з зовнішнім API TMDb, кешування відповідей, а також забезпечення безпеки доступу до ключів API. У рамках цієї частини дипломної роботи буде детально описано структуру головного серверного модуля, його маршрути, механізми кешування та логіку обробки даних. Це дозволить прослідкувати повний цикл взаємодії між клієнтським інтерфейсом та зовнішніми джерелами даних через власний серверний посередник.

Опис серверного модуля app.js

Файл app.js є основним серверним модулем застосунку, який реалізує логіку обробки запитів клієнтської частини, взаємодію із зовнішнім API (The Movie Database – TMDb), а також механізми кешування даних для зменшення навантаження та прискорення відповіді. Сервер побудовано за допомогою Node.js з використанням бібліотеки Express.js.

У верхній частині файлу здійснюється підключення ключових залежностей:

- dotenv — для зчитування секретного API-ключа з файлу .env;
- express — для створення маршрутизатора та обробки HTTP-запитів;
- cors — для дозволу міждоменної взаємодії клієнта з сервером;
- axios — для виконання запитів до TMDb API;
- node-cache — для зберігання тимчасових відповідей та реалізації кешу на рівні сервера.

```
require('dotenv').config();
const express = require('express');
const cors = require('cors');
const axios = require('axios');
const NodeCache = require('node-cache');
```

Рис 3.16

Базовий маршрут, який відповідає простим текстом:

/popular

Цей маршрут реалізує отримання списку популярних фільмів з TMDb. Він приймає параметр page (за замовчуванням — 1) і кешує відповідь за ключем

popular_page_{page}. Якщо відповідь уже збережена в кеші — вона повертається без повторного звернення до зовнішнього API.

```
app.get('/popular', async (req : … , res : Response<ResBody, LocalsObj> ) => {
  const page = req.query.page || 1;
  const cacheKey = `popular_page_${page}`;

  const cached = movieCache.get(cacheKey);
  if (cached) {
    console.log('Popular movies from cache');
    return res.json(cached);
  }

  try {
    const response = await axios.get( url: 'https://api.themoviedb.org/3/movie/popular', config: {
      params: {
        api_key: process.env.TMDB_API_KEY,
        language: 'en-US',
        page
      }
    });

    movieCache.set(cacheKey, response.data);
    console.log('Popular movies from TMDb API');
    res.json(response.data);
  } catch (error) {
    console.error('TMDb popular fetch error:', error.message);
    res.status( code: 500).json( body: { error: 'Failed to fetch popular movies' });
  }
});
```

Рис. 3.17

/movie/:id

Динамічний маршрут для отримання детальної інформації про фільм за його id. Кеш працює за ключем movie_{id}. Повертається повна інформація про фільм — назва, опис, дата виходу, жанри, рейтинг тощо.

Рис. 3.18

/genres

Повертає повний список жанрів фільмів з TMDb. Даний маршрут не приймає параметрів та використовується на клієнті для формування списку фільтрів і назви жанрів у картках.

Рис. 3.19

/search?query=...

Реалізує пошук фільмів за назвою, що вводить користувач у SearchBar. Запити кешуються за значенням параметра query. Відповідь повертає масив фільмів, які найбільше відповідають пошуковому запиту.

```
app.get('/search', async (req : … , res : Response<ResBody, LocalsObj> ) => {
  const query = req.query.query?.toLowerCase();
  if (!query) return res.status( code: 400).json( body: { error: 'Query is required' });

  const cached = searchCache.get(query);
  if (cached) {
    console.log('Search from cache:', query);
    return res.json(cached);
  }

  try {
    const response = await axios.get( url: 'https://api.themoviedb.org/3/search/movie', config: {
      params: {
        api_key: process.env.TMDB_API_KEY,
        query,
        language: 'en-US'
      }
    });

    searchCache.set(query, response.data);
    console.log('Search from TMDb:', query);
    res.json(response.data);
  } catch (err) {
    console.error('Search error:', err.message);
    res.status( code: 500).json( body: { error: 'Failed to search' });
  }
})
```

Рис. 3.20

Кешування

Для оптимізації продуктивності на сервері створено два об’єкти кешу:

- movieCache — для кешування результатів /popular, /movie/:id, /genres;
- searchCache — для кешування результатів /search.

Кеш реалізовано через бібліотеку node-cache з часом життя (TTL) 300 секунд (5 хвилин), після чого дані автоматично видаляються.

```
const movieCache = new NodeCache({ stdTTL: 300 }); // TTL = 5 хвилин
```

Рис. 3.21

```
const searchCache = new NodeCache({ stdTTL: 300 });
```

Рис. 3.22

Запуск сервера

На завершальному етапі вказано запуск сервера:

```
// Запуск сервера
app.listen(PORT, hostname: () => {
  console.log(` Server is running on http://localhost:${PORT}`);
});
```

Рис. 3.23

Далі ініціалізується об'єкт сервера через express() та вказується порт PORT = 4000, на якому він буде працювати. Підключаються middleware cors() та express.json() для обробки JSON-запитів і налаштування CORS-заголовків.

Цей виклик активує сервер і дозволяє клієнтському застосунку здійснювати до нього запити.

```
const app = express();
const PORT = 4000;
```

Рис. 3.24

```
app.use(cors());
app.use(express.json());
```

Рис. 3.25

Файл app.js реалізує повноцінну серверну логіку для веб-застосунку у вигляді REST API. Він служить проміжною ланкою між клієнтом і зовнішнім сервісом TMDb, додає кешування, обробку помилок та захист API-ключа. Такий підхід відповідає вимогам до сучасних клієнт-серверних архітектур і забезпечує гнучкість, масштабованість та розширюваність застосунку.

ВИСНОВКИ

В межах виконаної кваліфікаційної роботи було реалізовано повноцінний клієнт-серверний веб-застосунок, що забезпечує функціонал пошуку, перегляду та персональної каталогізації кінофільмів. Розробка велась відповідно до сучасних принципів створення SPA (Single Page Application) із використанням бібліотеки React для клієнтської частини та Node.js з Express для побудови серверної логіки.

У процесі дослідження проведено аналіз актуальних технологічних рішень у сфері веб-розробки, здійснено порівняльну характеристику популярних фреймворків JavaScript, обґрунтовано вибір стеку технологій, що забезпечує оптимальне співвідношення продуктивності, гнучкості, масштабованості та зручності підтримки. Особливу увагу приділено формалізації архітектурної моделі системи, проєктуванню структури інтерфейсу та реалізації ефективного механізму взаємодії з API зовнішньої бази даних TMDb.

На етапі реалізації було створено компонентно-орієнтовану клієнтську частину із підтримкою маршрутизації, глобального стану та фільтраційного механізму. Серверна частина виконувала функції проксі-сервера, відповідального за кешування, захист конфіденційного API-ключа та агрегацію запитів до стороннього джерела. Тестування підтвердило відповідність системи функціональним та нефункціональним вимогам, зокрема в аспектах швидкодії, стабільності та адаптивності інтерфейсу.

Результатом роботи є легкий у використанні, автономний веб-застосунок, що надає користувачам зручний інструмент для пошуку та впорядкування фільмографічного контенту. Побудована архітектура системи є гнучкою до розширення, що дозволяє у перспективі інтегрувати модулі авторизації, збереження даних у базі, багатомовності, а також додаткові джерела інформації.

Отримані під час виконання проєкту результати мають не лише практичну цінність у межах створення спеціалізованого інформаційного продукту, але й демонструють ефективність обраних методів і підходів у побудові сучасних інтерактивних веб-систем. Робота може бути використана як основа для подальших

наукових досліджень або приклад для розробки більш складних цифрових рішень у суміжних сферах.