6. Introduction to JSON Web Tokens. *Auth0 Docs*. URL: https://jwt.io/introduction/.
7. Спиваковський О. В., Щедролосєв Д. Є. Побудова безпечних розподілених веб-систем. Херсон : ХДУ, 2019. 150 с.

УДК 004.41

ПРОГРАМОТЕХНІКА ТА ПРОЄКТУВАННЯ КОМП'ЮТЕРНИХ СИСТЕМ: АРХІТЕКТУРА КЛІЄНТСЬКОГО РІВНЯ ТА РЕАКТИВНІ ПАТЕРНИ

Кондратенко Є.С.
ekqwert12345@gmail.com
Черкаський державний фаховий бізнес-коледж
Подорошко Д.І.
м. Черкаси, Україна

У сучасній розробці програмного забезпечення клієнтська частина (front-end) вже не є лише оболонкою для відображення даних. Великі односторінкові застосунки (SPA) та корпоративні системи стикаються з подібними архітектурними ризиками, як і серверні рішення: неконтрольоване зростання зв'язаності модулів, каскадні збої та деградацію підтримуваності з часом. Браузерні застосунки функціонують автономно, реалізуючи складні обчислення, структурні патерни та бізнес-логіку на рівні, що відповідає серверній частині [1]. Проте специфіка середовища виконання, зокрема обмежені ресурси пристрою, відкритість коду та непередбачуваність мережі, унеможливлює пряме перенесення серверних рішень без адаптації. Архітектурні рішення, прийняті на ранніх етапах розробки клієнтського застосунку, визначають його довгострокову підтримуваність і стійкість до збоїв [2].

Компонентний підхід і сучасні практики у фреймворку Angular. Розробка інтерфейсів корпоративного рівня вимагає чіткої модульності та відокремлення бізнес-логіки від логіки користувацького інтерфейсу. Angular демонструє застосування інженерних методів у створенні клієнтських систем [3]. На відміну від легких бібліотек, Angular використовує статичну типізацію (TypeScript). Це

дозволяє виявляти помилки на етапі компіляції й зменшує їх кількість під час виконання [4]. Основою Angular є вбудований механізм впровадження залежностей (Dependency Injection, DI). Він забезпечує слабкий зв'язок між компонентами та сервісами. Це спрощує заміну реалізацій модулів, наприклад, для тестування, без зміни основного коду [3]. Компонентна архітектура інкапсулює HTML, CSS і логіку в окремі блоки, що сприяє повторному використанню коду [5].

На відміну від Angular, фреймворки React і Vue орієнтовані на гнучкість і мінімалізм. React пропонує декларативний підхід на основі компонентів і односторонній потік даних. Він не нав'язує суворої структури застосунку чи механізму DI, залишаючи архітектурні рішення на розсуд розробників. Vue акцентує на простоті й поступовому впровадженні. Він дозволяє інтегрувати сучасні концепції навіть у невеликі проєкти. Однак ні React, ні Vue не мають вбудованої системи DI чи обов’язкової статичної типізації. Тому великі команди часто впроваджують додаткові інструменти для забезпечення структурованості коду. Angular завдяки інтегрованій архітектурі є оптимальним вибором для масштабованих корпоративних систем. Тут важливими залишаються стандартизація та суворе дотримання інженерних практик.

Реактивне керування у клієнтських архітектурах є ключовим аспектом при застосуванні компонентного підходу. Управління станом у розподілених клієнтських архітектурах належить до найскладніших інженерних завдань, оскільки неузгодженість стану між компонентами або між клієнтом і сервером може порушити цілісність даних [2]. В екосистемі Angular ця проблема вирішується за допомогою реактивного програмування із використанням бібліотеки RxJS [7]. Застосування патерна Observables дозволяє декларативно моделювати асинхронні процеси, формуючи єдиний потік даних із відповідей REST API, WebSocket-з'єднань і подій від користувача [8]. Такий підхід усуває необхідність імперативного відстеження змін, оскільки дані автоматично поширюються системою через підписки. Це забезпечує узгодженість і дає змогу реалізовувати складні сценарії обробки даних із мінімальними витратами

ресурсів пристрою [7]. Наприклад, у корпоративному поштовому клієнті на Angular із RxJS обробка вхідних листів організована так, що всі події об'єднуються в один потік, що дозволяє автоматично оновлювати список листів лише за виникнення змін. Система миттєво реагує на дії користувача без дублювання логіки в окремих компонентах. Використання RxJS підвищує узгодженість стану, зменшує ймовірність помилок при оновленні та полегшує супровід складних систем.

Обробка даних, ізоляція API та використання патерна «Фасад». Переходячи до аспектів інтеграції та взаємодії з даними, слід зазначити, що інтеграція із зовнішніми підсистемами та обробка складних форматів даних вимагають строгої інженерної ізоляції. Для цього впроваджуються сервіси-адаптери, які трансформують DTO (Data Transfer Objects) з сервера у внутрішні доменні моделі клієнтського застосунку. Для стандартизації взаємодії компонентів із сервісами даних та зовнішніми API застосовується патерн «Фасад» [9]. Фасад приховує складну логіку управління станом, наприклад, реалізацію через NgRx або Akita, і надає компонентам спрощений інтерфейс для взаємодії [9]. Внаслідок цього компоненти інтерфейсу залишаються абстрагованими від обробки даних. Навантаження з обробки даних перенесено на рівень сервісів.

Масштабування клієнтських систем за допомогою впровадження мікрофронтендів вирішує проблеми тісного зв'язування, характерні для монолітної клієнтської архітектури, подібно до бекенд-монолітів. Паралельна розробка ускладнюється, збільшується час компіляції та зростає ризик каскадних помилок [6, 10]. Індустрія відповіла на ці виклики впровадженням мікрофронтендів (Micro-frontends). Технології, такі як Webpack Module Federation, дозволяють розділити єдиний корпоративний інтерфейс на незалежні застосунки [6, 10]. Ці мікрозастосунки розробляються й розгортаються окремими командами, а об'єднуються у єдиний продукт під час виконання. Це відображає перенесення класичних принципів розподілених систем, таких як незалежне розгортання та ізоляція контекстів (Bounded contexts), на рівень

клієнта [1]. Однак впровадження мікрофронтендів супроводжується і недоліками: зростає складність інтеграції окремих модулів, виникають труднощі з уніфікацією стилів і бібліотек, а також необхідність синхронізувати версії залежностей. Збільшується навантаження на інфраструктуру, можливі проблеми із продуктивністю через зростання мережевих запитів або дублювання коду. Для успішного застосування підходу мікрофронтендів необхідно впроваджувати чіткі стандарти інтеграції та активно контролювати архітектурну цілісність.

Висновки: Проведений аналіз підтверджує, що сучасний front-end несе ті самі архітектурні ризики, що й серверні системи, і вимагає застосування надійних архітектурних рішень. Angular забезпечує базу для таких рішень завдяки DI, статичній типізації та компонентній моделі; RxJS перетворює асинхронні події на керовані потоки даних; патерни Фасад і Адаптер ізолюють компоненти від деталей реалізації, а мікрофронтенди переносять принцип незалежного розгортання на клієнтський рівень. Разом ці інструменти формують архітектуру, здатну зберігати підтримуваність і стійкість у довгостроковій перспективі.

Список використаних джерел:

1. Martin R. C. The Clean Architecture. 8th Light Blog, 2012. URL: https://blog.cleancoder.com/uncle-bob/2012/08/13/the-clean-architecture.html (дата звернення: 08.04.2026).
2. Kazman R. et al. A Holistic View of Architecture Definition, Evolution, and Analysis. Carnegie Mellon University, Software Engineering Institute, 2023. (CMU/SEI-2023-TR-004). URL: https://www.sei.cmu.edu/documents/5720/2023_005_001_983542.pdf (дата звернення: 08.04.2026).
3. Angular Documentation. Architecture Overview & Dependency Injection. Angular.dev. URL: https://angular.dev/guide/architecture (дата звернення: 08.04.2026).

4. Microsoft. The TypeScript Handbook. TypeScript Official Documentation. URL: https://www.typescriptlang.org/docs/handbook/intro.html (дата звернення: 08.04.2026).
5. Google. Angular Overview. Angular Official Documentation, 2023. URL: https://angular.dev/overview (дата звернення: 08.04.2026).
6. Microsoft. Cloud Design Patterns. Azure Architecture Center. URL: https://learn.microsoft.com/en-us/azure/architecture/patterns/ (дата звернення: 08.04.2026).
7. ReactiveX. RxJS: Reactive Extensions Library for JS. URL: https://rxjs.dev/guide/overview (дата звернення: 08.04.2026).
8. Angular University. RxJs for Beginners: A Beginner Friendly Introduction (Functional Reactive Programming for Angular Developers). URL: https://blog.angular-university.io/functional-reactive-programming-for-angular-2-developers-rxjs-and-observables/ (дата звернення: 08.04.2026).
9. Nrwl / Nx. Nx Architecture Concepts. URL: https://nx.dev/concepts/more-concepts/applications-and-libraries (дата звернення: 08.04.2026).
10. Jackson C. Micro Frontends. MartinFowler.com, 2019. URL: https://martinfowler.com/articles/micro-frontends.html (дата звернення: 08.04.2026).

УДК 004.415.2

ГІБРИДНА АРХІТЕКТУРА АДАПТИВНОГО ІНФЕРЕНСУ ДЛЯ АВТОМАТИЗОВАНОГО АНАЛІЗУ МЕДИЧНИХ ЗОБРАЖЕНЬ

Кудінов М. А.
kudinov@gmail.com
Черкаський державний фаховий бізнес-коледж
Марченко С. В.
м.Черкаси, Україна

Сучасні підходи до автоматизованого аналізу медичних зображень базуються на застосуванні глибоких нейронних мереж, які демонструють високі показники точності при виявленні патологічних змін у різних модальностях