

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ЗАПОРІЗЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ

Кваліфікаційна наукова праця
на правах рукопису

ФАЛЬКЕВИЧ ВІТАЛІЙ ГЕННАДІЙОВИЧ

УДК 004.75:004.738.5:004.272:004.415.2

ДИСЕРТАЦІЯ

**МЕТОДОЛОГІЯ ПРОЄКТУВАННЯ ВИСОКОНАВАНТАЖЕНИХ
ВЕБСИСТЕМ: АРХІТЕКТУРА, КЕШУВАННЯ ТА ОПТИМІЗАЦІЯ**

122 Комп'ютерні науки

12 Інформаційні технології

Подається на здобуття наукового ступеня доктора філософії з **комп'ютерних наук**

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

_____ В. Г. Фалькевич

Науковий керівник: Лісняк Андрій Олександрович,
кандидат фізико-математичних наук, доцент

Запоріжжя – 2026

АНОТАЦІЯ

Фалькевича В.Г. Методологія проєктування високонавантажених вебсистем: архітектура, кешування та оптимізація. – Кваліфікаційна наукова праця на правах рукопису. Дисертація на здобуття наукового ступеня доктора філософії за спеціальністю 122 «Комп’ютерні науки». – Запорізький національний університет, Запоріжжя, 2026.

У вступі обґрунтовано актуальність теми дисертаційної роботи, сформульовано мету, завдання, об’єкт, предмет дослідження, розкрито наукову новизну та практичну значущість отриманих результатів, визначено особистий внесок здобувача, апробацію результатів дослідження та їх висвітлення у наукових публікаціях. Висвітлено відомості щодо структури та обсягу дисертаційної роботи.

У першому розділі «Аналіз сучасних підходів до проєктування високонавантажених вебсистем» розглянуто та проаналізовано актуальні напрями розвитку архітектури вебдодатків, особливості їх масштабування й оптимізації в умовах мікросервісної архітектури та мультифронтендного середовища.

У підрозділі 1.1 охарактеризовано еволюцію архітектурних підходів – від монолітних рішень до мікрофронтендів – із фокусом на патерн BFF. Розглянуто альтернативні підходи – GraphQL і API Gateway – та наведено порівняльну характеристику їх застосовності в масштабованих системах.

У підрозділі 1.2 здійснено огляд викликів кешування в мікросервісній архітектурі, де важливим завданням є своєчасна інвалідація застарілих даних без порушення узгодженості між сервісами. Проаналізовано сучасні підходи до управління кешем, зокрема централізовані рішення, що базуються на подіях, TTL-стратегіях та тегованому кешуванні.

Підрозділ 1.3 присвячено інфраструктурним аспектам масштабування високонавантажених вебсистем. Розглянуто особливості CI/CD у мультифронтендній архітектурі, зокрема проблематику надмірної зв’язаності модулів, кешування результатів збірки, ефективного управління артефактами та

розподілу обчислювальних ресурсів. Проаналізовано практики оптимізації пайплайнів у середовищах із великою кількістю паралельних компонентів. Зазначено обмеження сучасних інструментів автоматизації та окреслено напрями подальших досліджень щодо підвищення ефективності процесів деплоюменту, зокрема з урахуванням специфіки мультикомандної розробки.

У другому розділі «Моделювання архітектури взаємодії клієнтських і серверних компонентів на основі патерну BFF у високонавантажених вебсистемах» розглянуто прикладні аспекти побудови архітектури взаємодії між клієнтським інтерфейсом і мікросервісами.

У підрозділі 2.1 проаналізовано ключові проблеми, пов'язані з необхідністю агрегування даних із кількох API-провайдерів, що характерно для мікросервісної архітектури.

У підрозділі 2.2 окреслено специфічні обмеження наявних інтеграційних рішень у контексті B2B-сегменту, де висока варіативність партнерських API, відсутність уніфікованих контрактів та потреба в гнучкій маршрутизації даних призводять до ускладнення побудови масштабованих вебсистем. Показано, що класичні підходи до інтеграції REST і GraphQL не забезпечують необхідної ізольованості доменних контекстів та не враховують потреби розділення зовнішніх (публічних) і внутрішніх (приватних) інтерфейсів.

У підрозділі 2.3 здійснено порівняльний аналіз поширених рішень – WunderGraph, Apollo Federation та GraphQL Mesh. Визначено низку обмежень, зокрема: жорстку структуру конфігурацій, складність підтримання окремих публічних і внутрішніх API-контрактів, залежність від централізованих схем, а також труднощі з генерацією типізованих SDK у сценаріях із великою кількістю мікросервісів і неоднорідних доменів.

У підрозділі 2.4 запропоновано BFF-підхід, що базується на фабриці провайдерів API, централізованому API-шлюзі та чіткому розділенні публічного та приватного API. Такий підхід забезпечує гнучку маршрутизацію запитів, контроль доступу на рівні шлюзу та можливість адаптації інтеграцій під різні сценарії без зміни клієнтської логіки. Обґрунтовано доцільність використання BFF як

проміжного рівня, який дозволяє стандартизувати взаємодію з гетерогенними сервісами у масштабованих вебсистемах.

У третьому розділі «Формалізація декларативного підходу до інвалідації кешу у мікросервісних системах» проаналізовано проблематику забезпечення консистентності даних у кеші високонавантажених архітектур.

У підрозділі 3.1 розглянуто особливості роботи сучасних кеш-систем, таких як Redis, Memcached та NCache. Окреслено ключові труднощі інвалідації кешу, пов'язані з асинхронною природою мікросервісів, що ускладнює своєчасне оновлення застарілих записів. Окремо розглянуто подієвий підхід на прикладі NCache як одного з поширених способів вирішення цієї проблеми.

У підрозділі 3.2 запропоновано архітектурні принципи побудови декларативної моделі інвалідації кешу, які передбачають чітке розмежування відповідальності між сервісами. Викладено концептуальну модель, яка дозволяє описувати правила інвалідації на рівні конфігурацій, без жорсткої прив'язки до коду. Показано, що така модель забезпечує зменшення міжсервісних залежностей, спрощує масштабування і покращує відгук системи за рахунок ефективного управління додаванням, оновленням та видаленням кешованих елементів.

У підрозділі 3.3 представлено концептуальну реалізацію запропонованого підходу, а також розглянуто алгоритмічні аспекти кешування в контексті динамічних навантажень.

У четвертому розділі «Побудова масштабованої інфраструктури та впровадження оптимізацій у вебсистемах з високим навантаженням» розглянуто практичні підходи до організації фронтенд-інфраструктури та удосконалення процесів розгортання вебсистем.

У підрозділі 4.1 проаналізовано архітектурні засади мультифронтендного середовища в межах монорепозиторіїв. Розкрито роль спільних і брендових компонентів у підвищенні повторного використання коду, а також проблематику управління залежностями та технічного боргу при масштабуванні кількох продуктів у єдиному репозиторії.

У підрозділі 4.2 висвітлено методи оптимізації CI/CD-процесів на основі кешування та паралельної збірки. Розглянуто техніки збереження залежностей, артефактів та Docker-образів між етапами, що дозволяє значно скоротити загальний час збірки. Наведено ефективність розподілу навантаження між GitLab CI Runners і впровадження стратегій паралельної обробки фронтенд-додатків у мультикомпонентних середовищах.

У підрозділі 4.3 представлено результати порівняння продуктивності пайплайнів за умов послідовного та паралельного виконання. Проведено аналіз потенціалу подальших оптимізацій, зокрема через гібридні стратегії кешування та балансування навантаження. Визначено обмеження наявних підходів та окреслено виклики, пов'язані з масштабуванням систем у складних розподілених середовищах.

У висновках наведено наукову новизну роботи, її практичну значимість та перспективи подальшого розвитку.

У дисертаційній роботі отримано такі наукові результати:

- *отримав подальший розвиток* архітектурний патерн BFF, що на відміну від існуючих архітектур поєднує фабрики API-провайдерів, централізований API-шлюз, підтримує одночасну взаємодію з REST та GraphQL, забезпечує агрегацію даних, масштабованість та керування доступом;
- *вперше* розроблено концепцію декларативної інвалідації кешу, що у порівнянні з існуючими рішеннями передбачає автономне керування кешем, зменшення залежностей між окремими сервісами та забезпечує консистентність даних у вебсистемах з мікросервісною архітектурою;
- *отримали подальшого розвитку* архітектурні підходи для побудови мультифронтендного середовища в межах монорепозіторіїв за рахунок структуризації спільних і брендових компонентів, контролю залежностей та ізоляції середовищ, що підвищує стабільність та керованість процесу збирання та розгортання систем з мікросервісною архітектурою;
- *отримали подальшого розвитку* методики оптимізації CI/CD-процесів для багатокомпонентних вебсистем на основі кешування артефактів, паралельного

виконання, умовного запуску завдань і використання легковагих контейнерних образів, що дозволяє скоротити час розгортання та зменшити навантаження на інфраструктуру.

Програмну реалізацію запропонованих підходів здійснено мовою програмування TypeScript із використанням таких технологій та інструментів: Node.js, GitLab CI, Next.js, NestJS, Redis і GraphQL.

Отримані результати дослідження використовуються на практиці під час проєктування та розробки високонавантажених вебсистем, зокрема для формування архітектури фронтенд-рішень, організації кешування та оптимізації CI/CD процесів у мультифронтендному середовищі. У роботі також враховано сучасні тенденції розвитку вебтехнологій, які формують загальний технічний контекст галузі та доповнюють розуміння сучасних підходів до розробки вебсистем.

Ключові слова: патерн BFF, мікросервіси, інвалідація кешу, високонавантажені вебсистеми, фронтенд-архітектура, декларативний підхід, оптимізація збірки, кешування артефактів, API-шлюз, інформаційні системи.

ABSTRACT

Falkevych V. Methodology for Designing High-Load Web Systems: Architecture, Caching, and Optimization. – PhD Thesis. Manuscript. Qualifying scientific.

Thesis submitted in partial fulfilment of the requirements for the degree of Doctor of Philosophy. Study program: 122 – Computer Science. Zaporizhzhia National University of the Ministry of Education and Science of Ukraine, Zaporizhzhia, 2026.

In the introduction, the relevance of the dissertation topic is substantiated; the aim, objectives, object, and subject of the research are formulated; the scientific novelty and practical significance of the obtained results are revealed; the personal contribution of the author is defined; and the approbation of the research results as well as their presentation in scientific publications are described. Information on the structure and scope of the dissertation is also provided.

In the first chapter, "Analysis of Modern Approaches to Designing High-Load Web Systems", current trends in the development of web application architectures are examined and analyzed, along with the specifics of their scaling and optimization under microservice architectures and multi-frontend environments.

Subsection 1.1 characterizes the evolution of architectural approaches – from monolithic solutions to microfrontend architectures with a focus on the BFF pattern. Alternative approaches, namely GraphQL and API Gateway, are also considered, and a comparative analysis of their applicability in scalable systems is provided.

Subsection 1.2 presents an overview of caching challenges in microservice architectures, where a key task is the timely invalidation of stale data without compromising consistency between services. Modern cache management approaches are analyzed, including centralized solutions based on event-driven mechanisms, TTL strategies, and tag-based caching.

Subsection 1.3 is devoted to the infrastructural aspects of scaling high-load web systems. It examines the specifics of CI/CD in multi-frontend architectures, including

issues of excessive module coupling, build result caching, efficient artifact management, and the distribution of computational resources. Practices for pipeline optimization in environments with a large number of parallel components are analyzed. The limitations of contemporary automation tools are identified, and directions for further research aimed at improving deployment process efficiency – particularly in the context of multi-team development – are outlined.

In the second chapter, "Modeling the Architecture of Interaction Between Client and Server Components Based on the BFF Pattern in High-Load Web Systems", applied aspects of designing the interaction architecture between the client interface and microservices are examined.

Subsection 2.1 analyzes key issues related to the need to aggregate data from multiple API providers, which is characteristic of microservice architectures.

Subsection 2.2 outlines specific limitations of existing integration solutions in the B2B context, where high variability of partner APIs, the lack of unified contracts, and the need for flexible data routing complicate the construction of scalable web systems. It is shown that classical approaches to integrating REST and GraphQL do not provide sufficient isolation of domain contexts and do not account for the necessity of separating external (public) and internal (private) interfaces.

Subsection 2.3 presents a comparative analysis of widely used solutions – WunderGraph, Apollo Federation, and GraphQL Mesh. A number of limitations are identified, including rigid configuration structures, difficulties in maintaining separate public and internal API contracts, dependence on centralized schemas, and challenges in generating strongly typed SDKs in scenarios involving a large number of microservices and heterogeneous domains.

Subsection 2.4 proposes a BFF-based approach that relies on an API provider factory, a centralized API gateway, and a clear separation between public and private APIs. This approach enables flexible request routing, access control at the gateway level, and the ability to adapt integrations to different use cases without modifying client-side logic. The feasibility of using BFF as an intermediary layer that standardizes interaction with heterogeneous services in scalable web systems is substantiated.

In the third chapter, "Formalization of a Declarative Approach to Cache Invalidation in Microservice Systems", the issues of ensuring data consistency in the cache of high-load architectures are analyzed.

Subsection 3.1 examines the operational characteristics of modern caching systems such as Redis, Memcached, and NCache. Key challenges of cache invalidation related to the asynchronous nature of microservices are outlined, as this significantly complicates the timely update of stale records. Particular attention is given to the event-driven approach, illustrated by NCache as one of the commonly used solutions to this problem.

Subsection 3.2 proposes architectural principles for constructing a declarative cache invalidation model, which assumes a clear separation of responsibilities among services. A conceptual model is presented that enables the specification of invalidation rules at the configuration level without a rigid coupling to application code. It is shown that this model reduces inter-service dependencies, simplifies system scaling, and improves system responsiveness through effective management of cache entry creation, update, and eviction.

Subsection 3.3 presents a conceptual implementation of the proposed approach and examines the algorithmic aspects of caching in the context of dynamic workloads.

In the fourth chapter, "Designing a Scalable Infrastructure and Implementing Optimizations in High-Load Web Systems", practical approaches to organizing frontend infrastructure and improving web system deployment processes are examined.

Subsection 4.1 analyzes the architectural foundations of multi-frontend environments within monorepositories. The role of shared and brand-specific components in enhancing code reuse is discussed, along with the challenges of dependency management and technical debt when scaling multiple products within a single repository.

Subsection 4.2 presents methods for optimizing CI/CD processes based on caching and parallel builds. Techniques for preserving dependencies, artifacts, and Docker images across pipeline stages are examined, enabling a significant reduction in overall build time. The effectiveness of load distribution among GitLab CI Runners and the implementation of parallel processing strategies for frontend applications in multi-component environments are demonstrated.

Subsection 4.3 presents the results of a comparative performance analysis of pipelines under sequential and parallel execution conditions. The potential for further optimization is analyzed, particularly through hybrid caching strategies and load balancing. The limitations of existing approaches are identified, and challenges associated with scaling systems in complex distributed environments are outlined.

The conclusions summarize the scientific novelty of the work, its practical significance, and directions for future research and development.

The dissertation presents the following scientific results:

- the BFF architectural pattern has been *further developed*, which, unlike existing architectures, integrates API provider factories with a centralized API gateway, supports simultaneous interaction with REST and GraphQL, and ensures data aggregation, scalability, and access control;
- *for the first time*, a concept of declarative cache invalidation has been developed, which, compared to existing solutions, enables autonomous cache management, reduces inter-service dependencies, and ensures data consistency in web systems based on microservice architecture;
- architectural approaches to building multi-frontend environments within monorepositories have been *further developed* through the structuring of shared and brand-specific components, dependency control, and environment isolation, which increases the stability and manageability of the build and deployment processes in microservice-based systems;
- methodologies for optimizing CI/CD processes for multi-component web systems have been further developed based on artifact caching, parallel execution, conditional job triggering, and the use of lightweight container images, making it possible to reduce deployment time and decrease infrastructure load.

The software implementation of the proposed approaches was carried out using the TypeScript programming language and the following technologies and tools: Node.js, GitLab CI, Next.js, NestJS, Redis, and GraphQL.

The obtained research results are applied in practice in the design and development of high-load web systems, for shaping frontend architectures, organizing caching

mechanisms, and optimizing CI/CD processes in multi-frontend environments. The dissertation also considers current trends in the development of web technologies, which form the broader technological context of the field and complement the understanding of modern approaches to web system development.

Keywords: BFF pattern, microservices, cache invalidation, high-load web systems, frontend architecture, declarative approach, build optimization, artifact caching, API gateway, information systems.

СПИСОК ПУБЛІКАЦІЙ ЗДОБУВАЧА

Статті у наукових фахових виданнях України:

1. Falkevych V.G., Lisniak A.O. Client state management using backend for frontend pattern architecture in B2B segment. Artificial Intelligence. 2024. № 2. С. 49–60.
2. Falkevych V.G., Lisniak A.O. Optimization of infrastructure deployment for multi-frontends in monorepo. Artificial Intelligence. 2025. № 2. С. 63–70.

Статті у виданнях включених до наукометричної бази Scopus:

3. Falkevych V.G., Lisniak A.O. Cache invalidation based on a declarative approach for separating business logic of microservices from cache update rules. Eastern-European Journal of Enterprise Technologies. 2025. № 2. С. 68–74.

Наукові праці, які засвідчують апробацію матеріалів дисертації:

4. Фалькевич В.Г., Лісняк А.О. Архітектурні рішення для створення сучасних веб-додатків. Інформаційні системи та технології : матеріали 11-ї Міжнародної науково-технічної конференції ICT-2022.
5. Falkevych V.G., Lisniak A.O. Methodology of cache invalidation in microservices architecture of the web applications. Вчені записки ТНУ імені В. І. Вернадського. Серія: Технічні науки. 2023. С. 131–135.
6. Falkevych V.G., Lisniak A.O. Internal and External Threats in Cyber Security and Methods for Their Prevention. Advanced Computer Information Technologies : матеріали 13-ї Міжнародної конференції (ACIT 2023). 2023. С. 414–419.

ЗМІСТ

Перелік умовних позначень	16
Вступ.....	17
1 Аналіз сучасних підходів до проєктування високонавантажених вебсистем	22
1.1 Аналіз архітектурних паттернів для побудови високонавантажених вебсистем.....	22
1.2 Аналіз методів кешування у мікросервісних архітектурах	29
1.3 Інфраструктурні аспекти масштабування та оптимізації вебсистем	33
1.3.1 Мультифронтендна архітектури в умовах масштабування	34
1.3.2 Автоматизація CI/CD для багатокomпонентних вебсистем.....	35
1.3.3 Оптимізація пайплайнів	36
1.3.4 Обмеження сучасних підходів.....	37
Висновки до розділу 1	38
2. Моделювання архітектури взаємодії клієнтських і серверних компонентів на основі патерну BFF у високонавантажених вебсистемах	40
2.1 Необхідність об'єднання даних з кількох мікросервісів для забезпечення цілісного стану клієнта	40
2.2 Обмеження наявних рішень у контексті B2B-сегменту.....	42
2.2.1 Аналіз та обмеження WunderGraph.....	44
2.2.2 Аналіз та обмеження Apollo Federation	47
2.2.3 Аналіз та обмеження GraphQL Mesh	49
2.3 Фабрика провайдерів API, шлюз, розділення API на публічне та приватне	51
2.3.1 Особливості розділення публічного та приватного API.....	53
2.3.2 Обмеження у використанні конфігураційних файлів та генерації клієнтського коду	56

	14
2.3.3 Фабрика провайдерів API як рівень абстракції взаємодії з мікросервісами	58
2.3.4 API-шлюз як єдина точка маршрутизації та контролю доступу	62
Висновки до розділу 2	64
3 Формалізація декларативного підходу до інвалідації кешу у мікросервісних системах.....	66
3.1 Проблематика інвалідації кешу у високонавантажених мікросервісних архітектурах	66
3.1.1 Огляд популярних кеш-систем	67
3.1.2 Проблематика інвалідації кешу	72
3.1.3 Огляд реалізації подієвої інвалідації кешу на прикладі Ncache	74
3.1.4 Декларативний підхід як відповідь на сучасні виклики	75
3.2 Архітектурні принципи побудови декларативної моделі інвалідації кешу ..	77
3.2.1 Поділ відповідальності між мікросервісами для автономного управління кешем та мінімізації міжсервісних залежностей	78
3.2.2 Концептуальна модель декларативного підходу інвалідації кешу	80
3.2.3 Оптимізація механізмів додавання, оновлення та видалення елементів кешу для зменшення затримок і прискорення отримання даних.....	85
3.3 Оцінка ефективності та порівняльний аналіз.....	88
Висновки до розділу 3	90
4 Побудова масштабованої інфраструктури та впровадження оптимізацій у вебсистемах з високим навантаженням	92
4.1 Архітектура мульти-фронтендного середовища у монорепозіторіях	92
4.1.1 Визначення концепції мульти-фронтендного проєкту: спільні та брендові компоненти	93

4.1.2 Роль монорепо-архітектури у підтримці масштабованості та модульності	94
4.1.3 Проблеми повторного використання бібліотек і управління залежностями.....	96
4.2 Оптимізація CI/CD пайплайнів за допомогою кешування та паралелізації .	97
4.2.1 Реалізація кешування між етапами збірки: залежності, артефакти, Docker-образи	98
4.2.2 Впровадження паралельної збірки пакетів і фронтенд-додатків	99
4.2.3 Оптимізація використання ресурсів: артефакти та GitLab CI Runners.	101
4.3 Аналіз продуктивності та напрями оптимізації CI/CD	106
4.3.1 Порівняння продуктивності: послідовне та паралельне виконання (експериментальні результати).....	107
4.3.2 Перспективи подальшої оптимізації: баланс кешу, гібридні стратегії розгортання	109
4.3.3 Обмеження оптимізацій та виклики масштабування	111
Висновки до розділу 4	112
Висновки	114
Список використаних джерел	116
Додатки.....	127

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ

AOF – Append-Only File (файл лише для дописування)
AJAX – Asynchronous JavaScript and XML
API – Application Programming Interface
B2B – Business to Business
BFF – Backend for frontend
CI/CD – Continuous Integration / Continuous Delivery (Deployment)
CRM – Customer Relationship Management
CSS – Cascading Style Sheets
FPGA – Field-Programmable Gate Array
HTTP – HyperText Transfer Protocol
PAAS – Platform as a Service
SUB – Subscriber (підписник або одержувач повідомлень)
REST – Representational State Transfer
SDK – Software Development Kit
SPA – Single Page Application
TTL – Time to Live (час життя даних у кеші або DNS перед видаленням)
URL – Uniform Resource Locator
YAML – YAML Ain't Markup Language

ВСТУП

Актуальність теми. Вебсистеми прикладного призначення відіграють важливу роль у функціонуванні інформаційних систем різного масштабу та призначення. До цього класу програмного забезпечення належать платформи електронної комерції, фінансові сервіси, хмарні SaaS-рішення, інформаційні портали та CRM-системи, які характеризуються високою інтенсивністю запитів, значною кількістю одночасних користувачів і підвищеними вимогами до надійності.

Розширення сфер застосування та зростання функціональних можливостей вебсистем, а також постійне збільшення кількості користувачів зумовлюють підвищені вимоги до доступності, масштабованості, продуктивності, безпеки та відмовостійкості. У таких умовах проектування високонавантажених вебсистем потребує застосування сучасних архітектурних рішень, ефективних підходів до кешування та оптимізації процесів безперервної інтеграції й розгортання програмного забезпечення.

Водночас значна частина прикладних вебсистем реалізується у форматі мультифронтендних середовищ, що передбачають одночасне функціонування кількох клієнтських застосунків, орієнтованих на різні бренди, регіони або функціональні сегменти. Такі системи можуть включати кілька інтернет-магазинів або цифрових сервісів зі спільною функціональною основою, бізнес-логікою та інфраструктурними компонентами, проте з відмінностями у конфігурації, стилізації або параметрах локалізації. Застосування мультифронтендного підходу сприяє повторному використанню програмних модулів і зменшенню витрат на розробку, однак призводить до зростання архітектурної складності, ускладнення керування залежностями, організації кешування та процесів розгортання в багатокомпонентних архітектурах [3,5].

Таким чином, існуючі підходи до розробки та експлуатації вебсистем здебільшого орієнтовані на вирішення окремих технічних задач і не забезпечують

цілісного методологічного підходу до узгодження архітектури, механізмів кешування та інфраструктурної оптимізації з урахуванням мультифронтендних середовищ. Це обумовлює актуальність наукового дослідження, спрямованого на розробку методології проєктування високонавантажених вебсистем.

Вагомий внесок у розвиток підходів до проєктування сучасних вебсистем, зокрема в контексті мікросервісних архітектур, мікрофронтендів та організації взаємодії клієнтських і серверних компонентів у розподілених середовищах, зроблено у працях N. Savani, Y. Abgaz, A. McCarren, P. Elger, A. Solan, S. Singh та інших науковців. У зазначених дослідженнях розглядаються архітектурні принципи масштабованості, декомпозиції функціональності та підвищення продуктивності вебсистем в умовах високих навантажень [1, 2, 3].

Крім того, проблематикою проєктування вебсистем прикладного призначення також займалися вітчизняні дослідники О.А. Дмитренко, І.С. Бурлаченко, Д.О. Заворотній, М.Р. Мельник, С.О. Субботін, С.І. Гоменюк, Ю.О. Лимаренко у роботах яких досліджуються загальні питання архітектурної організації програмних систем, моделювання обчислювальних процесів і підходи до побудови ефективних інформаційних систем [105, 106, 107, 108, 109].

Мета і задачі дослідження. Метою дисертаційного дослідження є розробка методології проєктування високонавантажених вебсистем з урахуванням архітектурних патернів, підходів до кешування та інфраструктурної оптимізації.

Для досягнення поставленої мети в дисертації сформульовано наступні задачі:

- дослідити застосування архітектурних патернів проєктування у високонавантажених вебдодатках для вирішення проблем агрегації даних та реалізації механізмів контролю доступу;
- розвинути архітектурний патерн BFF для підтримки одночасної взаємодії з REST і GraphQL, агрегації даних з різноманітних джерел, масштабованості та розділення прав доступу;
- виконати аналіз методів та підходів до кешування даних у вебсистемах з мікросервісною архітектурою;

- розробити концепцію декларативного керування кешем у вебсистемах з мікросервісною архітектурою, що забезпечує автономне керування кешем на основі формалізованих правил з метою підтримки консистентності даних та зменшення зв'язаності у системах з мікросервісною архітектурою;
- розробити архітектурні підходи до побудови мультифронтендного середовища в межах монорепозіторіїв, за рахунок структуризації спільних і брендових компонентів, контролю залежностей та ізоляції середовищ;
- розробити методики оптимізації CI/CD-процесів у багатокомпонентних вебсистемах, для скорочення часу розгортання та зменшення навантаження на інфраструктуру.

Об'єкт дослідження – процес проектування розподілених, масштабованих, мультифронтендних вебсистем з мікросервісною архітектурою.

Предмет дослідження – моделі, методи та архітектурні підходи до проектування високонавантажених вебсистем, які охоплюють способи організації взаємодії фронтенд- і бекенд-компонентів, підходи до кешування даних, методи оптимізації обчислювальних та інфраструктурних процесів, зокрема CI/CD, у розподілених мультифронтендних вебсистемах з мікросервісною архітектурою.

Наукова новизна. Наукова новизна дисертаційного дослідження полягає в одержанні наступних наукових результатів:

- *отримав подальший розвиток* архітектурний патерн BFF, що на відміну від існуючих архітектур поєднує фабрики API-провайдерів, централізований API-шлюз, підтримує одночасну взаємодію з REST та GraphQL, забезпечує агрегацію даних, масштабованість та керування доступом;
- *вперше* розроблено концепцію декларативної інвалідації кешу, що у порівнянні з існуючими рішеннями передбачає автономне керування кешем, зменшення залежностей між окремими сервісами та забезпечує консистентність даних у вебсистемах з мікросервісною архітектурою;
- *отримали подальшого розвитку* архітектурні підходи для побудови мультифронтендного середовища в межах монорепозіторіїв за рахунок структуризації спільних і брендових компонентів, контролю залежностей та

ізоляції середовищ, що підвищує стабільність та керованість процесу збирання та розгортання систем з мікросервісною архітектурою;

- *отримали подальшого розвитку* методики оптимізації CI/CD-процесів для багатокомпонентних вебсистем на основі кешування артефактів, паралельного виконання, умовного запуску завдань і використання легковагих контейнерних образів, що дозволяє скоротити час розгортання та зменшити навантаження на інфраструктуру.

Практичне значення одержаних результатів. Практичне значення роботи полягає в можливості використання запропонованих методів та архітектурних підходів для побудови надійних, масштабованих, ефективних та безпечних вебсистем які, відповідають сучасним тенденціям розвитку галузі. Крім того, запропоновані підходи можуть бути застосовані при:

- реалізації вебсистем із великими навантаженнями, де критично важливо забезпечити швидкодію та цілісність стану клієнта;
- організації процесів розгортання у великих командах розробників із мультибрендовими або мультифронтендними проєктами;
- впровадженні систем кешування із чітко визначеною логікою інвалідації в рамках мікросервісної архітектури;
- оптимізації використання обчислювальних ресурсів CI/CD-інфраструктури в умовах обмежених потужностей або високої вартості виконання.

Апробація результатів дослідження. Результати дисертаційного дослідження було оприлюднено у наступних працях, семінарах, конференціях:

1. Фалькевич В.Г., Лісняк А.О. Архітектурні рішення для створення сучасних веб-додатків. Інформаційні системи та технології : матеріали 11-ї Міжнародної науково-технічної конференції ICT-2022.
2. Falkevych V.G., Lisniak A.O. Methodology of cache invalidation in microservices architecture of the web applications. Вчені записки ТНУ імені В. І. Вернадського. Серія: Технічні науки. 2023. С. 131–135.

3. Falkevych V.G., Lisniak A.O. Internal and External Threats in Cyber Security and Methods for Their Prevention. Advanced Computer Information Technologies : матеріали 13-ї Міжнародної конференції (ACIT 2023). 2023. С. 414–419.

Публікації. Основні положення дисертації викладено у 6 наукових публікаціях, серед яких: 3 статті опубліковані у наукових виданнях, які включені до Переліку наукових фахових видань України та визнаються як фахові з комп'ютерних наук, серед яких 1 стаття у періодичному науковому журналі, що індексується в наукометричній базі Scopus; 1 тези доповідей у міжнародному збірнику матеріалів науково-технічної конференції; 1 наукова стаття у збірнику матеріалів конференції, що входить до наукометричної бази Scopus; 1 стаття у науковому фаховому виданні України з програмної інженерії .

Структура та обсяг роботи. Дисертація складається зі вступу, чотирьох розділів, які містять 12 підрозділів, висновків, списку використаних джерел та додатків. Загальний обсяг дисертації становить 155 сторінок, у т. ч. основного тексту – 99 сторінок. Список використаних джерел налічує 109 найменування (11 сторінок). Додатки – 29 сторінок.

1 АНАЛІЗ СУЧАСНИХ ПІДХОДІВ ДО ПРОЄКТУВАННЯ ВИСОКОНАВАНТАЖЕНИХ ВЕБСИСТЕМ

1.1 Аналіз архітектурних паттернів для побудови високонавантажених вебсистем

Зростання цифрової взаємодії у бізнесі, освіті та повсякденному житті зумовило стрімке поширення вебдодатків як основного інструменту доступу до інформаційних сервісів. Їхня універсальність, кросплатформність і зручність використання створюють попит на все складніші рішення, здатні забезпечувати миттєву реакцію та безперебійну роботу під час високих навантажень. Водночас із розширенням функціональності та кількості користувачів зростають вимоги до високої доступності, безпеки, масштабованості, продуктивності та відмовостійкості таких систем [104].

Сучасні вебдодатки є розподіленими інформаційними системами і реалізуються відповідно до клієнт-серверної архітектури (рис. 1.1).

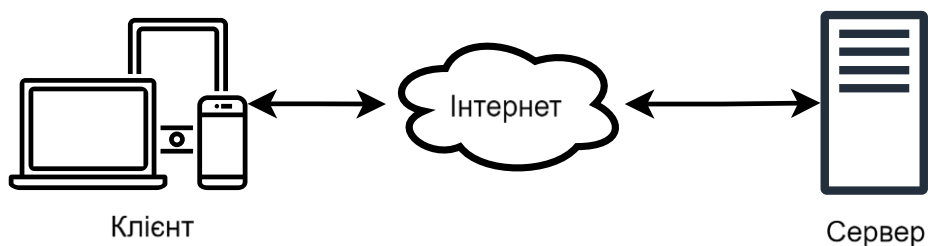


Рисунок 1.1 – Клієнт-серверна архітектура

Як і в інших сферах розробки програмного забезпечення на початковому етапі вебдодатки створювались на базі монолітної архітектури, де бізнес-логіка, прикладний програмний інтерфейс (API) та інтерфейс користувача (UI) були об'єднані в одну кодову базу (рис 1.2). Для такого підходу характерна низька масштабованість та складність у підтримці, особливо в умовах зростання обсягу функціональності та кількості користувачів та розробників проєкту. Із розвитком

технологій асинхронних запитів, JavaScript та XML (AJAX), каскадних таблиць стилів (CSS) та JavaScript у 2000-х роках сформувалися динамічні методи побудови користувацьких інтерфейсів, що згодом спричинило розмежування клієнтської та серверної частин вебдодатків за допомогою REST API та інших стандартних протоколів взаємодії [1].

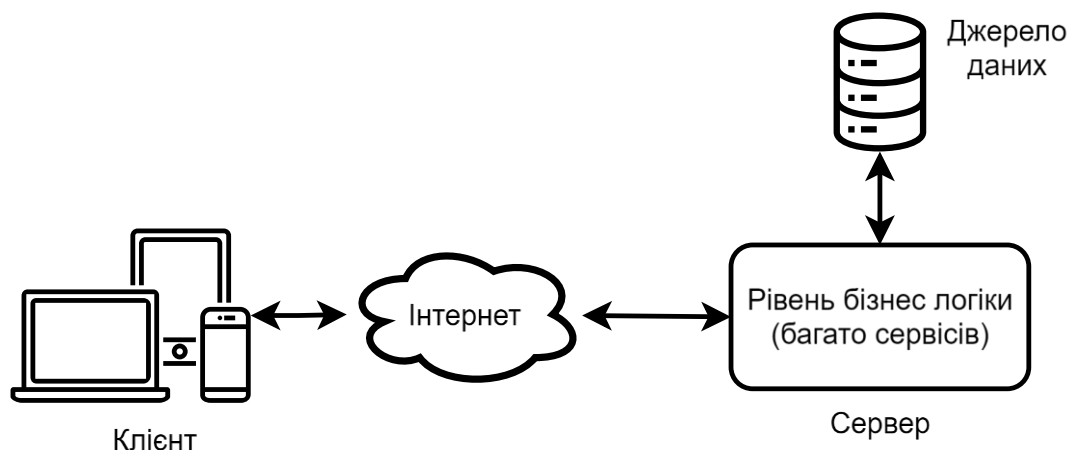


Рисунок 1.2 – Монолітна архітектура

Подальший розвиток привів до мікросервісної архітектури, яка принесла гнучкість у розробці бекенду, дозволивши розділяти бізнес-логіку на незалежні сервіси (рис. 1.3). Данна архітектура дає змогу незалежно масштабування окремих сервісів, забезпечую гнучкі механізми оновлення системи та забезпечує простоту розробки та підтримки різним командами розробників. Водночас, ускладнює управління архітектурою та вимагає правильного розподілу обов'язків між сервісами на етапі проектування системи [3, 6, 11].

Не зважаючи на розподіл обов'язків у серверній частині додатку, клієнтська частина у більшості випадків залишається монолітною (рис 1.4) . Цей дисбаланс став поштовхом до формування концепції мікрофронтендів – підходу, що дозволяє розбивати інтерфейс користувача на окремі автономні модулі, які можуть незалежно розроблятися і розгортатися різними командами [3, 12].

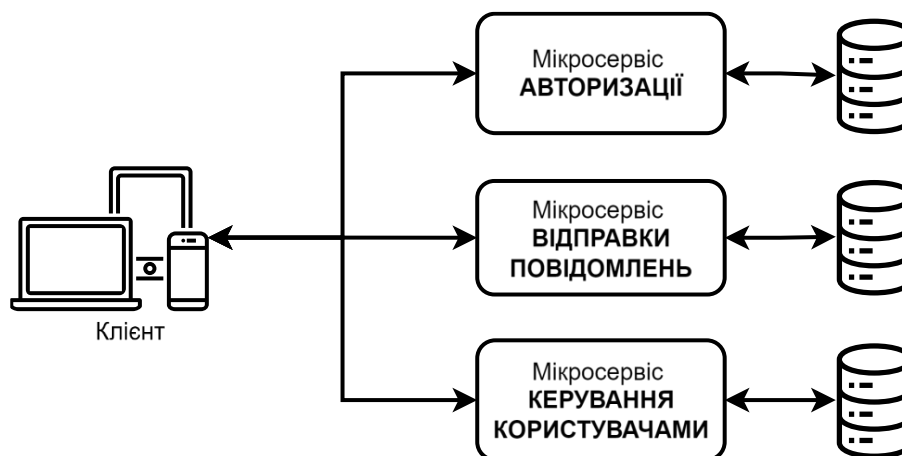


Рисунок 1.3 – Мікросервісна архітектура

Дослідження показують, що мікрофронтенди підвищують масштабованість і підтримуваність за рахунок децентралізації коду, зменшення взаємозалежностей і спрощення оновлення окремих частин користувацького інтерфейсу. Наприклад, у дослідженні [3] зазначено, що використання мікрофронтендів сприяє скороченню загального часу безперервної інтеграції та розгортання (CI/CD), що, у свою чергу, дозволяє швидше впроваджувати оновлення у багатокомандних проєктах. Водночас, це супроводжується зростанням сумарної цикломатичної складності в порівнянні з традиційними монолітними вебдодатками, що ускладнює підтримку при малій кількості розробників або в централізованих командах [4].

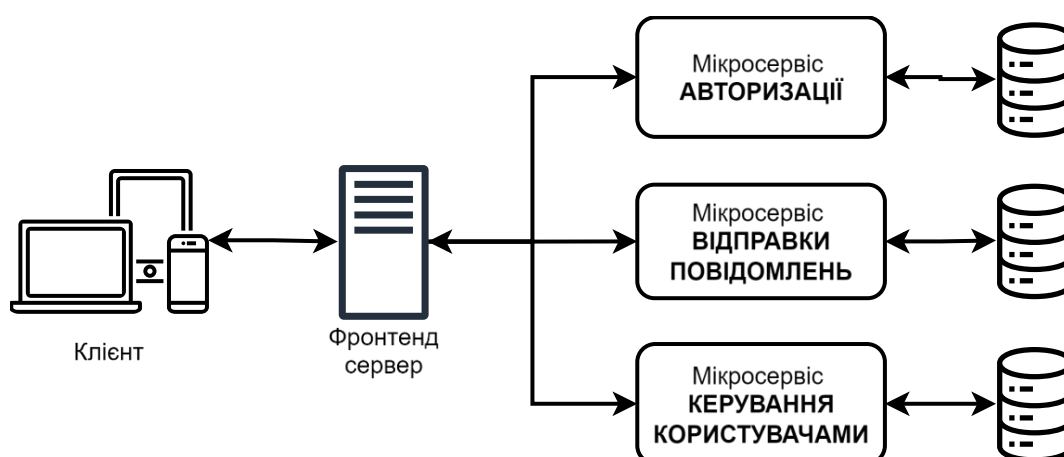


Рисунок 1.4 – Монолітна архітектура клієнтського кода

Сучасні наукові джерела підкреслюють важливість коректного вибору методології реалізації мікрофронтендів – від підходів, що базуються на

використанні односторінкових вебдодатків (SPA) і дозволяють поєднувати різні фреймворки в межах однієї сторінки, до моделей, які передбачають розгортання кількох незалежних вебдодатків за окремими URL-адресами з використанням механізму спільного використання модулів, наприклад у Webpack 5 [5].

Поряд із перевагами, впровадження мікрофронтедів супроводжується новими викликами: забезпеченням консистентності користувацького інтерфейсу, інтеграцією з API, спільним управлінням станом, а також потребою у стандартизованих протоколах для міжкомпонентної взаємодії. Незважаючи на це, переважає консенсус, що мікрофронтеди є стратегічно вигідним вибором для великих, динамічних організацій та проєктів з високими вимогами до гнучкості та масштабування [6].

У сучасних веб та мобільних системах дедалі частіше виникає необхідність одночасної підтримки різних типів клієнтів – вебдодатків, мобільних застосунків, Smart TV, а також вбудованих інтерфейсів (рис 1.5).

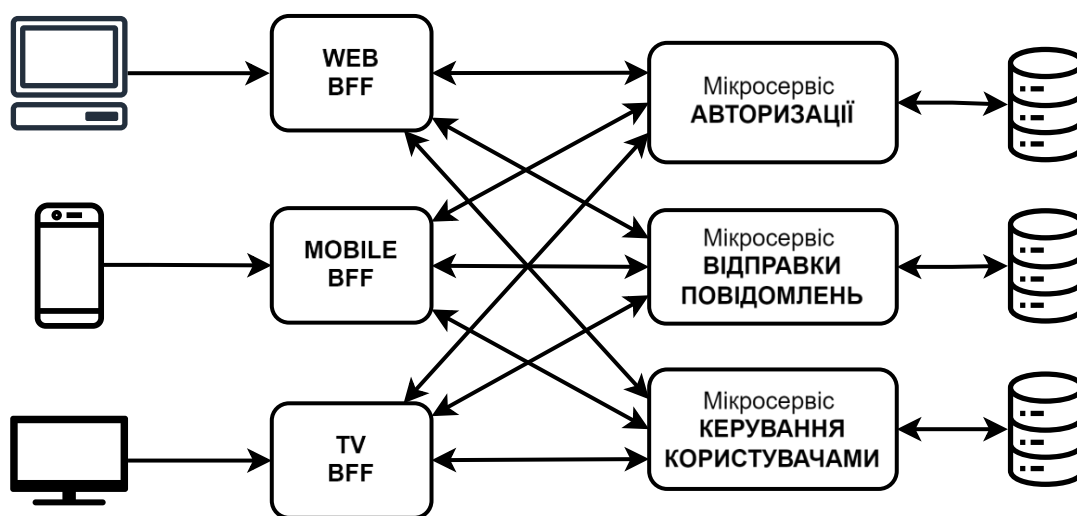


Рисунок 1.5 – Архітектура BFF

Як зазначено в роботі [7], різні компоненти інтерфейсу користувача часто взаємодіють з кількома мікросервісами одночасно, що ускладнює використання одного загального API. Внаслідок цього REST API, побудований за уніфікованими правилами, не завжди здатен забезпечити належну гнучкість для таких сценаріїв.

Це створює передумови для запровадження адаптаційного шару у вигляді патерну Backend for Frontend (BFF), який дозволяє централізовано керувати агрегацією та трансформацією даних відповідно до вимог кожної платформи [1, 8].

Патерн BFF пропонує ефективне рішення цієї проблеми. Він передбачає створення окремого бекенд-шару для кожного типу клієнта або навіть окремих інтерфейсних-компонентів. Кожен BFF виконує адаптацію, агрегацію та трансформацію даних, забезпечує обробку помилок, кешування, авторизацію і цим зменшує навантаження на клієнтський код. Це дозволяє уникнути дублювання бізнес-логіки у фронтенді, спрощує розробку і підвищує гнучкість системи [9].

Особливу роль BFF відіграє у складних користувацьких-сценаріях, де необхідно агрегувати дані з кількох мікросервісів. Наприклад, таблиці, фільтрація, пошук чи динамічне відображення контенту вимагають паралельної взаємодії з кількома джерелами. BFF дозволяє фронтенду звертатися лише до одного сервісу, який всередині оптимізовано взаємодіє з усіма необхідними мікросервісами. Це знижує латентність і зменшує обсяг переданих даних, що підтверджено у випадку мобільного клієнта системи, де BFF зменшив кількість запитів і трафіку без шкоди для функціональності [7].

У масштабованих командах, де розробка розділена між фронтенд та бекенд-підрозділами, BFF дозволяє зменшити міжкомандну залежність. Кожна команда може розробляти свій BFF, адаптований до її інтерфейсу, що дозволяє швидко впроваджувати зміни, мінімізуючи ризики інтеграційних конфліктів. Такий підхід підтримує гнучку організаційну структуру, що особливо актуально у великих компаніях або проєктах з частими змінами функціоналу [7].

Крім того, BFF підвищує безпеку та масштабованість у мікросервісних архітектурах. Він виступає посередником між клієнтом і внутрішніми сервісами, інкапсулюючи політики доступу, формати даних і специфічну логіку. Це відповідає принципу єдиної відповідальності та дозволяє внутрішнім сервісам залишатися вузькоспеціалізованими [6, 15].

Результати дослідження Samer Alkhodary (2022) демонструють, що впровадження BFF сприяє зниженню латентності, обсягу переданих даних і

складності клієнтського коду, особливо в умовах мікросервісного середовища. Хоча у дослідженні не вказується фіксоване значення приросту продуктивності у відсотках, експериментально підтверджено, що перехід до BFF знижує кількість HTTP-запитів, покращує підтримуваність і спрощує архітектуру обміну даними між клієнтом і сервером [9, 13, 14, 16].

Патерн BFF розглядається як ефективне рішення для адаптації даних до конкретних потреб клієнтського інтерфейсу. Проте існують альтернативні підходи, які також вирішують задачі агрегації, маршрутизації й трансформації даних. Серед найбільш поширених варто відзначити GraphQL, API Gateway і такі інструменти, як WunderGraph [10].

GraphQL є специфікацією запитів, що дозволяє клієнту отримувати саме ті поля, які йому потрібні, без надмірних даних. Ця особливість зменшує обсяг переданих даних. Проте запровадження GraphQL потребує перегляду бекенд-логіки, а також ретельного налаштування кешування, контролю доступу й механізмів безпеки. Особливо складним є забезпечення консистентності при роботі з динамічним контентом. У дослідженні [10] відзначено, що GraphQL API складно піддаються повноцінному кешуванню, а підтримка інвалідації залишається слабкою.

Іншим поширеним підходом є використання API Gateway – інфраструктурного рішення, яке виступає єдиною точкою входу до мікросервісної системи (рис 1.6). Згідно з дослідженням [11], API Gateway виконує низку функцій, зокрема маршрутизацію запитів, балансування навантаження, аутентифікацію, логування та обмеження трафіку. Він ефективний у сценаріях централізованого управління мережею, однак не виконує адаптацію даних під клієнтський інтерфейс, що часто ускладнює використання у фронтенд-орієнтованих системах із сильною залежністю від користувацького інтерфейсу.

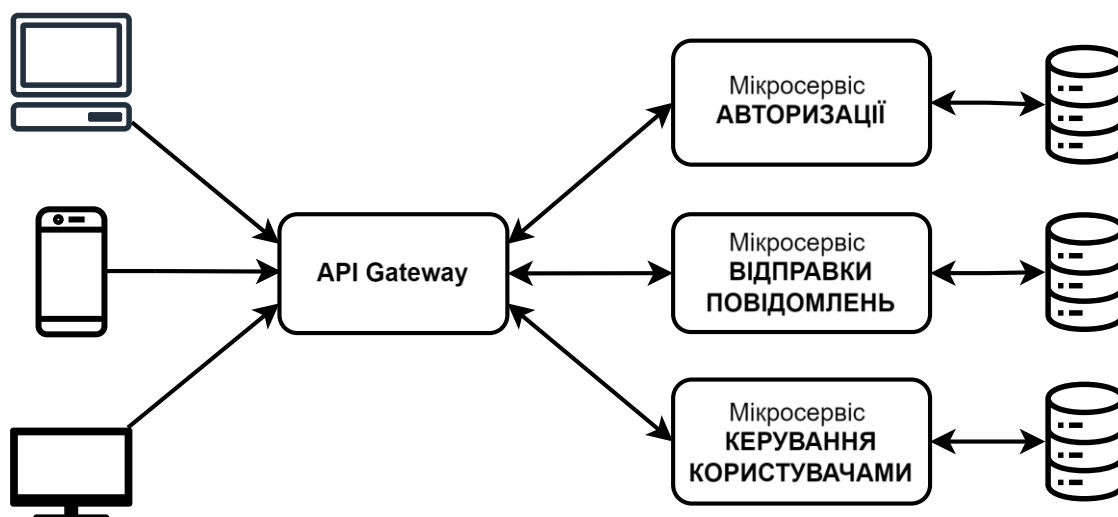


Рисунок 1.6 – Архітектура API Gateway

Окремий інтерес становить підхід, реалізований у WunderGraph – інструменті, що поєднує риси як API Gateway, так і BFF. Він дозволяє створювати віртуальні графи з кількох джерел (REST, GraphQL тощо), формуючи єдину типізовану точку входу.

У дослідженні [7] зазначається, що WunderGraph є ефективним рішенням для простих застосунків, однак у сценаріях, орієнтованих на бізнес із декількома фронтендами, складною авторизацією та потребою в ізоляції API (наприклад, поділом на публічні та приватні) виникають обмеження масштабованості. Зокрема, кожна зміна у конфігурації API потребує оновлення в усіх фронтенд-інстансах, що суттєво ускладнює підтримку системи.

Таким чином, підхід BFF вирізняється максимальною адаптивністю до потреб клієнтського інтерфейсу та високою гнучкістю масштабування за рахунок відокремлення логіки обробки запитів для кожного конкретного фронтенду. У порівнянні з GraphQL і API Gateway, BFF дає змогу делегувати більшу частину специфічної логіки на сторону frontend-команди, зберігаючи при цьому інтеграцію з мікросервісною інфраструктурою через контрольований інтерфейс. У цьому контексті вибір між зазначеними підходами має базуватися на специфіці домену, командної структури та вимог до масштабованості й безпеки.

1.2 Аналіз методів кешування у мікросервісних архітектурах

Мікросервісна архітектура широко використовується при побудові сучасних високо навантажених вебсистем завдяки своїй здатності забезпечувати масштабованість, гнучкість та незалежність розгортання окремих компонентів [17]. Однак зі зростанням обсягів оброблюваних даних, збільшенням кількості одночасних користувачів та необхідністю в оперативному доступі до актуальної інформації перед такими системами постають нові виклики, пов'язані з продуктивністю. Одним із ключових засобів для вирішення цих проблем є кешування, яке дозволяє зменшити навантаження на сервери, знизити частоту звернень до баз даних і прискорити обробку запитів користувачів.

Типову схему взаємодії між мікросервісом, базою даних і розподіленим кешем наведено на рис. 1.7, де показано процес оновлення даних, інвалідації кешу та реакцію системи на наявність та відсутність кешу. Помилки в інвалідації можуть призводити до неконсистентності даних, надмірної інфраструктурної складності або втрати переваг кешування [20, 21].



Рисунок 1.7 – Взаємодія мікросервісу з розподіленим кешем

Попри очевидні переваги, кешування в мікросервісних системах супроводжується низкою складних проблем. Найбільш критичними є: забезпечення актуальності кешованих даних, синхронізація кешу з фактичними змінами у внутрішній структурі сервісів, а також ризик використання застарілої інформації. Відсутність уніфікованих та стандартизованих підходів до керування

кешем створює труднощі в його супроводі та може негативно впливати на цілісність і надійність системи [18, 21, 22].

У сучасній практиці застосовується низка підходів до кешування, кожен з яких має свої переваги та обмеження. Зокрема, Spring Cache пропонує простий і інтегрований механізм кешування для екосистеми Spring, який дозволяє зберігати результати викликів методів з мінімальними змінами в коді. Його підтримка анотацій робить його зручним у використанні. Проте цей підхід не забезпечує глобальної узгодженості кешу між мікросервісами і вимагає ручного управління інвалідацією, що може бути критичною точкою відмови у розподілених системах [23].

Іншим прикладом є GraphQL та Apollo Client, які реалізують нормалізоване кешування на клієнтському рівні. Такий підхід забезпечує автоматичне оновлення даних під час виконання нових запитів та дозволяє відстежувати взаємозв'язки між сутностями, що значно підвищує узгодженість клієнтських станів. Однак, попри ці переваги, дане рішення має суттєве обмеження: воно не вирішує проблему синхронізації кешу між різними сервісами, а налаштування політик оновлення потребує додаткових зусиль і ручного втручання з боку розробників [19].

Широко застосовуваним інструментом у масштабованих розподілених системах є Redis. Його функціональні можливості охоплюють механізми встановлення часу життя даних (TTL), політики видалення застарілих записів (евікції) та підтримку механізму публікації та підписки (Pub/Sub) для повідомлення про зміни у даних. Redis дає змогу реалізувати гнучкі стратегії інвалідації кешу, але потребує ретельної конфігурації та координації для підтримання консистентності між мікросервісами [21, 24, 25, 27].

Іншим важливим підходом є автоматизоване керування залежностями між сервісами та застосування механізмів відкладеної інвалідації кешу, що детально розглянуто у роботі [23].

Дослідження показує, що автоматизація процесів інвалідації кешу дозволяє знизити затримки при оновленні даних, однак водночас виникають труднощі з точністю відстеження змін у складних графах викликів між мікросервісами. Це

обумовлено непередбачуваністю сценаріїв взаємодії та можливим конфліктом паралельних змін.

Інноваційне рішення, що передбачає інтеграцію кешу на рівні мережесх інтерфейсів, запропоновано у роботі [22], що дозволяє мінімізувати звернення до основного процесора та значно знизити затримки в роботі Memcached.

У роботі [27] досліджено інтеграцію Redis із ZooKeeper для забезпечення високої надійності та автоматизації процесів відновлення у кластерних середовищах. Показано, що дана інтеграція забезпечує ефективне управління розподіленими вузлами, однак у випадках високої конкуренції оновлень виникають додаткові затримки, пов'язані з необхідністю синхронізації станів між нодами. Це підтверджує актуальність розробки нових стратегій балансування навантаження та реплікації даних.

Окремо розглядається централізовані механізми керування кешем, зокрема у роботі [21], де описано централізований підхід до інвалідації кешу в масштабованих вебзастосунках. Перевагами цього підходу є підвищена узгодженість та швидке оновлення даних. Проте він обмежений масштабованістю, оскільки вимагає потужного сервера управління кешем, що стає єдиною точкою відмови.

Одним із традиційних підходів до інвалідації кешу в розподілених вебсистемах є використання централізованого управління кешем, коли відповідальність за актуалізацію даних зосереджена на окремому сервісі. Така архітектурна модель дозволяє централізовано координувати процеси оновлення кешу, що, в теорії, забезпечує високу узгодженість даних та зменшує затримки, пов'язані з розповсюдженням змін у системі [28].

Централізоване кешування широко застосовується в системах, де критично важлива консистентність – наприклад, в електронній комерції, банківських платформах або CRM-системах. Основною перевагою цього підходу є спрощення логіки кешування для окремих мікросервісів. Оскільки правила оновлення і видалення кешованих даних зосереджені у спеціалізованому компоненті,

розробникам не потрібно дублювати логіку інвалідації у кожному сервісі, що знижує ймовірність помилок [29].

У роботі [25, 26] розглянуто централізований механізм управління кешем у контексті мікросервісної архітектури. Дослідження показує, що централізація дозволяє зменшити час реакції на зміни даних, оскільки оновлення поширюється від одного джерела до всіх зацікавлених компонентів. Завдяки використанню брокерів повідомлень (наприклад, Redis Pub/Sub, Kafka або RabbitMQ), система здатна швидко інформувати відповідні мікросервіси про необхідність оновлення кешу або його інвалідацію.

Однак впровадження централізованого підходу не позбавлене істотних обмежень. Найважливішим з них є ризик перевантаження центрального сервера кеш-менеджменту, особливо у випадках з великим навантаженням або великою кількістю взаємозалежних сервісів. У такій конфігурації точка централізації може стати «вузьким місцем» та єдиною точкою відмови, від якої залежить коректна робота всієї кеш-інфраструктури. Крім того, централізоване управління вимагає від усіх сервісів дотримання єдиних протоколів комунікації та підключення до одного джерела істини щодо стану кешу. Це створює додаткову тісну зв'язаність між компонентами системи, що суперечить базовим принципам мікросервісної архітектури – автономності та слабкої залежності між сервісами. Зі збільшенням масштабу системи витрати на підтримку такої централізованої координації суттєво зростають [30, 31, 32].

Іншою проблемою є затримки при поширенні змін, які можуть виникати внаслідок чергування повідомлень, мережових збоїв або обмежень продуктивності брокера. У високонавантажених середовищах це може призводити до тимчасової несинхронності кешу між сервісами, що, у свою чергу, негативно позначається на консистентності даних з точки зору кінцевого користувача. Таким чином, централізоване управління кешем забезпечує важливу функціональність – узгодженість і контроль над актуальністю даних, – однак має низку обмежень, що ускладнюють його використання у масштабованих мікросервісних системах [34]. Зокрема, зростаюча складність координації, ризик перевантаження центрального

компонента та необхідність підтримки спільної інфраструктури створюють передумови для пошуку нових рішень [33, 34, 35].

1.3 Інфраструктурні аспекти масштабування та оптимізації вебсистем

Масштабування сучасних вебсистем охоплює не лише прикладну логіку, а й складну інфраструктурну взаємодію між компонентами, середовищами та командами. Із зростанням кількості фронтендів, сервісів і середовищ розгортання особливого значення набуває запровадження чітко організованих інструментів автоматизації, що забезпечують стабільність, контрольованість і передбачуваність змін [36].

Типовий процес безперервної інтеграції та розгортання (CI/CD) подано на рис. 1.8, де відображено основні етапи: написання коду, збірка, автоматичне тестування та викладка. Такий підхід забезпечує скорочення часу між розробкою та впровадженням змін, підвищуючи якість і надійність програмного забезпечення.

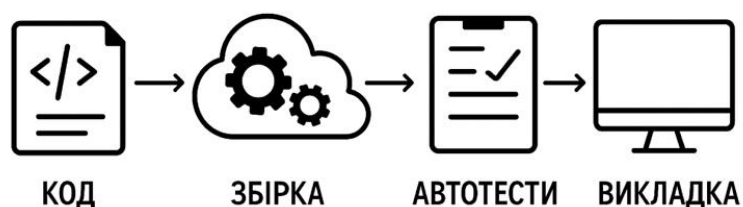


Рисунок 1.8. – Процес безперервної інтеграції та розгортання

Багатофронтендна архітектура породжує додаткові виклики для процесів CI/CD, зокрема щодо керування залежностями, забезпеченням узгодженості версій та раціонального використання обчислювальних ресурсів. У таких умовах зростає роль ефективного кешування, повторного використання артефактів і оптимізації пайплайнів для прискорення циклів розробки без втрати керованості та якості [37].

Особливу увагу привертає потреба в балансі між гнучкістю інфраструктури та її уніфікацією. Автоматизоване розгортання, централізоване управління конфігураціями й інтеграція з інструментами моніторингу дозволяють масштабувати системи без пропорційного збільшення витрат на підтримку.

1.3.1 Мультифронтендна архітектури в умовах масштабування

Розвиток вебсистем із підтримкою інтерфейсів користувача, орієнтованих на різні бренди або функціональні сегменти, привів до зростання популярності мультифронтендної архітектури, яка дозволяє реалізовувати кілька фронтенд-додатків на базі спільного коду. Такий підхід забезпечує масштабованість, повторне використання компонентів, уніфікацію користувацького досвіду та централізоване управління залежностями [38].

Ключовим технічним рішенням у цьому контексті є застосування монорепозиторної стратегії, яка передбачає зберігання кількох фронтенд-проектів, бібліотек і спільних компонентів в одному репозиторії. Даний підхід спрощує керування версіями, спільне тестування та стандартизацію архітектурних патернів, але водночас створює ризики неявних міжпроектних залежностей, які можуть призводити до неузгоджених змін та конфліктів [39].

Одним із фундаментальних викликів при масштабуванні мультифронтендних систем є розмежування бізнес-логіки та стилізації, що дозволяє підтримувати функціональну єдність при збереженні візуальної гнучкості. Це особливо важливо у мультибрендових проєктах, де спільні функції повинні зберігати консистентність, а оформлення – відповідати вимогам бренду. Забезпечення цієї модульності вимагає чітко структурованої архітектури, яка враховує поділ на ядро, теми, бібліотеки компонентів і адаптаційні шари для брендів [40].

Крім того, монорепозиторії з великою кількістю компонентів ускладнюють контроль залежностей, оскільки зміни в одній загальній бібліотеці можуть мати

каскадний ефект на десятки проєктів. Це підвищує вимоги до засобів перевірки сумісності та автоматизованого тестування на рівні всього репозиторію [41].

Таким чином, мультифронтендні архітектури відкривають значні можливості для масштабування вебсистем, але водночас створюють нові виклики, пов'язані з керуванням залежностями, модулярністю та адаптивністю.

1.3.2 Автоматизація CI/CD для багатокomпонентних вебсистем

Автоматизація процесів інтеграції, тестування та розгортання є невід'ємною складовою забезпечення ефективного життєвого циклу розробки програмного забезпечення. В умовах багатокomпонентних вебсистем, зокрема в архітектурах із кількома фронтендами, процеси CI/CD відіграють ключову роль, забезпечуючи стабільність, узгодженість та масштабованість розгортання [42].

Одна з головних переваг впровадження CI/CD – скорочення часу релізу, зниження кількості помилок, підвищення якості програмного забезпечення. Результати дослідження [43] демонструють, що автоматизовані пайплайни можуть знизити кількість помилок при розгортанні на 70% і скоротити обсяг незапланованих робіт на 22%. Однак, більшість наявних підходів до CI/CD зосереджені на загальних принципах автоматизації та не враховують специфіку розгортання мультифронтендних систем в межах монорепоzиторію.

Конкретною складністю у таких проєктах є оркестрація пайплайнів, яка має забезпечувати незалежну та паралельну збірку кількох фронтендів, що користуються спільними бібліотеками. У цьому контексті, дослідження [44] підкреслює важливість гнучких конфігурацій, які враховують залежності між модулями й дозволяють оптимізувати виконання на рівні окремих компонентів.

Іншою важливою особливістю є потреба у підтримці різних середовищ – локальних, тестових та продакшн. CI/CD повинна забезпечувати автоматичне тестування і перевірку кожного фронтенду з урахуванням особливостей його збірки й залежностей. Крім того, використання контейнеризації через Docker і

оркестрацію за допомогою Kubernetes надає широкі можливості для автоматизації, проте такі підходи не завжди адаптовані до потреб моногеро-архітектури без додаткових налаштувань [40].

1.3.3 Оптимізація пайплайнів

Оптимізація ресурсів при побудові й розгортанні застосунків у багатокомпонентних вебсистемах є критичним аспектом інфраструктурної ефективності. Основними напрямками покращення CI/CD-процесів у цьому контексті є використання кешування, ефективне управління артефактами та оптимізація обчислювальних ресурсів [45].

Кешування в межах CI/CD-пайплайнів дозволяє зберігати проміжні результати – такі як встановлені залежності або скомпільовані пакети – для повторного використання в наступних збірках. Кешування між етапами пайплайну сприяє зменшенню часу відновлення середовища, що є особливо важливим для проєктів із високою частотою змін. Крім того, коректно сформована політика інвалідації кешу є критично важливою, оскільки застарілі або неконсистентні кешовані артефакти можуть призводити до помилок під час процесів розгортання [44].

Окрему роль відіграє керування артефактами, тобто скомпільованими пакетами, бібліотеками, Docker-образами, що зберігаються для повторного використання в різних середовищах. Як зазначено у [44], належна стратегія зберігання артефактів дозволяє уникнути зайвих перезбірок і спрощує відтворення збірки для попередніх версій системи. Проте для забезпечення оптимального використання сховища важливими є політики автоматичного очищення, про які йдеться у: регулярне видалення застарілих збірок і логів дає змогу запобігти перевантаженню інфраструктури [4].

Ще одним напрямом оптимізації є раціональне використання CI/CD-ранерів. Повторне використання ранерів для послідовних комітів в одній гілці та пропуск

збірки для незмінних компонентів знижує час виконання пайплайнів без шкоди для якості. Окрім того, пріоритетизація легких завдань дозволяє розвантажити систему й пришвидшити зворотний зв'язок для розробників [44].

Як засвідчено у роботі [36], паралельне виконання завдань в рамках монорепозиторію є ще одним ефективним методом оптимізації. Такий підхід дозволяє запускати збірки незалежних пакетів одночасно, значно скорочуючи загальний час пайплайну (до 30% у практичних кейсах). Проте ефективність цього методу залежить від грамотної сегментації задач і уникнення конфліктів при одночасному доступі до спільних ресурсів.

Таким чином, комплексне впровадження кешування, артефакт-менеджменту та оптимального використання обчислювальних ресурсів є ключем до побудови швидких, стабільних і масштабованих CI/CD-процесів для мультифронтендних архітектур.

1.3.4 Обмеження сучасних підходів

Попри широке впровадження DevOps-практик і розвиток інструментів автоматизації, інфраструктура CI/CD для мультифронтендних архітектур в межах монорепозиторію все ще залишається недостатньо опрацьованою [46]. Більшість досліджень зосереджені на узагальнених аспектах автоматизації, таких як інтеграція безперервної доставки, контейнеризація або безпека пайплайнів [47, 48], проте не розглядають специфіку керування кількома фронтендами, що використовують спільні бібліотеки та мають ізольовані середовища виконання.

CI/CD у хмарних середовищах активно реалізується із застосуванням технологій Terraform, Kubernetes та інших інструментів інфраструктури як коду [49]. Однак такі підходи рідко фокусуються на проблематиці розгортання проєктів, зокрема – повторному використанні бібліотек, паралельному збиранні залежностей або збереженні ізольованих збірок у спільному монорепозиторію. Аналогічно, дослідження [5] висвітлює переваги мікрофронтендів у плані

модульності й автономії команд, однак питання їх ефективної інтеграції у централізовані CI/CD-пайплайни залишається відкритим.

Наявні приклади, зокрема наведені у роботі [43], демонструють успішне застосування монорепозиторних інструментів у комерційних проєктах. Проте навіть у таких прикладах увага зосереджена переважно на організації структури репозиторію, тоді як інфраструктурна частина CI/CD, зокрема кешування, інвалідація артефактів, тестування залежних модулів потребують подальшого дослідження.

Розробка узагальненої CI/CD-моделі для таких систем є перспективним напрямом у галузі інженерії програмного забезпечення.

Висновки до розділу 1

Сучасні тенденції розвитку вебтехнологій демонструють поступовий перехід від монолітних архітектур до розподілених систем із високим ступенем масштабованості, адаптивності та стійкості до навантажень. Такий перехід зумовлює необхідність переосмислення архітектурних рішень, механізмів кешування та інфраструктурної організації життєвого циклу розгортання й супроводу програмного забезпечення. У цьому контексті особливої уваги набувають підходи, що забезпечують ефективну взаємодію фронтенду з мікросервісним середовищем, серед яких патерн(BFF) посідає ключове місце.

Водночас, попри наявні розробки у сфері його застосування, залишається низка відкритих питань, що потребують подальшого дослідження. Зокрема, не повністю вирішено проблему формалізації архітектурних принципів побудови BFF у високонавантажених системах з динамічною структурою мікросервісів. Недостатньо розробленими залишаються методи оптимізації агрегації даних із різнорідних API та механізми забезпечення узгодженості клієнтського стану при паралельній взаємодії з REST і GraphQL сервісами. Також відкритим напрямом є створення уніфікованої моделі фабрики API-провайдерів, здатної адаптивно

керувати маршрутизацією запитів, розмежуванням доступу та кешуванням у контексті мультифронтендних архітектур.

Проблематика інвалідації кешу у мікросервісних архітектурах залишається одним із ключових чинників, що впливають на узгодженість даних і загальну продуктивність систем. Імперативні підходи до керування кешем ускладнюють підтримку, масштабування та супровід, особливо в умовах великої кількості взаємопов'язаних сервісів. У відповідь на ці виклики дедалі більшої уваги набуває декларативний підхід до інвалідації кешу, який дає змогу формалізувати правила оновлення, мінімізувати ризики, пов'язані з людським фактором, та забезпечити прозорість і передбачуваність процесів підтримання актуальності даних. Такий підхід розглядається як ефективна альтернатива централізованим рішенням, зберігаючи при цьому переваги автоматизованого керування кешем.

Інфраструктурні аспекти, зокрема автоматизація процесів CI/CD, відіграють визначальну роль у забезпеченні стійкості та масштабованості багатокомпонентних вебсистем. У мультифронтендних архітектурах особливо актуальними є питання ефективного управління залежностями, артефактами та ресурсами. Комплексна реалізація кешування, оптимізації пайплайнів та стандартизації сценаріїв розгортання сприяє підвищенню надійності й швидкодії процесів постачання оновлень.

2. МОДЕЛЮВАННЯ АРХІТЕКТУРИ ВЗАЄМОДІЇ КЛІЄНТСЬКИХ І СЕРВЕРНИХ КОМПОНЕНТІВ НА ОСНОВІ ПАТЕРНУ BFF У ВИСОКОНАВАНТАЖЕНИХ ВЕБСИСТЕМАХ

Патерн BFF активно використовується у масштабованих вебсистемах для адаптації API до потреб конкретного клієнтського інтерфейсу. Його застосування дозволяє зменшити навантаження на фронтенд, централізувати бізнес-логіку та покращити контроль над клієнтським станом. Однак у високонавантажених додатках виникають специфічні вимоги до узгодженості даних, безпеки доступу та масштабованості, що потребують додаткового аналізу та вдосконалення архітектури BFF-рішень. Під високонавантаженими вебдодатками у цьому контексті розуміють системи, що одночасно обробляють великі обсяги трафіку, працюють у розподіленому середовищі з підвищеними вимогами до продуктивності, узгодженості даних, безпеки та відмовостійкості, а також інтегрують численні мікросервіси та зовнішні API. У даному розділі розглянуто наявні підходи до побудови BFF та запропоновано альтернативну архітектуру для складних сценаріїв інтеграції.

2.1 Необхідність об'єднання даних з кількох мікросервісів для забезпечення цілісного стану клієнта

Мікросервісна архітектура передбачає поділ системи на незалежні сервіси, кожен з яких реалізує окрему бізнес-функцію або доменну область. Така декомпозиція підвищує масштабованість та гнучкість розробки, однак у клієнтських застосунках вона створює додаткові виклики: дані опиняються розподіленими між різними сервісами, що ускладнює їх цілісне подання та потребує агрегування на фронтенді або на проміжному серверному шарі [48,50].

Без наявності єдиного рівня агрегації, фронтенд змушений самостійно координувати численні запити до окремих мікросервісів, обробляти помилки, вирішувати конфлікти даних та синхронізувати стан між компонентами. Така модель призводить до:

- зростання складності клієнтської логіки, оскільки необхідно враховувати взаємозалежності між сервісами;
- збільшення часу завантаження сторінки через паралельні або послідовні запити до багатьох джерел;
- низької стабільності інтерфейсу, особливо при зміні структур API або появи тимчасової недоступності одного з мікросервісів;
- відсутності централізованого контролю над кешуванням, трансформацією та авторизацією запитів.

Для усунення зазначених проблем доцільним є використання проміжного шару агрегації – зокрема, патерну BFF, який дозволяє централізовано формувати представлення стану клієнта на основі узгоджених даних з кількох джерел. BFF бере на себе відповідальність за оркестрацію запитів, об'єднання результатів, нормалізацію відповідей, обробку помилок і кешування, що значно знижує навантаження на фронтенд і покращує узгодженість клієнтського інтерфейсу [49].

У складних вебсистемах, де взаємодія з клієнтом часто передбачає високий рівень кастомізації, багатокористувацький доступ, підтримку сесій або різні ролі, централізоване агрегування даних є критичним для стабільності й надійності. Крім того, агрегація дає змогу реалізовувати складну логіку бізнес-правил, яка не може бути повністю делегована до фронтенду з міркувань безпеки та продуктивності [8].

У деяких випадках BFF може виступати як агрегатор не лише HTTP API, але й подієвих потоків (наприклад, через Kafka або вебсокет), забезпечуючи узгодженість стану в реальному часі. Такий підхід є особливо актуальним для дашбордів, систем моніторингу, онлайн-операційних панелей тощо [50].

Таким чином, забезпечення цілісного стану клієнта в умовах мікросервісної архітектури вимагає наявності окремого шару об'єднання та адаптації даних.

Використання шаблону BFF дозволяє не лише спростити клієнтську логіку, а й підвищити надійність, масштабованість та узгодженість клієнтського досвіду.

2.2 Обмеження наявних рішень у контексті B2B-сегменту

У контексті B2B-систем, де ключовими вимогами є надійність, контроль доступу, масштабованість і підтримка різномірних клієнтських інтерфейсів, застосування шаблону BFF стикається з низкою практичних обмежень. Попри універсальність підходу та доступність готових рішень, таких як WunderGraph, Apollo Federation або GraphQL Mesh, не всі з них повною мірою відповідають специфіці B2B-сценаріїв [51].

У B2B-середовищі кожен клієнт (компанія, партнер, зовнішній провайдер) має власний набір ролей, прав доступу та бізнес-логіки. Наявні інструменти BFF зазвичай не передбачають гнучких механізмів контекстної авторизації, яка враховує не лише користувача, а й організаційні межі. Це ускладнює реалізацію ізольованого доступу до мікросервісів для окремих бізнес-клієнтів або інстансів застосунку [52].

У великих B2B-платформах різні клієнти можуть працювати за зміненими або версійованими контрактами API. Статична генерація SDK або GraphQL-схем, притаманна багатьом інструментам (наприклад, WunderGraph, Apollo Federation чи GraphQL Mesh), не враховує можливість адаптації логіки в реальному часі без повного перекомпілювання та перевикладки. Це обмежує швидкість реакції на специфічні вимоги клієнтів [12].

B2B-системи часто мають складну ієрархію користувачів, організацій, контрактів, тарифних планів та обмежень. Типові BFF-рішення не включають вбудованої підтримки для генерації тимчасових ключів доступу, токенів із обмеженим терміном дії або інтеграції з багаторівневими системами управління ідентифікацією та доступом. Це вимагає додаткової реалізації логіки безпеки на рівні кастомного посередника або окремих сервісів [53].

Інструменти, орієнтовані на універсальність, часто не оптимізовані під сценарії з великою кількістю одночасних підключень, запитів на агрегацію або потокової обробки даних. У B2B-системах, де клієнти можуть бути повноцінними інтеграційними партнерами з великою кількістю запитів (наприклад, системи планування ресурсів підприємства, або системи управління взаєминами з клієнтами), виникає потреба у поглибленому налаштуванні механізмів кешування, систем черг обробки запитів, засобів контролю частоти звернень та підсистем логіювання. Багато платформ таких можливостей не надають в базовому функціоналі або надають з обмеженою конфігурацією [54].

Часто у B2B-додатках фронтенд не лише відображає дані, а й виконує динамічну бізнес-логіку – наприклад, верифікацію форм, застосування правил ціноутворення, фільтрацію даних згідно з умовами контракту. У типових реалізаціях BFF дана логіка або дублюється у фронтенд, або змушує створювати окремі мікросервіси, що ускладнює підтримку. В ідеалі така логіка має існувати безпосередньо в адаптаційному BFF-шарі, що потребує розширюваної структури сервісу, якої не завжди передбачено [13].

B2B-рішення часто орієнтовані на модель, де кожен клієнт ізольований у рамках своєї «віртуальної організації». Більшість готових рішень не пропонує вбудованих інструментів для побудови ізольованих BFF-екземплярів або динамічної маршрутизації залежно від домену чи токена користувача, що ускладнює реалізацію сегментації даних [55].

Розвиток шаблону BFF спричинив появу низки платформ та фреймворків, що дозволяють швидко інтегрувати декілька API-провайдерів у єдину точку доступу для фронтенд-застосунків. Такі інструменти, як WunderGraph, Apollo Federation, GraphQL Mesh надають розробникам готові засоби для конфігураційного опису джерел даних, генерації клієнтського коду та централізації взаємодії з мікросервісами [51,56, 60, 62].

Попри зручність і продуктивність, подібні рішення мають низку архітектурних і організаційних обмежень, особливо у високонавантажених системах з великою кількістю фронтенд-екземплярів, динамічними вимогами

клієнтів та вимогами до багаторівневого доступу. Проблеми виникають як на рівні побудови віртуального API-графа, так і у процесах управління конфігураціями, контролю доступу, масштабування клієнтських інтерфейсів і підтримки версійності.

Ефективне впровадження BFF у B2B-контексті вимагає критичного переосмислення можливостей стандартних інструментів, виявлення їхніх слабких місць та адаптації до потреб реального середовища розробки.

2.2.1 Аналіз та обмеження WunderGraph

Фреймворк WunderGraph є одним з найпоширеніших рішень для реалізації архітектурного шаблону BFF, орієнтованого на інтеграцію множинних API-провайдерів у єдиний віртуальний інтерфейс. Основною особливістю WunderGraph є можливість створення так званого віртуального графа, який об'єднує декілька GraphQL (та REST) сервісів у єдину схему, доступну через одну кінцеву точку [57].

Після імпорту необхідних API-ендпоінтів до конфігураційного файлу WunderGraph автоматично генерує код клієнта (SDK), включаючи типи, запити та мутації. Це дозволяє фронтенд-розробникам швидко інтегрувати агреговані дані без потреби ручної конфігурації HTTP-запитів. Таким чином, WunderGraph позиціонується як рішення для централізації API-взаємодій у фронтенд-застосунках [57].

Однак на практиці, особливо в ентєрпрайз, або B2B-контексті, виникають специфічні сценарії, які виявляють обмеження підходу WunderGraph. Одним із таких сценаріїв є потреба в розмежуванні публічного та приватного API-доступу. У ситуаціях, коли існує кілька екземплярів фронтенду, кожен із яких працює в окремому просторі імен і використовує спільні мікросервіси, виникає необхідність у генерації унікальних облікових даних для кожного проєкту [57].

У подібних випадках застосовується шаблон, за яким запити від фронтенду формуються як єдиний POST-запит зі спеціальною структурою, що надходить до

централізованого сервера. Цей сервер перехоплює запит, проводить авторизацію, розділяє його на підзапити (REST або GraphQL) до внутрішніх мікросервісів і повертає агреговану відповідь. Такий підхід дозволяє приховати реальну структуру API, підвищуючи безпеку, і водночас забезпечити адаптивність доступу до ресурсів залежно від ідентичності клієнта.

Разом з тим, підтримка віртуальних графів у WunderGraph має свої виклики. Зокрема, при оновленні структури одного з мікросервісів необхідно вручну відслідковувати і змінювати всі пов'язані запити у графі. Це створює ризик помилок та ускладнює супровід у масштабних системах. Як альтернатива, в даному дослідженні пропонується концепція фабрики провайдерів API, яка дозволяє централізовано створювати адаптери для кожного сервісу. У разі змін в API мікросервісу необхідно оновити лише один компонент провайдера, а не всю конфігурацію віртуального графа (рис. 2.1).

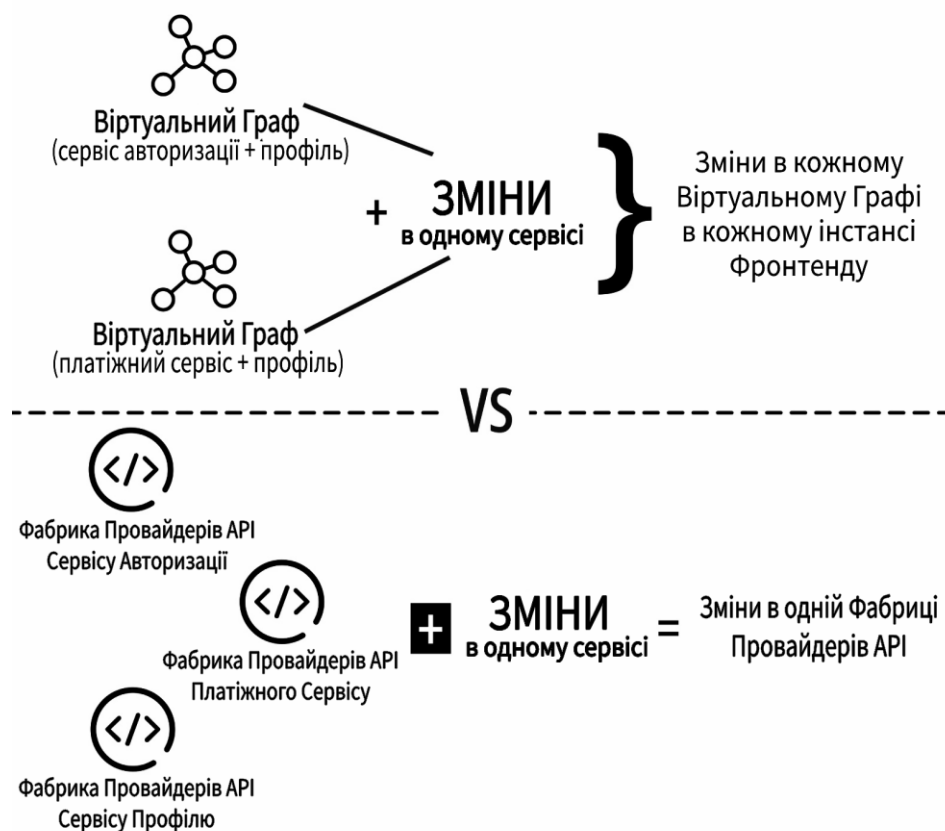


Рисунок 2.1 – Порівняння підтримки Віртуальних Графів з фабрикою провайдерів API

Подібним чином постає питання масштабування фронтенд-екземплярів. У випадку, коли платформа обслуговує десятки клієнтських інтерфейсів (наприклад, для різних партнерів або внутрішніх систем), зв'язування кожного фронтенду з окремою конфігурацією WunderGraph стає непрактичним. Фабрика провайдерів API у поєднанні з централізованим Публічним API шлюзом дозволяє використовувати спільну логіку доступу, мінімізуючи дублювання та спрощуючи підтримку (рис. 2.2).

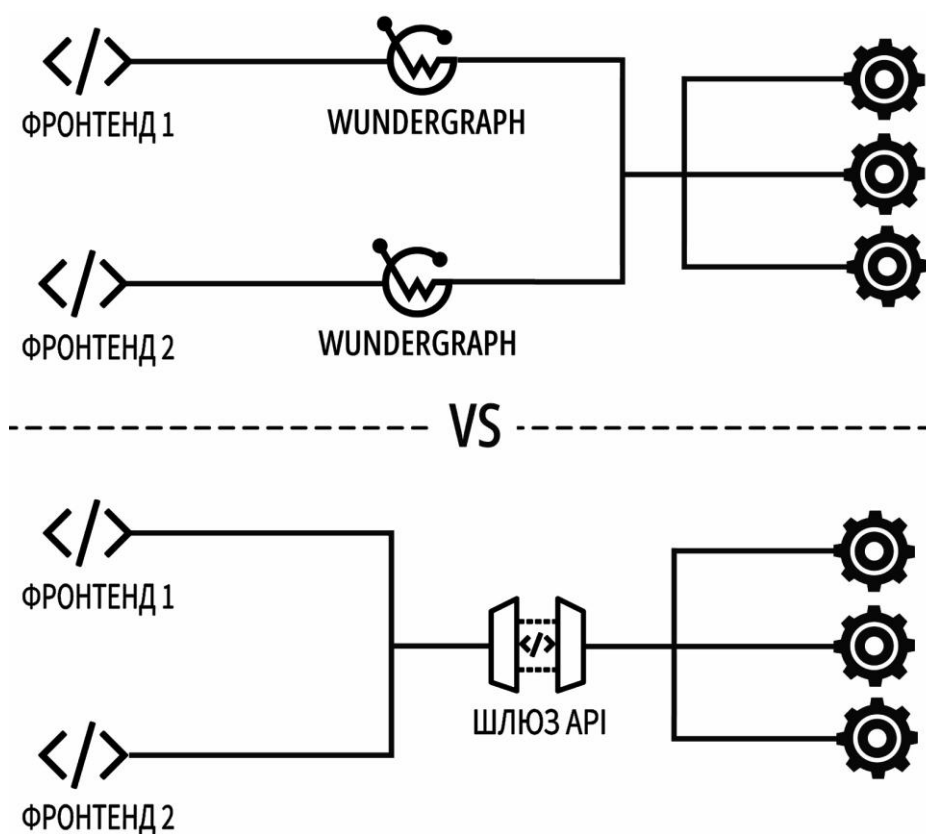


Рисунок 2.2 – Порівняння масштабування Фронтенд інстансів при використанні Wundergraph та рішення з використанням фабрики провайдерів API через API шлюз

Таким чином, WunderGraph є потужним інструментом для об'єднання розрізнених сервісів у єдиний API-рівень, однак його ефективне використання у вебсистемах потребує розширення базової архітектури за допомогою додаткових

шаблонів та модулів, зокрема механізмів ізоляції доступу, фабрик провайдерів API і адаптивного API-маршрутизатора.

2.2.2 Аналіз та обмеження Apollo Federation

Apollo Federation широко застосовується у мікросервісних архітектурах для об'єднання множини GraphQL-сервісів у єдиний семантичний суперграф, з яким працюють усі клієнтські застосунки. У цій моделі кожний субграф представляє окрему доменну область, тоді як централізований рівень композиції формує єдину глобальну схему [60]. Фронтенд-застосунки звертаються до суперграфа як до одного логічного API-шару, а надалі запити розподіляються між субграфами відповідно до планувальника запитів. Такий підхід спрощує інтеграцію мікросервісів та забезпечує уніфікований спосіб доступу до даних, що зображено на рис. 2.3.

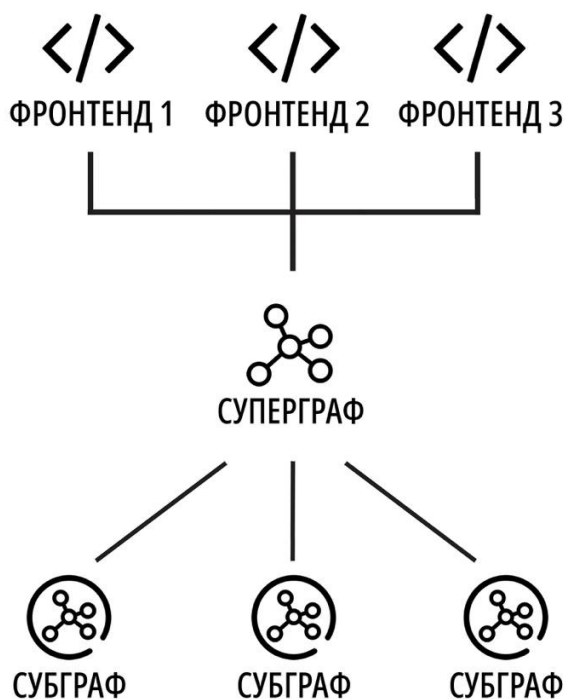


Рисунок 2.3 – Принцип роботи Apollo Federation у мікросервісній архітектурі

Разом із перевагами Apollo Federation має низку суттєвих обмежень, що ускладнюють його використання у мультибрендових або багатопрокольних вебплатформах. Оскільки Apollo Federation вимагає, щоб усі доменні сервіси були оформлені як GraphQL-субграфи та експортували схеми у форматі SDL, стає неможливим розділення описів сервісів за пакетами або форматами. Непридатними виявляються сценарії, де частина доменів реалізована як REST OpenAPI, а інша – як GraphQL, адже для будь-якого REST-сервісу необхідно створювати окремий GraphQL-адаптер. Це породжує дублювання логіки, збільшує операційні витрати, ускладнює супровід та порушує модульність архітектури [58, 62].

Ще одне фундаментальне обмеження полягає у відсутності можливості виконувати додаткову нормалізацію даних на рівні шлюзу. Субграфи у Apollo Federation мають генерувати фінальні дані, тоді як шлюз або роутер відповідають лише за маршрутизацію та композицію результатів. На цьому рівні відсутні механізми об'єднання або перетворення структур між різними доменами, а також інструменти для адаптивної логіки залежно від типу клієнта. Складно реалізувати базову нормалізацію або бізнес-логіку обробки вихідних даних, що є критично важливим для систем із потребою централізованої узгодженості або перетворення відповідей від неоднорідних сервісів [58, 61].

Також Apollo Federation не забезпечує гнучкого управління доступом у багатоклієнтських і мультибрендових системах. Усі фронтенди взаємодіють із одним і тим самим суперграфом, який визначає глобальну структуру API. Apollo Federation не підтримує формування індивідуальних API-профілів для різних партнерів, неможливо ввімкнути або вимкнути окремі домени для конкретних брендів, і не передбачено окремих правил доступу на рівні субграфів для окремих клієнтів. У результаті будь-які зміни у структурі суперграфа впливають на всі фронтенд-застосунки, а спроби обмежити доступ або змінити форму даних для окремого партнерського рішення потребують створення додаткових прошарків або дублювання конфігурації [56].

Незважаючи на зручність у побудові єдиного глобального графа даних, Apollo Federation має суттєві обмеження у сценаріях, де необхідні

мультипротокольність, проміжна нормалізація даних, або гнучке формування індивідуальних API-профілів для різних клієнтів. Ці характеристики значно звужують можливість застосування даної технології у високонавантажених вебсистемах із множинними незалежними фронтендами, які потребують адаптивної поведінки API залежно від контексту використання.

2.2.3 Аналіз та обмеження GraphQL Mesh

GraphQL Mesh є інструментом інтеграції, орієнтованим на уніфікацію доступу до різнорідних джерел даних шляхом автоматичного перетворення REST, gRPC, OpenAPI, SOAP, GraphQL та інших протоколів у єдину узгоджену GraphQL-схему. На відміну від Apollo Federation, який працює виключно з GraphQL-сервісами, Mesh дозволяє під'єднувати різнорідні мікросервіси без необхідності створення ручних адаптерів, що робить підхід привабливим для систем з історичним або неоднорідним API-ландшафтом. Загальний принцип роботи GraphQL Mesh, представлений на рис. 2.4, демонструє, що всі фронтенд-застосунки взаємодіють із єдиним Mesh-шаром, який агрегує декілька протоколів і формує спільну GraphQL-схему поверх них [59].

Попри універсальність, GraphQL Mesh має низку суттєвих обмежень, які знижують його придатність для використання у великомасштабних вебсистемах із множинними незалежними фронтендами. Центральною проблемою є те, що GraphQL Mesh формує один глобальний інтеграційний шар, де вся логіка перетворення, нормалізації та об'єднання даних концентрується у межах єдиної згенерованої схеми [60]. У результаті зміни будь-якого з під'єднаних сервісів потребують оновлення всієї Mesh-конфігурації, що створює високий ризик каскадного впливу та ускладнює незалежну еволюцію доменних модулів. Крім того, суттєво зростає складність супроводу, оскільки логіка трансформацій перестає бути модульною й втрачає можливість ізольованого розвитку окремих компонентів [63].

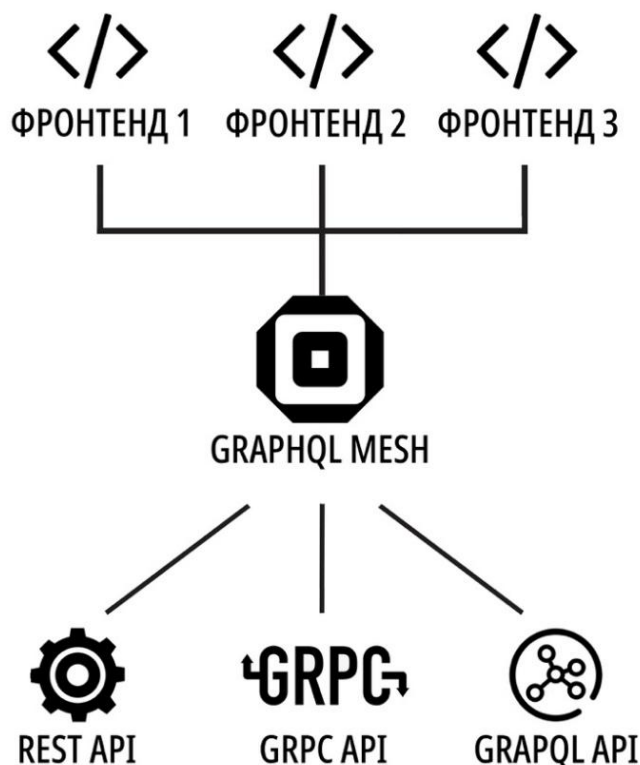


Рисунок 2.4 – Принцип роботи GraphQL Mesh у мультисервісній архітектурі

Попри універсальність, GraphQL Mesh має низку суттєвих обмежень, які знижують його придатність для використання у великомасштабних вебсистемах із множинними незалежними фронтедами. Центральною проблемою є те, що GraphQL Mesh формує один глобальний інтеграційний шар, де вся логіка перетворення, нормалізації та об'єднання даних концентрується у межах єдиної згенерованої схеми [60]. У результаті зміни будь-якого з під'єднаних сервісів потребують оновлення всієї Mesh-конфігурації, що створює високий ризик каскадного впливу та ускладнює незалежну еволюцію доменних модулів. Крім того, суттєво зростає складність супроводу, оскільки логіка трансформацій перестає бути модульною й втрачає можливість ізольованого розвитку окремих компонентів [59].

Іншим важливим обмеженням є відсутність нативної підтримки багатоклієнтних сценаріїв. GraphQL Mesh створює єдину GraphQL-схему, спільну для всіх клієнтів, незалежно від їхнього доменного контексту, рівня доступу чи

функціональних потреб. Це означає, що неможливо сформувати індивідуальні API-профілі для різних партнерів або брендів, а також неможливо вимкнути чи приховати окремі домени без додаткового шару авторизації. Така модель фактично унеможливорює створення ізольованих API-відповідей для різних фронтендів, що є критично важливим для мультибрендових платформ з різними бізнес-вимогами [60].

Хоча GraphQL Mesh забезпечує уніфікований доступ до різнорідних API та зменшує потребу у ручному створенні адаптерів, його централізована природа, відсутність модульної локалізації змін, недостатня підтримка мультибрендовості та значні обмеження продуктивності роблять його менш придатним для високонавантажених та мультибрендових вебсистем. У контексті цього дослідження такі обмеження є принципово важливими, оскільки поставлене завдання передбачає створення архітектури, здатної чітко розділяти публічний та приватний API-рівні, забезпечувати адаптивність доступу для різних типів клієнтів і водночас підтримувати модульність інтеграційного шару [49].

З огляду на ці вимоги у рамках роботи розробляється концепція фабрики провайдерів API – підходу, який дозволяє реалізувати індивідуальні адаптери для окремих сервісів, локалізувати зміни в межах одного доменного модуля та забезпечувати протокольну гнучкість без централізації логіки перетворення даних. Ключовим елементом запропонованої архітектури є Публічний API-шлюз, що виконує функції маршрутизації, агрегування політик доступу та формування клієнтсько-орієнтованого API-профілю на основі контексту запиту.

2.3 Фабрика провайдерів API, шлюз, розділення API на публічне та приватне

Розглянемо ключові архітектурні сутності, що формують контекст дослідження, а також сформулюємо базові вимоги до взаємодії між ними.

На рисунку 2.5 схематично зображено три основні компоненти моделі:

- API мікросервісів;
- фронтенд сервер (BFF-шар);
- фронтенд клієнт.

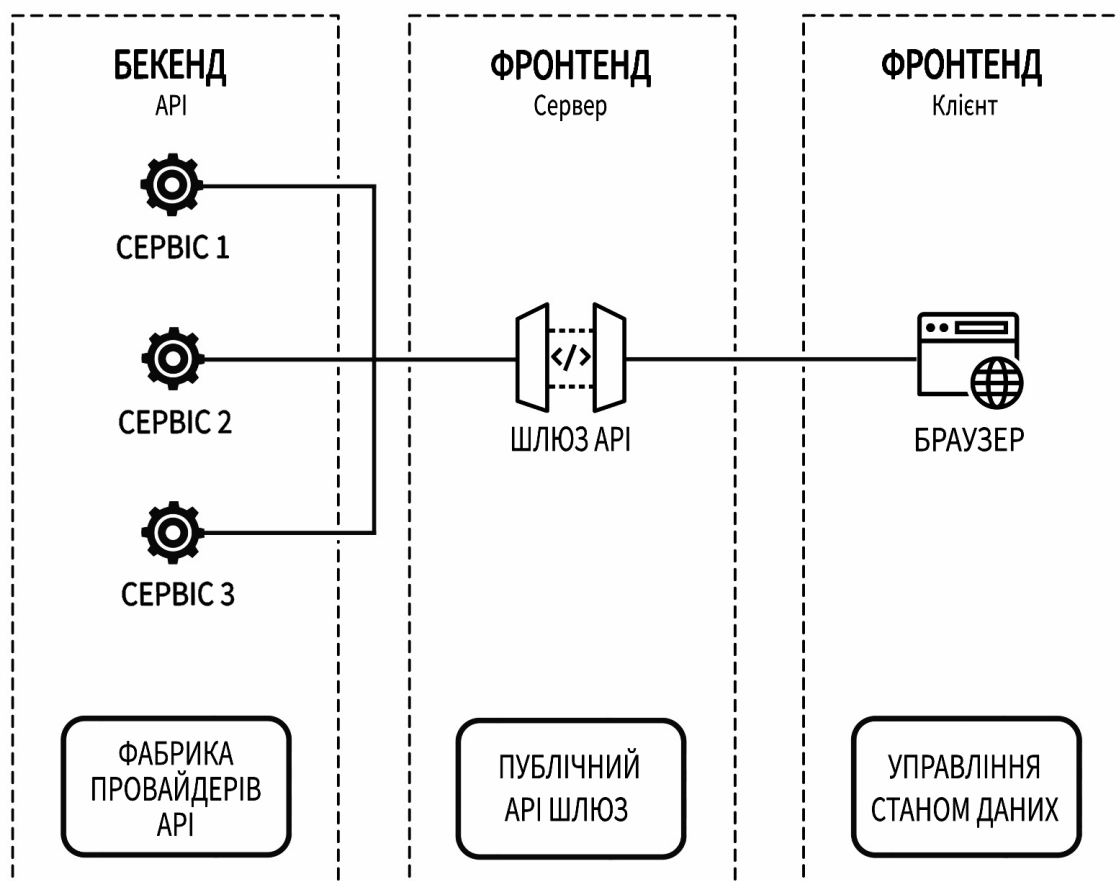


Рисунок 2.5 – Ключові сутності архітектури побудови фронтенд за шаблоном BFF

API мікросервісів – це базовий рівень бізнес-логіки, що складається з окремих мікросервісів із власними API. Для спрощення інтеграції з цими API у межах фронтенд-архітектури доцільно використовувати фабричний підхід до провайдерів API, який дозволяє централізовано конфігурувати доступ до мікросервісів, із можливістю гнучкого керування автентифікацією, авторизацією та маршрутизацією [61].

Фронтенд сервер (BFF) – даний компонент відіграє роль проксі-шару та логічного посередника між клієнтським застосунком і мікросервісами. Його функціональність включає:

- формування агрегованих відповідей із кількох мікросервісів;
- попередній рендер UI (для рендера на стороні сервера/клієнта);
- валідацію даних та впровадження додаткових політик безпеки;
- інкапсуляцію бізнес-логіки, пов’язаної з презентаційним шаром.

У рамках даного підходу фронтенд сервер також реалізує публічний API шлюз, який визначає зовнішній інтерфейс взаємодії для окремих фронтенд-інстансів і забезпечує розмежування доступу до мікросервісів через публічні та приватні API.

Фронтенд клієнт – повнофункціональний UI-застосунок, який працює безпосередньо у браузері користувача, відповідає за:

- рендер графічного інтерфейсу;
- ініціацію запитів до фронтенд сервера (а не напряму до мікросервісів);
- управління локальним станом, включно з кешуванням та реактивною передачею даних між компонентами;
- авторизацію користувача в контексті виділеного інстансу застосунку.

У даному дослідженні особлива увага приділяється механізму генерації ізольованих облікових даних для окремих інстансів фронтенду, що дає змогу застосовувати різні політики доступу до мікросервісів залежно від контексту користувача.

2.3.1 Особливості розділення публічного та приватного API

У складних вебсистемах, особливо у високонавантажених веб додатках, розділення публічного та приватного API є важливим аспектом безпеки, масштабованості та управління доступом. Це дозволяє чітко відмежувати

зовнішній інтерфейс взаємодії з клієнтами (фронтенд) від внутрішніх сервісів та логіки обробки даних (бекенд) [62].

Публічне API (рис. 2.6), виступає в ролі зовнішнього шлюзу – єдиного доступного клієнтам входу в систему. Це API має бути захищеним від прямого впливу з боку користувачів та зовнішніх агентів, обробляючи лише визначені запити. Основне завдання – забезпечити стандартизовану, контрольовану точку входу для фронтенд-додатків, мобільних клієнтів або зовнішніх систем.

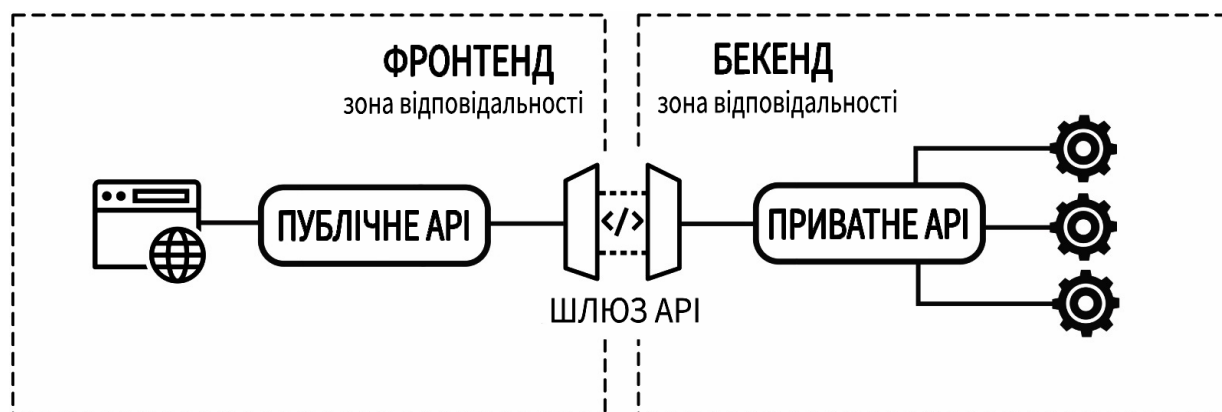


Рисунок 2.6 – Розділення API на публічне та приватне

Приватне API, навпаки, обслуговує низькорівневі внутрішні сервіси та мікросервіси, які не повинні бути доступними безпосередньо ззовні. Такі API можуть містити складну бізнес-логіку, чутливі дані або мати слабкі місця вразливості, якщо їх зробити загальнодоступними. Тому вони повинні бути приховані за API-шлюзом, який виконує роль захисного проміжного прошарку – трансформує, перевіряє, маршрутизує та фільтрує запити до внутрішньої інфраструктури [63].

Однак реалізація такого розмежування породжує низку технічних та організаційних викликів.

Однією з основних проблем у процесі проєктування публічних API для високонавантажених вебсистем є складність керування правами доступу. У мультифронтендних системах виникає необхідність коректного розподілу облікових даних і токенів відповідно до повноважень кожного клієнта з метою

ізоляції доступу до внутрішніх ресурсів. Це зумовлює потребу у впровадженні формалізованої та надійної системи авторизації, яка зазвичай реалізується на рівні API-шлюзу.

Додатковим ускладнювальним чинником є маршрутизація запитів, що надходять через публічне API. Такі запити часто мають агрегований характер і включають декілька логічно пов'язаних операцій, які повинні бути розподілені між відповідними мікросервісами. Реалізація механізмів декомпозиції та координації подібних запитів може здійснюватися як у межах BFF, так і на рівні API-шлюзу, що призводить до підвищення складності архітектурних рішень [63].

Важливим аспектом є також уніфікація форматів даних. Внутрішні сервіси часто відрізняються за структурою відповідей, форматом повідомлень про помилки та способом представлення даних. У зв'язку з цим публічне API повинне забезпечувати узгоджене та стандартизоване подання інформації для клієнтських застосунків, абстрагуючи їх від внутрішніх особливостей реалізації сервісів.

Окрему увагу необхідно приділяти питанням безпеки. Надання доступу до внутрішніх API через публічні інтерфейси суттєво підвищує ризик реалізації потенційних атак. Тому проєктування публічного API має передбачати мінімізацію поверхні атаки шляхом обмеження кількості доступних кінцевих точок, а також застосування механізмів автентифікації, авторизації та контролю доступу [68].

Крім того, складність архітектури зростає у разі підтримки кількох фронтенд-застосунків, орієнтованих на різні категорії користувачів або партнерів. У таких умовах виникає необхідність динамічного формування облікових даних і адаптації публічного API до контексту кожного клієнта, що зумовлює додаткове навантаження на процеси адміністрування та потребує централізованого підходу до керування конфігураціями та доступом [64].

У типовому підході (як зображено на рисунку 2.6) API-шлюз виконує функцію не лише проксі, а й механізму авторизації, логування, фільтрації запитів та обробки помилок. Він забезпечує логічний розподіл відповідальності між зоною клієнта (фронтенд) та зоною обробки (бекенд), дозволяючи впроваджувати політики безпеки, обмеження за ролями та контроль доступу на рівні запитів [65].

Таким чином, ефективне розділення API на публічне та приватне – це не лише про архітектурне проектування, а про впровадження чіткої моделі безпеки, масштабованості та обслуговуваності, що має вирішальне значення для стійкості та розвитку вебсистем.

2.3.2 Обмеження у використанні конфігураційних файлів та генерації клієнтського коду

У сучасних інструментах розробки вебзастосунків дедалі частіше застосовуються підходи на основі конфігурацій – опис API, політик доступу, операцій запитів та генерації SDK-файлів здійснюється у вигляді декларативних файлів. Такий підхід використовується у фреймворках на зразок WunderGraph, GraphQL Code Generator, OpenAPI Generator та інших. Його перевагами є швидкий старт, типобезпека, автоматизація рутинних процесів та підвищення узгодженості між клієнтською і серверною частинами. Втім, із зростанням складності проєкту конфігураційно-орієнтована розробка породжує низку структурних та організаційних обмежень [66].

Однією з поширених проблем є розпорошення логіки по конфігураційних файлах, зокрема в багатофронтендних середовищах. Коли декілька клієнтських застосунків використовують спільні API, але з різними наборами доступних операцій чи моделей даних, конфігурації стають об'ємними й важкими для супроводу. Часто виникає дублювання запитів, схожих фрагментів або типів – для різних сценаріїв чи клієнтів доводиться створювати окремі конфігурації, навіть якщо відмінності між ними мінімальні. Це збільшує ризик розсинхронізації між конфігураційною логікою та фактичною поведінкою системи [67, 68].

У системах, які швидко еволюціонують, підтримка конфігураційної узгодженості стає серйозним викликом. Будь-яка зміна у внутрішньому API, наприклад, додавання або перейменування поля, потребує оновлення конфігурацій, повторної генерації коду, перевірки сумісності та тестування змін на всіх рівнях. За

відсутності централізованої системи керування контрактами це призводить до ланцюгової реакції оновлень у багатьох модулях, репозиторіях або середовищах. У великих командах або проектах із частими релізами така ситуація є джерелом регулярних затримок, помилок і зростання вартості обслуговування [69, 70].

Конфігураційні файли добре підходять для опису структурованих, повторюваних і типових сценаріїв. Проте вони погано масштабуються до складної бізнес-логіки, яка вимагає умов, динамічного контексту, виклику сторонніх сервісів або адаптації до ролей користувача. У таких випадках розробники змушені виходити за межі декларативного опису і реалізовувати додаткові шари логіки вручну – у вигляді посередників, хуків, або адаптерів, що зменшує переваги централізованого опису і знову вносить фрагментацію. У результаті частина логіки залишається у конфігурації, а частина – в імперативному коді, що ускладнює налагодження, тестування й документацію [71].

У багатьох випадках різні клієнти працюють із різними версіями API або потребують варіативних відповідей (наприклад, адаптованих під свою модель даних). У таких ситуаціях генерація клієнтського коду з однієї спільної конфігурації стає обмеженням [72]. Підтримка декількох версій API вимагає дублювання конфігурацій або запуску окремих генераторів для кожної версії, що збільшує складність CI/CD-процесів, затримує релізи й ускладнює контроль якості. Також часто виникає потреба в розділенні SDK для внутрішнього і зовнішнього використання, що не завжди передбачено типовими інструментами генерації [73].

Більшість систем автогенерації коду працюють на основі шаблонів, які не завжди адаптовані під конкретні бізнес-вимоги або інженерні практики. Часто виникає необхідність кастомізації шаблонів, що тягне за собою залежність від структури генератора, знання інтерфейсів інструмента і складну інтеграцію з CI/CD. Відсутність або нестабільність підтримки певних мов, фреймворків або нових версій API також створює ризики для проекту [74].

Попри явні переваги конфігураційно-орієнтованих підходів і генерації клієнтського коду для зниження рутини, у складних масштабованих системах ці рішення мають суттєві обмеження. Вони добре працюють на ранніх етапах або в

однорідному середовищі, але з часом призводять до зростання складності, дублювання логіки, фрагментації конфігурацій та втрати гнучкості у реакції на бізнес-зміни. Для подолання цих викликів необхідне впровадження гібридних підходів: поєднання декларативного опису з модульною архітектурою, чітке розмежування відповідальності між рівнями API [75].

2.3.3 Фабрика провайдерів API як рівень абстракції взаємодії з мікросервісами

Фабрики провайдерів API реалізуються з метою уніфікації доступу до мікросервісних API в межах BFF-архітектури. Як приклад, розглянемо два сервіси: сервіс авторизації, що взаємодіє через GraphQL API, та сервіс роботи з профілем користувача, що використовує REST-підхід. Для кожного з них розробляється окрема фабрика провайдерів API [76].

Для інтеграції з сервісом авторизації доцільно застосувати бібліотеку GraphQL Code Generator, яка забезпечує автоматичну генерацію типізованого клієнтського коду на основі заданої схеми та запитів. Основні етапи створення провайдера для GraphQL-сервісу включають:

- формування конфігураційного файлу генератора;
- опис запитів та мутацій у відповідних файлах;
- виконання процесу генерації API та супутніх типів.

Приклад конфігурації для генерації GraphQL-запитів і мутацій подано в лістингу 2.1. У ньому визначаються шляхи до описів GraphQL-операцій, адреса схеми, а також параметри плагінів для генерації коду. Також задається вихідна директорія для сформованого пакета.

Після налаштування схеми генерації та запуску генератора формується SDK, що реалізує метод `getSdk`. Ця функція повертає об'єкт з функціями-обгортками для взаємодії з описаними GraphQL-запитами та мутаціями. Усі функції є

типізованими, а згенеровані інтерфейси можуть бути використані у клієнтському коді для підвищення типобезпеки та зручності розробки [77].

Іншим прикладом є створення фабрики для REST-сервісу управління профілем користувача. REST (перенесення репрезентації стану) визначає набір принципів і обмежень для побудови масштабованих розподілених систем. Основними характеристиками RESTful API є:

- ресурсно-орієнтована модель, де об'єкти системи мають унікальні ідентифікатори (URL);
- використання HTTP-методів (GET, POST, PUT, DELETE тощо);
- статусна незалежність запитів (сервер не зберігає контекст клієнта);
- єдиний інтерфейс для взаємодії.

```
const config = {
  schema: {
    [schemaUrl]: {
    },
    documents: './operations/**/*.graphql',
    generates: {
      [path.join(__dirname, 'src/index.ts')]: {
        plugins: [
          'typescript',
          'typescript-operations',
          'typescript-graphql-request'
        ],
      },
    },
    hooks: false
  }
}

const output = await cli.generate(config)
output.forEach(({filename, content}) => {
  fs.writeFileSync(filename, content)
})
```

Лістинг 2.1 – Налаштування генерації GraphQL API служби авторизації

Для опису REST API використовується стандарт OpenAPI (раніше Swagger). Однією з ключових переваг цього підходу є можливість декларативної генерації типізованого клієнтського коду, що полегшує інтеграцію та забезпечує єдність контрактів між клієнтом і сервером [78].

Процес створення REST-провайдера включає:

- опис API у YAML-файлі відповідно до специфікації OpenAPI;
- налаштування генератора коду;
- формування вихідного клієнтського пакета.

YAML-файл містить обов’язкові розділи: версію OpenAPI, мета інформацію, теги, маршрути, компоненти (схеми типів). У лістингу 2.2 наведено приклад конфігурації, де можна побачити структуру опису запиту, відповідного ідентифікатора операції «operationId», формату аргументів і відповіді, зокрема через посилання «\$ref» на схеми, як «#/components/schemas/ProfilePayload».

```
openapi: 3.0.0
info:
  version: '1.0.0'
  title: Profile Frontend API
tags:
  - name: Profile
paths:
  /profile/user:
    patch:
      tags:
        - Profile
      operationId: updateProfile
      requestBody:
        content:
          application/json:
            schema:
              $ref: '#/components/schemas/ProfilePayload'
components:
  schemas:
```

```

ProfilePayload:
  type: object
  properties:
    first_name:
      type: string
    last_name:
      type: string

```

Лістинг 2.2 – Приклад опису у форматі YAML для OpenAPI

Генерація коду здійснюється за допомогою інструмента OpenAPI Generator, який дозволяє створити повноцінний клієнт на базі описаної специфікації. Автоматизований процес може бути організований за допомогою скрипта, що виконує:

- генерацію клієнтського коду;
- копіювання згенерованого API до пакета;
- створення фінальної збірки для використання в проєкті.

Результатом є набір класів, імена яких відповідають тегам у YAML-файлі. Кожен з них є нащадком базового класу «BaseAPI», що дозволяє передавати конфігурацію (заголовки, базовий URL тощо). Методи класів відповідають операціям, визначеним через «operationId». Виклики здійснюються через HTTP-клієнт Axios, з можливістю розширення, наприклад, створення інтерцептор-функцій для обробки відповідей API [79].

Розроблені фабрики провайдерів – як для GraphQL, так і для REST – мають уніфікований інтерфейс доступу. У випадку GraphQL використовується функція «getSdk», яка повертає об’єкт функцій-обгортки, тоді як REST-версія реалізується у вигляді класів з методами для відповідних запитів. Такий підхід забезпечує єдність способу взаємодії з API, спрощуючи підтримку коду, інтеграцію у фронтенд-застосунок і масштабування системи [80].

2.3.4 API-шлюз як єдина точка маршрутизації та контролю доступу

У рамках реалізації BFF-підходу впроваджується API-шлюз як єдина точка входу для взаємодії з мікросервісами. У межах запропонованої архітектури фабрики провайдерів API використовуються як внутрішній механізм API-шлюзу на стороні фронтенд-сервера (BFF-шару). Для налаштування API-шлюзу необхідно реалізувати низку технічних вимог:

- створити екземпляри клієнтів на основі фабрик провайдерів API;
- реалізувати REST API, що прийматиме запити від фронтенд-клієнта та делегуватиме їх до відповідних мікросервісів.

Для взаємодії з сервісом авторизації, який працює через GraphQL API, необхідно імпортувати відповідну фабрику провайдерів, яка надає функцію «getSdk». Ця функція створює об'єкт з функціями-обгортками для описаних запитів і мутацій. У лістингу 2.3 наведено приклад функції «getAuthApi», яка виконує налаштування GraphQL-клієнта:

- встановлюється URL-адреса GraphQL-сервісу;
- конфігуруються заголовки запиту (зокрема токени автентифікації);
- генерується SDK-клієнт, який інкапсулює виклики до GraphQL API.

```
export const getAuthApi = () => {
  const cookieStore = cookies()
  const authToken = cookieStore.get('accessToken')?.value
  const client = new GraphQLClient(
    process.env.AUTH_API,
    setConfigApi({ authToken }) ,
  )
  return authApiSDK(client)}
```

Лістинг 2.3 – Налаштування клієнта для роботи з API авторизації

Аналогічно, у випадку взаємодії з REST-сервісом для управління профілем користувача, реалізується клієнт, приклад якого наведено у лістингу 2.4. Функція «getProfileApi» створює інстанс класу «ProfileApi», який містить методи, що відповідають REST-ендпоінтам, описаним у специфікації OpenAPI. Налаштування також включає передачу заголовків автентифікації та конфігурацію URL адреси API. [81]

У кожному з описаних підходів використовується єдиний інтерфейс виклику: виклик функції ініціалізації («getAuthApi» або «getProfileApi»), що повертає об'єкт з методами доступу до API відповідного мікросервісу. Така уніфікація дозволяє абстрагуватися від конкретної технології API (REST або GraphQL) і забезпечити узгоджене використання провайдерів на рівні API-шлюзу [82].

Хоча GraphQL і REST мають різні парадигми взаємодії, у контексті BFF-підходу вони можуть оброблятися єдиним маршрутизатором завдяки схожості у транспортному рівні (HTTP) та чітко описаним контрактам API. Наприклад, GraphQL використовує стандартні POST-запити з текстовим запитом у тілі, що дозволяє легко інтегрувати його обробку до загального шлюзу через єдиний контролер або маршрут [83].

```
export const getProfileApi = () => {
  const cookieStore = cookies()
  const authToken = cookieStore.get('accessToken')?.value
  return new ProfileApi(setAPIConfiguration({ authToken }))
}
```

Лістинг 2.4 – Налаштування клієнта для роботи з API профілю користувача

REST API, у свою чергу, може реалізовуватися на різних технологічних платформах – таких як серверна частина Next.js, NestJS, Laravel або інші фреймворки. У будь-якому випадку реалізація API-шлюзу вимагає:

- створення маршрутів, які приймають запити від фронтенд-клієнта;

- делегування обробки до відповідного клієнта, створеного на базі фабрики провайдерів;
- додаткової логіки обробки помилок, трансформації даних, кешування або валідації.

Таким чином, API-шлюз є ключовим елементом інтеграційного шару, який інкапсулює виклики до мікросервісів, забезпечує гнучкість у підключенні нових сервісів та дозволяє централізовано реалізовувати політики доступу, кешування, валідації та моніторингу запитів.

Висновки до розділу 2

Розглянуто проблематику інтеграції даних та побудови API-рівня у мікросервісній архітектурі високонавантажених вебсистем. Збільшення кількості сервісів, різноманітність протоколів та необхідність формування індивідуальних варіантів доступу для різних фронтенд-застосунків створюють суттєві технічні виклики для традиційних моделей побудови інтеграційного шару.

Аналіз сучасних інструментів таких як: WunderGraph, Apollo Federation та GraphQL Mesh дав змогу виявити низку структурних обмежень, що ускладнюють їх застосування у мультибрендових системах. WunderGraph демонструє залежність від централізованої конфігурації віртуальних графів і створює труднощі при масштабуванні у випадку використання багатьох фронтендів. Apollo Federation обмежений прив'язкою до GraphQL-сервісів, не підтримує модульної локалізації змін та не забезпечує гнучкого формування API-профілів для різних клієнтів. GraphQL Mesh, хоча й уніфікує доступ до різноманітних API, базується на централізованому інтеграційному шарі, що перешкоджає гнучкому масштабуванню та ізольованій еволюції окремих доменів.

Виявлені недоліки обґрунтовують доцільність використання BFF-орієнтованої архітектури, здатної забезпечити ізоляцію логіки доступу, модульність інтеграційних компонентів та адаптацію API під потреби конкретних

клієнтських застосунків. У межах дослідження запропоновано модель, що ґрунтується на застосуванні Фабрики Провайдерів API та централізованого API-шлюзу. Фабрика Провайдерів API забезпечує модульне проєктування адаптерів для окремих сервісів, локалізацію змін у межах відповідного провайдера та можливість незалежної еволюції інтеграційних модулів. API-шлюз виконує функції маршрутизації, нормалізації, застосування політик доступу та формування відмінних API-профілів для різних типів клієнтів. Підтримка як GraphQL-, так і REST-протоколів дозволяє працювати з неоднорідними мікросервісами без потреби у їх примусовій уніфікації.

Запропонований підхід дає змогу усунути дублювання логіки на фронтенді, скоротити обсяг шаблонного коду, підвищити керованість та масштабованість системи, а також зменшити залежність від централізованих конфігурацій. Модульність фабрики провайдерів і можливість формування індивідуальних API-профілів роблять архітектуру стійкою до змін та придатною до експлуатації у високонавантажених вебплатформах.

Хоча впровадження запропонованої архітектури потребує додаткового проєктування та підготовки команди, її переваги в аспектах масштабованості, адаптивності та структурної гнучкості підтверджують доцільність використання моделі як основи для побудови сучасних інтеграційних рішень у складних вебсистемах.

3 ФОРМАЛІЗАЦІЯ ДЕКЛАРАТИВНОГО ПІДХОДУ ДО ІНВАЛІДАЦІЇ КЕШУ У МІКРОСЕРВІСНИХ СИСТЕМАХ

Інвалідація кешу є критичним компонентом забезпечення узгодженості даних у мікросервісних архітектурах, де численні сервіси взаємодіють через спільні точки доступу. У високонавантажених вебсистемах, особливо у B2B-сегменті, традиційні імперативні підходи до керування кешем призводять до надмірної складності, зростання кількості помилок та труднощів масштабування. Водночас декларативний підхід, що базується на декларативному описі інвалідації замість ручного керування кешем, відкриває нові можливості для підвищення надійності, повторного використання та автоматизації кеш-стратегії.

У даному розділі досліджено та запропоновано формалізовану декларативний підхід інвалідації, який дозволяє зменшити когнітивне навантаження на розробників, мінімізувати затримки синхронізації та покращити контроль над актуальністю даних у розподілених системах.

3.1 Проблематика інвалідації кешу у високонавантажених мікросервісних архітектурах

Зі зростанням складності мікросервісних вебдодатків постає все більше практичних викликів, пов'язаних з інвалідацією кешу – особливо у випадках, коли дані, змінені в одному сервісі, а використовуються або кешуються в інших контекстах. Така ситуація часто призводить до появи застарілих або неконсистентних даних на інтерфейсі, особливо в умовах високого навантаження та паралельної обробки великої кількості запитів.

У класичному підході розробники змушені явно вказувати, коли, де і як має очищатися кеш, що породжує високу щільність бізнес-логіки, важкість тестування і труднощі в підтримці. Це особливо ускладнюється, коли оновлення даних мають

непрямий вплив на інші сутності або агреговані представлення – наприклад, зміна профілю користувача повинна інвалідовувати також кеш списку користувачів, сторінку адміністратора та панель сповіщень [23].

Окремою проблемою є те, що у більшості архітектур відсутня централізована система опису залежностей між кешованими об'єктами та джерелами змін, що унеможливорює автоматичну або подієво-орієнтовану інвалідацію без ризику помилкових рішень. Як результат – системи або страждають від надмірної інвалідації (що знижує ефективність кешування), або, навпаки, допускають витoki застарілих даних. Наявні підходи вимагають переосмислення в бік декларативних та формалізованих рішень, де зв'язки між сутностями, умови оновлення та сфери впливу інвалідації визначаються на рівні конфігурації, а не розпорошеної логіки у коді [28].

3.1.1 Огляд популярних кеш-систем

Серед найпопулярніших і найбільш поширених кеш-систем виділяють такі рішення, як Memcached, Redis і Hazelcast. Кеш являє собою сховище «ключ-значення», що використовується для тимчасового зберігання даних з метою прискорення доступу до них. Зазвичай дані в кеші зберігаються в оперативній пам'яті (RAM), що забезпечує швидкий доступ, хоча, залежно від реалізації, кеш може також використовувати інші рівні зберігання, включаючи дискові системи. Це дозволяє балансувати між швидкістю доступу та обсягом збережених даних [84, 86].

Сьогодні існують численні комерційні рішення, що надають готові сервіси кешування, зокрема хмарні платформи (PaaS). Серед них Amazon ElastiCache, Google Memorystore та інші, які дозволяють організувати кешування у хмарному середовищі, оптимізуючі інфраструктурні витрати та забезпечуючи масштабованість системи [85, 86].

Перш ніж заглибитися у дослідження механізму інвалідації кешу, розглянемо детальніше системи кешування даних.

Redis (Remote Dictionary Server) – це система управління базами даних типу «ключ-значення», яка зберігає дані в оперативній пам'яті, що забезпечує високу швидкість доступу. Завдяки такому підходу Redis є одним із найбільш ефективних рішень для кешування у масштабованих вебсистемах [87]. Redis підтримує кілька складних типів даних, що дозволяє використовувати його у різноманітних сценаріях:

- рядки «базовий тип даних, який використовується для зберігання тексту або чисел»;
- хеші «структури, що дозволяють зберігати пов'язані дані у вигляді наборів «поле-значення»;
- списки «упорядковані колекції, які забезпечують ефективне додавання та видалення елементів на початку або кінці»;
- множини «неупорядковані колекції унікальних елементів, що оптимально підходять для перевірки наявності або відсутності значень»;
- впорядковані множини «колекції, в яких кожен елемент має пов'язаний з ним рейтинг, що дозволяє сортувати елементи»;
- бітові масиви «бітові структури, що дозволяють ефективно зберігати і працювати з бінарними даними»;
- гіперлоглоги «структури для оцінки кількості унікальних елементів у великих наборах даних»;
- підписки/публікації «механізм, що підтримує асинхронну обробку подій і дозволяє створювати систему обміну повідомленнями».

Завдяки зберіганню даних у пам'яті Redis забезпечує надшвидкий доступ до даних, що робить його ідеальним для кешування. Redis підтримує декілька механізмів персистентності, що дозволяє зберігати дані на диску. Для забезпечення збереження даних можуть використовуватися механізми RDB (знімки стану даних на диску) або AOF (журналювання змін у форматі Append Only File). Redis може бути розгорнутий у кластерному режимі, що дозволяє масштабувати систему за

допомогою горизонтального масштабування. Redis може використовуватися як кеш, брокер повідомлень, система черг і навіть база даних [88].

Redis надає API, доступний через кілька клієнтів для різних мов програмування. Основний API Redis заснований на командному рядку, але доступний і через бібліотеки [88, 89].

Redis використовується в багатьох сферах, зокрема для кешування результатів запитів до бази даних, зберігання сесій користувачів, реалізації черг для обробки завдань, зберігання лічильників і статистики в реальному часі.

Однак Redis має обмеження, пов'язані з обсягом доступної оперативної пам'яті, що може ускладнювати його використання для великих наборів даних або тривалого зберігання інформації. Також, незважаючи на підтримку механізмів персистентності (RDB та AOF), варто враховувати, що вони можуть вплинути на продуктивність або ускладнити конфігурацію, особливо у випадках, коли потрібна висока частота оновлень даних [88].

Memcached – популярна система для реалізації кешування, що зберігає дані в оперативній пам'яті з метою підвищення продуктивності вебдодатків. Завдяки розподіленій архітектурі Memcached дозволяє зберігати дані у форматі «ключ-значення», що робить його особливо ефективним для задач, де потрібен швидкий доступ до даних без необхідності складної структури збереження [91].

Особливістю Memcached є автоматичне видалення старих або неактивних ключів, що забезпечує більш раціональне використання пам'яті. Проте, як система кешування без персистентності, Memcached не зберігає дані на диску, а тому втрачає їх після перезавантаження або відключення сервера. Це робить його ідеальним для задач, де критичне значення має швидкість доступу до даних, але допускається можливість втрати кешованих значень у разі збою [92].

До сильних сторін Memcached можна віднести:

- швидкість (Memcached забезпечує дуже швидкий доступ до даних, що дозволяє зменшити затримки у веб-додатках);
- простота та легкість налаштування (Memcached має простий інтерфейс і легко налаштовується, що робить його зручним для використання);

- розподілена архітектура (можливість горизонтального масштабування дозволяє обробляти великі обсяги запитів без зниження продуктивності);
- крос-платформенність (Memcached може бути використаний з багатьма мовами програмування і платформами).

В той же час Memcached має наступні слабкі сторони:

- відсутність персистентності (Memcached не зберігає дані на диску, тому при перезавантаженні сервера всі дані будуть втрачені);
- обмеження типів даних (Memcached підтримує лише прості типи даних (рядки), що обмежує його використання в порівнянні з Redis);
- обмежений обсяг пам'яті (Memcached має обмеження на обсяг пам'яті, що використовується для кешування, що може бути проблемою для великих наборів даних);
- кешування за ключем (якщо ключ не існує в кеші, запит до бази даних або іншого джерела даних є обов'язковим, що може призвести до зростання навантаження на систему).

Memcached є потужним і простим у використанні інструментом для кешування, особливо в середовищах, де швидкість є критично важливою. Його простота та ефективність роблять його популярним вибором для багатьох вебдодатків. Проте його обмеження в персистентності та обмежена підтримка типів даних можуть стати вирішальними факторами при виборі технології для конкретного проекту, особливо якщо потрібна більш складна обробка даних [90].

Hazelcast є платформою розподіленого кешування та обробки даних, що реалізує концепцію розподіленої сітки даних в оперативній пам'яті (IMDG). Hazelcast розроблено для роботи з великими обсягами даних і забезпечує високу продуктивність, масштабованість та доступність [93].

Основні характеристики Hazelcast:

- розподілена сітка даних в оперативній пам'яті (Hazelcast зберігає дані в пам'яті, що забезпечує швидкий доступ і знижує затримки при обробці запитів);

- розподіленість (Hazelcast підтримує горизонтальне масштабування, що дозволяє додавати нові вузли до кластеру без зупинки системи. Це робить його ідеальним для роботи в середовищах з високими вимогами до продуктивності);
- динамічне балансування навантаження (платформа автоматично розподіляє дані між вузлами, забезпечуючи рівномірне навантаження);
- підтримка різних типів даних (Hazelcast підтримує різноманітні структури даних, включаючи мапи (Maps), множини (Sets), списки (Lists) та черги (Queues));
- сховища даних (Hazelcast може використовуватися як кеш, а також для зберігання даних у реальному часі. Крім того, підтримує зовнішні бази даних, інтегруючись з ними);
- підтримка транзакцій (Hazelcast підтримує транзакційні операції для забезпечення узгодженості даних при виконанні кількох змін);
- підтримка Java EE (Hazelcast легко інтегрується з Java EE додатками та іншими популярними фреймворками) [94].

Hazelcast надає API для роботи з даними та керування кластером. Основний клас для взаємодії з кластером – це Hazelcast Instance, через який додатки отримують доступ до даних. Для керування даними використовуються інтерфейси, такі як `IMap<K, V>` для роботи з мапами, що підтримують розподілене кешування, `ISet<E>` для роботи з множинами та `IQueue<E>` для роботи з чергами [92].

Hazelcast також надає можливість працювати з розподіленими структурами даних, такими як `IAtomicLong`, `ICountDownLatch`, `ISemaphore` та інші. Для реалізації публікаційно-підписувальних систем існує інтерфейс `ITopic<E>`. Крім того, Hazelcast забезпечує API для моніторингу та управління кластером, включаючи можливість додавати та видаляти вузли [93].

До сильних сторін Hazelcast можна віднести наступне:

- висока продуктивність (зберігання даних в пам'яті забезпечує швидкий доступ і високу продуктивність при обробці запитів);
- масштабованість (Hazelcast легко масштабується горизонтально, що дозволяє системі рости в міру зростання навантаження);

- гнучкість (підтримка різноманітних структур даних і інтеграція з різними джерелами даних роблять Hazelcast універсальним інструментом);
- крос-платформенність (Hazelcast може використовуватися з різними мовами програмування та платформами, що робить його гнучким у впровадженні).

Окрім сильних сторін Hazelcast має декілька суттєвих мінусів, що впливають на вибір використання даної системи:

- витрати на пам'ять (зберігання даних в оперативній пам'яті може призвести до великих витрат на пам'ять, якщо обсяги даних значні);
- вартість (хоча Hazelcast має безкоштовну версію, платна версія може бути досить дорогою для великих підприємств);
- залежність від Java (Hazelcast побудований на Java, тому оптимально працює в середовищах Java. Це може бути обмеженням для проектів на інших мовах) [94].

Незважаючи на численні переваги Hazelcast, зокрема підтримку розподілених структур даних і високу продуктивність, його застосування може бути менш ефективним порівняно з Redis і Memcached у певних сценаріях через підвищену складність, значні вимоги до пам'яті та залежність від Java. Вибір між цими технологіями визначається специфічними потребами проекту та особливостями інфраструктури [95].

3.1.2 Проблематика інвалідації кешу

Під час обробки запиту система насамперед виконує перевірку наявності відповідних даних у кеші. Якщо ключ наявний у кеші, значення повертається негайно, оминаючи звернення до бази даних. У випадку кеш-промаху (cache miss) виконується запит до основного джерела даних (наприклад, реляційної БД), після чого отримані дані зберігаються в кеші для подальшого використання. Такий підхід реалізує патерн кешування з боку застосунку, який є де-факто стандартом у більшості сучасних систем [96].

Для управління життєвим циклом кешованих записів застосовуються часові мітки, які використовуються для визначення актуальності даних. Один із підходів – визначення фіксованого часу життя (TTL), після завершення якого значення вважається застарілим. Альтернативно може застосовуватися регулярне сканування кешу (наприклад, через системний планувальник періодичного виконання завдань (CRON)) з перевіркою відповідності між поточною датою та міткою останнього оновлення. Обидва підходи мають обмеження: TTL не гарантує миттєвого оновлення після зміни базових даних, а повний обхід ключів суттєво навантажує систему, особливо при великій кількості записів [97].

У контексті MVC-архітектури кеш зазвичай інтегрується на рівні моделі, яка відповідає за доступ до джерел даних (ORM, API тощо). Розробники змушені явно визначати, які запити слід кешувати, а які – оновлювати або виключати з кешу. Це ускладнює підтримку системи та створює ризики втрати актуальності даних [19].

Подібна ситуація спостерігається у фронтенд-архітектурах, де використовуються бібліотеки на кшталт React, Angular або VueJS. У таких додатках впроваджуються локальні механізми кешування – наприклад, через Apollo Client у GraphQL-застосунках. Розробникам необхідно явно визначати політики оновлення кешу (наприклад, стратегія пріоритетного доступу до кешу, кеш-орієнтована стратегія отримання даних), що створює потенційний конфлікт між продуктивністю та достовірністю даних [19].

Особливої складності набуває ситуація, коли логіка кешування інтегрується безпосередньо в бізнес-логіку. Це призводить до високого ступеня зчеплення, що суперечить принципам побудови масштабованих і тестованих систем. Зміни у правилах кешування вимагають перегляду бізнес-коду, що знижує гнучкість і підвищує ризик регресій [95].

Найбільш поширена причина, з якої розробники реалізують механізми інвалідації кешу – необхідність запобігання роботі з неактуальними даними. Застарілий кеш може стати джерелом критичних помилок, особливо в фінансових та транзакційних системах [96].

Як альтернативу імперативному підходу, Microsoft пропонує застосовувати NCache як просунуте рішення з підтримкою синхронізації з базами даних. У цьому випадку зміни у базі автоматично викликають оновлення відповідних записів у кеші. Таким чином, кеш стає реактивним компонентом, а не лише тимчасовим сховищем. Цей підхід знижує навантаження на кодову базу та відповідає п'ятому принципу SOLID – принципу інверсії залежностей, згідно з яким модулі повинні залежати від абстракцій, а не від конкретних реалізацій [97].

3.1.3 Огляд реалізації подієвої інвалідації кешу на прикладі Ncache

Одним із практичних рішень, що реалізує декларативний підхід до інвалідації кешу, є NCache – розподілена in-memory платформа з підтримкою публікаційно-підписного обміну повідомленнями, SQL-пошуку та інтеграції з базами даних у реальному часі.

Серед основних функціональних можливостей NCache виокремлюють:

- підтримку патерну публікація/підписка (pub/sub), який дозволяє сервісам реагувати на події без прямої залежності від джерела змін;
- кластеризацію кешу, що забезпечує горизонтальну масштабованість і відмовостійкість;
- пошук та агрегацію за допомогою розширень SQL, що спрощує обробку великих обсягів кешованих даних;
- синхронізацію кешу з базами даних, що дозволяє уникнути роботи з неактуальними значеннями [18].

Механізм публікації/підписки (pub/sub) в NCache реалізовано як брокер подій: мікросервіси, що публікують зміни (видавці), не мають прямого знання про підписників. Повідомлення класифікуються за темами або типами даних, а споживачі отримують лише ті події, на які підписані. Це дозволяє досягти слабкого зв'язування компонентів і підвищує гнучкість розгортання [18].

Особливе значення має можливість автоматичної синхронізації кешу з базою даних. NCache дозволяє налаштувати залежності між даними в кеші та об'єктами в БД: у разі зміни останніх система ініціює оновлення відповідних кешованих записів. Процедура включає:

- створення збереженої процедури або тригера в базі даних для відстеження змін;
- генерацію унікального ідентифікатора (ключа) елемента кешу;
- публікацію події про зміну або інвалідацію;
- реактивне оновлення або видалення відповідного об'єкта з кешу [22].

Даний підхід дозволяє досягти актуальності кешу з мінімальним втручанням у бізнес-логіку, що відповідає принципам декларативного управління. Проте така інтеграція має обмеження: сильна залежність від конкретного технологічного стеку, зокрема специфічних функцій NCache, SQL Server або API. У випадку міграції на Redis чи Memcached, що не мають аналогічної вбудованої підтримки подій і синхронізації, подібні функції доведеться реалізовувати вручну або через сторонні сервіси (наприклад, Kafka + Redis Streams) [87].

Отже, NCache ілюструє потенціал декларативної моделі кешування, зокрема в контексті мікросервісної архітектури з високими вимогами до узгодженості даних. Однак для досягнення повної гнучкості доцільним є проектування абстракцій, що дозволяють змінювати бекенд кешу без змін у бізнес-логіці застосунку.

3.1.4 Декларативний підхід як відповідь на сучасні виклики

На тлі зростаючих вимог до продуктивності, масштабованості та узгодженості мікросервісних систем дедалі більшої актуальності набуває декларативний підхід до інвалідації кешу, який ґрунтується на концепції розділення відповідальностей між бізнес-логікою і логікою керування кешем. Основна ідея цього підходу полягає у тому, що правила кешування та його

інвалідації визначаються на рівні конфігурацій або декларативних описів, а не реалізуються безпосередньо у коді додатка [19].

Такий підхід дозволяє досягти високого ступеня незалежності бізнес-логіки мікросервісів від внутрішніх механізмів зберігання та оновлення кешу. Це, у свою чергу, спрощує тестування, полегшує внесення змін до кеш-стратегій без необхідності модифікації прикладного коду, та знижує ймовірність людських помилок при реалізації логіки інвалідації [18].

Однією з ключових переваг декларативного підходу є можливість автоматичного керування залежностями між мікросервісами. У системах із складними графами взаємодії між сервісами оновлення однієї сутності може вимагати інвалідації кешу в кількох пов'язаних компонентах. Замість ручного опису цих залежностей декларативна модель дозволяє використовувати семантичні правила, які описують, які саме типи змін мають спричинити інвалідацію певного кешованого контенту [32].

Декларативна інвалідація також відкриває можливість уніфікованого конфігурування кешу на рівні всієї системи. Це означає, що незалежно від реалізації окремих мікросервісів, усі вони дотримуються спільних правил керування кешем, що значно спрощує підтримку й масштабування таких систем. В результаті відпадає потреба у централізованому сервері інвалідації, оскільки правила можуть бути розподілені або синхронізовані через конфігураційні сховища чи системи управління політиками [33].

Ще однією важливою перевагою є гнучкість адаптації до змін у робочому навантаженні. Зокрема, декларативний підхід дозволяє динамічно змінювати TTL, залежно від актуальності даних або бізнес-пріоритетів, не змінюючи структуру сервісів. Також з'являється можливість інтегрувати механізми адаптивного кешування з урахуванням частоти запитів, змін у даних чи сезонності бізнес-процесів [34].

З наукової точки зору впровадження декларативного підходу сприяє формалізації процесів кешування у мікросервісних архітектурах. Це відкриває перспективи для подальшої розробки високорівневих мов або DSL (предметно-

орієнтовані мови) для опису політик кешу, а також автоматичних систем верифікації, які можуть перевіряти коректність залежностей та правил інвалідації.

З практичного погляду, запропонований підхід може істотно знизити витрати на підтримку інфраструктури кешування у високонавантажених системах, забезпечити стійкість до помилок, пов'язаних із людським фактором, і підвищити загальну продуктивність. Його застосування є доцільним у сферах, де критично важлива швидкість обробки запитів та консистентність даних – у стримінгових платформах, фінансових технологіях, логістичних системах тощо.

3.2 Архітектурні принципи побудови декларативної моделі інвалідації кешу

Основним архітектурним недоліком традиційних рішень є щільне зв'язування бізнес-логіки з операціями інвалідації, що ускладнює супровід, тестування та повторне використання логіки кешування.

Декларативна модель інвалідації кешу покликана вирішити ці проблеми шляхом відокремлення механізмів очищення кешу від основної логіки мікросервісів. У центрі такого підходу – конфігураційні декларації, які визначають залежності між подіями у системі (наприклад, створення чи оновлення сутностей) та відповідними кешованими представленнями, що підлягають інвалідації.

Запропонована модель ґрунтується на подієвій архітектурі, де кожна значуща зміна стану системи породжує подію, яка передається до інвалідаційного механізму. Така декларація включає:

- опис події, яка ініціює інвалідацію;
- перелік залежних представлень або ресурсів, що мають бути очищені;
- параметри, що використовуються для динамічного формування ключів кешу;
- контекст, у якому має бути застосоване правило.
- Архітектурно така модель дозволяє:

- централізовано керувати правилами інвалідації без дублювання логіки у кожному сервісі;
- мінімізувати ризик помилок, пов'язаних із пропущеною або надлишковою інвалідацією;
- зберігати незалежність мікросервісів від кеш-інфраструктури;
- гнучко масштабувати та адаптувати кеш-стратегії до різних сценаріїв навантаження.

3.2.1 Поділ відповідальності між мікросервісами для автономного управління кешем та мінімізації міжсервісних залежностей

Ефективне управління кешем у мікросервісній архітектурі потребує не лише використання відповідних технологій, але й виваженого розподілу відповідальності між компонентами системи. Основним принципом побудови масштабованих і підтримуваних систем є автономність сервісів та мінімізація міжсервісних залежностей. Це передбачає винесення кеш-логіки за межі бізнес-сервісів та впровадження окремих компонентів, відповідальних за обробку кешованих даних [97].

На рис. 3.1 представлено архітектурну модель системи інвалідації кешу, яка демонструє структурований підхід до взаємодії між клієнтом, API-шлюзом, сервісом бізнес-логіки, сервісом кешування та кеш-сховищем. Така модель дозволяє забезпечити узгодженість даних, гнучке масштабування і високу керованість процесом інвалідації у розподіленому середовищі.

Компоненти архітектури:

- клієнт – ініціатор запитів до системи, який очікує швидкої та актуальної відповіді (саме очікування низької затримки відповіді і зумовлює потребу в ефективному кешуванні);
- API-шлюз виконує роль центрального маршрутизатора запитів, спрямовуючи їх до відповідних мікросервісів. Водночас він є точкою збору схем

інвалідації (тобто структурованих інструкцій, які передаються далі до кеш-сервісу для обробки. Це дозволяє централізовано керувати логікою оновлення кешу, навіть якщо запити обробляються кількома незалежними сервісами);

- сервіс бізнес-логіки реалізує основні функції додатку – обробку запитів, взаємодію з БД чи зовнішніми API та генерацію відповідей (у випадку модифікації даних він також формує схему інвалідації, яка містить перелік кешованих ключів, що потребують очищення);

- сервіс кешування – ключовий компонент, відповідальний за реалізацію операцій інвалідації, оновлення та отримання кешу (виступає абстрактним інтерфейсом між логікою додатку та кеш-сховищем, інтерпретує схеми інвалідації та приймає рішення про зміну стану кешу. Такий підхід дозволяє делегувати складну логіку відповідальній інфраструктурній одиниці, зменшуючи навантаження на мікросервіси та підвищуючи надійність системи);

- кеш-сховище – фізичний рівень архітектури, який реалізується з використанням високошвидкісних in-memory рішень, таких як Redis, Memcached та інші (тут зберігаються ключі кешованих об'єктів і виконуються операції «get», «set», «delete» згідно з інструкціями від кеш-сервісу) [21].

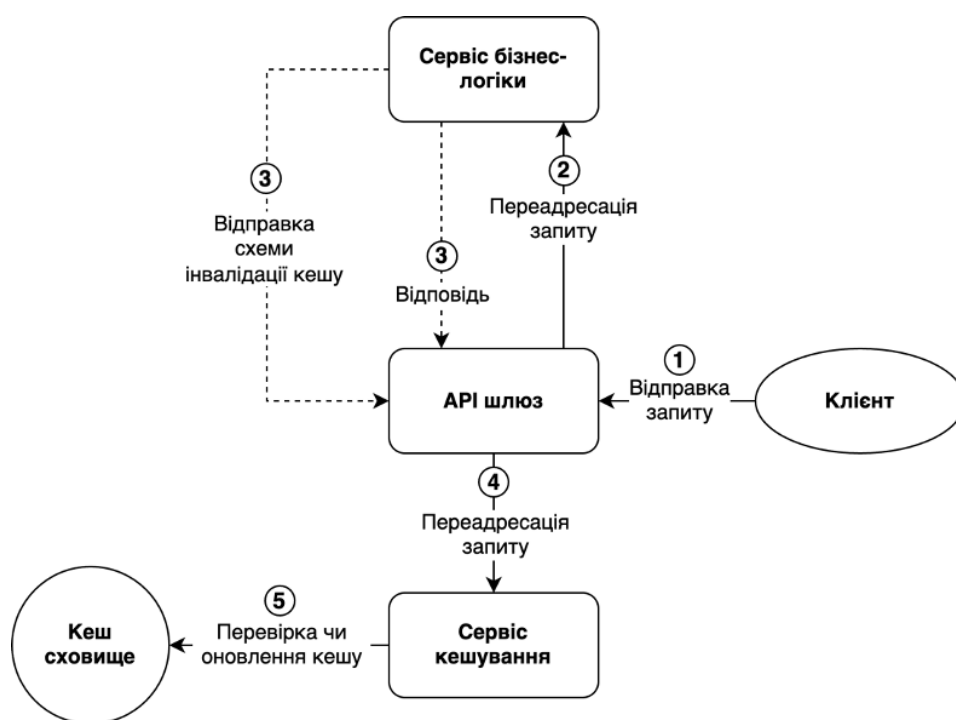


Рисунок 3.1 – Архітектура додатку з системою інвалідації кешу

Такий розподіл дозволяє:

- зменшити щільність логіки у бізнес-сервісах;
- уніфікувати обробку подій інвалідації;
- полегшити підтримку та впровадження нових правил очищення кешу без зміни коду сервісів;
- покращити масштабування, адже компоненти можуть масштабуватись незалежно.

Завдяки даній структурі механізми кешування стають повторюваними, тестованими та централізовано контрольованими, а самі мікросервіси зберігають автономність, що критично важливо в умовах зростання складності системи. Підхід також підвищує безпеку і стабільність – оскільки кеш не залежить від внутрішнього стану окремих сервісів, він підтримує цілісність навіть при частковому відмовленні підсистем.

3.2.2 Концептуальна модель декларативного підходу інвалідації кешу

Одним із ключових підходів до керування кешуванням у високонавантажених вебсистемах є декларативна інвалідація кешу, що ґрунтується на використанні зовнішніх конфігураційних файлів у форматі YAML. Такі файли виступають незалежними декларативними структурами, які описують правила очищення кешу у відповідь на визначені системні події та забезпечують відокремлення бізнес-логіки застосунку від інфраструктурних механізмів кешування.

Як наведено у лістингу 3.1, схема архітектурної інтеграції передбачає реєстрацію сервісу бізнес-логіки в API-шлюзі – централізованому компоненті мікросервісної архітектури, що забезпечує маршрутизацію запитів, агрегацію даних та контроль доступу до внутрішніх сервісів – під унікальним ідентифікатором «BUSINESS_SERVICE».

API-шлюз приймає запити клієнтів, маршрутизує їх до відповідних мікросервісів, а також ініціює виконання інструкцій інвалідації, передаючи їх до сервісу кешування. Для реалізації міжкомпонентного обміну повідомленнями використовується Redis, параметри підключення до якого визначаються через змінні середовища «HOST_REDIS» та «PORT_REDIS».

```
@Module({
    imports: [
        ClientsModule.register([
            {name: 'BUSINESS_SERVICE',
             transport: Transport.REDIS,
             options: {
                 host: HOST_REDIS,
                 port: PORT_REDIS,
             }
        ])],
    controllers: [GatewayController]})
```

Лістинг 3.1 – Реєстрація сервісу бізнес-логіки в модулі API шлюзу

Конфігурація YAML (лістинг 3.2) є декларативною інструкцією, яка пов’язує певні події з кешованими представленнями та правилами формування ключів кешу. Кожна така інструкція визначає, які саме дані потребують інвалідації після певної дії, наприклад, створення або оновлення сутності. У типовому сценарії запити (лістинг 3.3), мікросервіс спочатку перевіряє наявність потрібних даних у кеші. Якщо дані відсутні, виконується звернення до бази даних, після чого результат разом із відповідною інструкцією з YAML-файлу передається до сервісу кешування. У випадку створення нового елементу, мікросервіс формує інструкцію очищення кешованих списків, які можуть містити застарілу інформацію (лістинг 3.4). Сервіс кешування обробляє ці декларації за допомогою методу `runInstructions`, який інтерпретує YAML-конфігурацію та ініціює відповідні операції над кешем: вибір ключів, формування шаблонів, очищення або оновлення даних (лістинг 3.5).

Основні переваги запропонованої моделі:

- повне розділення відповідальності між бізнес-логікою та кеш-інфраструктурою;
- гнучкість і масштабованість кеш-правил без втручання в код сервісів;
- декларативність і повторюваність – інструкції описуються в централізованих конфігураціях, які легко перевірити та оновити.

```
title: 'cache invalidate instructions'
service: 'Business'
operations:
  get:
    actions:
      - operation: "add"
        target: "list"
        key: "item_get"
        action_type: "set"
      - operation: "add"
        target: "slug"
        key: "item_get_one"
        action_type: "set"
  getOne:
    actions:
      - operation: "add"
        target: "slug"
        key: "item_get_one"
        action_type: "set"
  create:
    actions:
      - operation: "add"
        target: "slug"
        key: "item_get_one"
        action_type: "update"
```

Лістинг 3.2 – Конфігурація оновлення кешу для сервісу бізнес-логіки

```

@MessagePattern({ cmd: BUSINESS_EVENTS.ITEMS_GET })
async.getItems({ payload }: PayloadDto<GetItemsDto>):
    Promise<Item[]> {
        const                                cacheConfig
this.businessService.getBusinessConfig();
        const dataFromCache = await firstValueFrom(
            this.cacheServiceClient.send(
                {cmd: CACHE_EVENTS.CACHE_CHECK},
                {key: BUSINESS_EVENTS.ITEMS_GET,payload}));
        if (dataFromCache) return dataFromCache;
        const items = await this.businessService.getItems();
        this.cacheServiceClient
            .send({cmd: CACHE_EVENTS.CACHE_RUN_INSTRUCTIONS},
                {cmd: INSTRUCTION_OPERATIONS.GET,
                    instructions: cacheConfig,payload, data: items})
        return items;}

```

Лістинг 3.3 – Метод обробки запиту на отримання списку елементів

```

@MessagePattern({ cmd: BUSINESS_EVENTS.ITEM_CREATE })
async.createItem({payload }: PayloadDto<Item>):
    Promise<SuccessWithoutDataResponseDto> {
        const cacheConfig = this.businessService.getBusinessConfig();
        const newItem = await
            this.businessService.createItem(payload);
        this.cacheServiceClient.send(
            {cmd: CACHE_EVENTS.CACHE_RUN_INSTRUCTIONS},
            {cmd: INSTRUCTION_OPERATIONS.CREATE,
                instructions: cacheConfig, data: newItem});
        return successEmptyResponse;}

```

Лістинг 3.4. Метод створення нового елементу

```

async runInstructions(data: { cmd: INSTRUCTION_OPERATIONS;
  instructions: any; data: any;
  payload?: Record<string, string | number>;
}): Promise<void> {
  data.instructions?.operations?.[data?.cmd]?.actions?.forEach(
    (operation) => {
      const args = {
        key: operation.key, filters: data.payload, data:
data?.data, actionType: operation.action_type};
      switch (operation) {
        case operation.operation === OPERATION.ADD:
          if (operation.target === TARGET.LIST)
            return this.setItemsToCache(args);
          if (operation.target === TARGET.SLUG) {
            if
(this.isObject(args.data)) this.setItemToCache(args);
            else if (Array.isArray(args.data)) {
              return (async () => {
                for (const item of args.data) {
                  await this.setItemToCache(item); })();
            }
            break;
          case operation.operation === OPERATION.REMOVE:
            if (operation.target === TARGET.SLUG) {
              if(this.isObject(args.data))
return this.removeItemFromCache(args);
              else if (Array.isArray(args.data)) {
                return (async () => {
                  for (const item of args.data) {
                    await
                      this.removeItemFromCache(item);
                }
              }
            }
          }
        }
      }
    }
  );
}

```

Лістинг 3.5 – Обробка інструкцій з оновлення кешу

Таким чином, використання зовнішніх YAML-конфігурацій у поєднанні з подієвою архітектурою забезпечує формалізований та гнучкий механізм керування

інвалідацією кешу, що відповідає вимогам високонавантажених, динамічних мікросервісних вебсистем.

Детальний приклад реалізації запропонованої концепції наведено у додатках А, Б, В, Г.

3.2.3 Оптимізація механізмів додавання, оновлення та видалення елементів кешу для зменшення затримок і прискорення отримання даних

В умовах високої навантаженості мікросервісних систем особливу увагу необхідно приділяти оптимізації операцій роботи з кешем.

У лістингу 3.6 наведено реалізацію алгоритму додавання або оновлення масиву елементів у кеші, що забезпечує ефективне керування записами через унікальні ключі та структуру зворотних зв'язків. Ключовим принципом цього підходу є генерація унікального ключа для кожного окремого елемента та його контексту використання, що дозволяє уникнути конфліктів при повторному кешуванні або оновленні даних.

```
generateKey(baseKey, filters) {
  if (!filters || Object.keys(filters).length === 0) return
baseKey;
  const sortedFilters = Object.entries(filters)
    .sort(([a], [b]) => a.localeCompare(b))
    .map([key, value] => `${key}=${value}`)
    .join(':');
  return `${baseKey}:${sortedFilters}`;
}
async setItemsToCache({
  key, filters = {}, data, actionType
}: ISetItemsToCache): Promise<void> {
  const listKey = this.generateKey(key, filters);
  const list = (await this.getCacheByKey(listKey)) || [];
  switch (actionType) {
```

```

case ACTION_TYPE.SET:
    await this.setCacheByKey(listKey, data);
    break;
case ACTION_TYPE.UPDATE:
    await this.setCacheByKey(listKey, [...list, ...data]);}}

```

Лістинг 3.6 – Алгоритм додавання/оновлення масиву елементів у кеші

Задля забезпечення узгодженості структури застосовується механізм зворотних посилань (reverse mapping). Цей механізм дозволяє системі відстежувати всі контексти, у яких використовується певний елемент (наприклад, категорії, списки, пошукові запити), та оновлювати або видаляти його одночасно у всіх залежних структурах.

Для індивідуального керування елементами використовується функція «setItemToCache» (лістинг 3.7), яка виконує такі дії:

- створює основний ключ списку, у якому знаходиться елемент;
- генерує унікальний ключ самого елемента;
- додає посилання у таблицю зворотних залежностей, що дозволяє пізніше ідентифікувати всі місця, де цей елемент фігурує.

```

async setItemToCache({ key, filters = {}, data, actionType }) {
    const { slug } = data;
    const listKey = this.generateKey(key, filters);
    const slugKey = `${key}:${slug}`;
    const referencesKey = `${key}:references:${slug}`;
    const references = [listKey, slugKey];
    switch (actionType) {
        case ACTION_TYPE.SET:
            await this.setCacheByKey(slugKey, data);
            await this.setCacheByKey(referencesKey, references);
            break;
        case ACTION_TYPE.UPDATE:
            const existingData = await this.getCacheByKey(slugKey);

```

```

const updatedData = { ...existingData, ...data };
await this.setCacheByKey(slugKey, updatedData);
await this.setCacheByKey(referencesKey, references);
break;
default:
  throw new HttpException(
    exceptions.UNSUPPORTED_ACTION_TYPE,
    HttpStatus.UNPROCESSABLE_ENTITY);}

```

Лістинг 3.7. Алгоритм додавання/оновлення елемента кешу

Такий підхід забезпечує високу швидкість оновлення та видалення, навіть у випадках складних структурах або повторного використання елемента.

Алгоритм «removeItemFromCache» (лістинг 3.8) виконує зворотну операцію – видаляє елемент із кешу разом з усіма пов’язаними ключами та записами, гарантуючи, що жоден залишковий слід не залишиться. Це критично важливо для запобігання появі застарілих або неконсистентних представлень на інтерфейсі користувача.

```

async removeItemFromCache({ key, data }) {
  const { slug } = data;
  const slugKey = `${key}:${slug}`;
  const referencesKey = `${key}:references:${slug}`;
  const relatedKeys = await this.getCacheByKey(referencesKey) || [];
  for (const key of relatedKeys) {
    const list = await this.getCacheByKey(key);
    if (Array.isArray(list)) {
      const updatedList = list.filter((book) => book.slug !== slug);
      if (updatedList.length === 0) await this.removeCacheByKey(key);
      else await this.setCacheByKey(key, updatedList);}
    await this.removeCacheByKey(slugKey);}

```

Лістинг 3.8 – Алгоритм видалення/оновлення окремого елемента в кеші

Переваги запропонованого підходу:

- узгоджене оновлення всіх списків та структур, які містять змінений елемент;
- мінімізація кількості операцій очищення – оновлюються лише пов'язані представлення, а не весь кеш цілком;
- зменшення ризику дублювання записів або утворення розірваних зв'язків між сутностями;
- підтримка масштабованості кешу в умовах постійного збільшення кількості контекстів і об'єктів.

Хоча реалізація кешування зі зворотними посиланнями на рівні API-шлюзу призводить до появи додаткових накладних витрат, пов'язаних із обробкою запитів та підтримкою внутрішніх структур даних, ці витрати мають лінійний характер і не є критичними для систем із горизонтальним масштабуванням. Натомість такий підхід забезпечує зменшення латентності на рівні клієнтських запитів і сприяє формуванню узгодженого та передбачуваного стану інтерфейсу.

3.3 Оцінка ефективності та порівняльний аналіз

Запропонована модель декларативного підходу до інвалідації кешу орієнтована на зменшення міжсервісних залежностей, що є важливою передумовою підвищення автономності компонентів мікросервісної системи. У межах цієї моделі кожен мікросервіс відповідає не лише за реалізацію бізнес-логіки, а й за ініціацію оновлення кешу в межах власної зони відповідальності, без необхідності врахування внутрішнього стану інших сервісів. Такий підхід спрощує взаємодію між компонентами та сприяє масштабованості системи.

Концептуальна модель ґрунтується на принципі володіння даними (data ownership), відповідно до якого кожен сервіс самостійно відповідає за актуальність і життєвий цикл сформованих ним бізнес-даних. Як показано на рис. 3.1, інвалідацію кешу ініціює виключно сервіс-джерело даних, що усуває потребу в

централізованій координації. Реєстрація сервісів через API-шлюз (лістинг 3.1) забезпечує уніфікований доступ до механізмів кешування, а винесення правил інвалідації у декларативні конфігурації (лістинг 3.2) дозволяє відокремити аспекти кешування від бізнес-логіки.

Практична реалізація моделі передбачає використання набору спеціалізованих алгоритмів, спрямованих на ефективну взаємодію з кешем у розподіленому середовищі. Алгоритми доступу до списків і створення нових елементів (лістинги 3.3, 3.4) забезпечують оптимізований доступ до кешованих даних з автоматичним оновленням відповідно до визначених правил. Обробка інструкцій оновлення кешу (лістинг 3.5) гарантує узгоджену зміну кешованих представлень, а механізми пакетного оновлення (лістинг 3.6) дозволяють зменшити кількість операцій читання та запису при роботі з великими обсягами змін.

Керування окремими кешованими сутностями реалізується за допомогою алгоритмів «setItemToCache» та «removeItemFromCache» (лістинги 3.7, 3.8), які забезпечують коректне оновлення та видалення даних з урахуванням залежностей і каскадної інвалідації. Сукупна робота цих алгоритмів забезпечує узгодженість стану кешу з основним джерелом даних у режимі, наближеному до реального часу.

Порівняльний аналіз з традиційними імперативними підходами до кешування, зокрема на основі Redis, показує, що декларативна модель забезпечує вищий рівень автоматизації процесів інвалідації та зниження зв'язності між сервісами. На відміну від моделей, у яких допускається тимчасове збереження застарілих даних до моменту запиту, запропонований підхід орієнтований на реактивну інвалідацію кешу у відповідь на зміну стану бізнес-даних, що зменшує ймовірність виникнення неконсистентних станів у розподіленому середовищі [22, 23, 24, 25].

До обмежень підходу слід віднести додаткову складність конфігурації та необхідність інтерпретації декларативних правил під час виконання. Водночас такі накладні витрати розглядаються як прийнятний компроміс з огляду на підвищення керованості кешування, спрощення супроводу системи та можливість еволюції логіки інвалідації без суттєвих змін архітектури [26].

Висновки до розділу 3

У межах даного розділу досліджено теоретичні та архітектурні аспекти побудови моделі декларативної інвалідації кешу в контексті високонавантажених мікросервісних систем. Проведено аналіз поширених кеш-систем, зокрема Redis, Memcached та NCache, та виявлено обмеження традиційних підходів до інвалідації кешу, пов'язані з імперативним керуванням, підвищеною зв'язністю між сервісами та складністю забезпечення узгодженості даних у масштабованих архітектурах.

На основі проведеного аналізу сформульовано принципи побудови декларативної моделі інвалідації кешу, що ґрунтується на концепції локальної відповідальності сервісів і принципі володіння даними. Запропонований підхід передбачає, що кожен мікросервіс самостійно ініціює оновлення кешу в межах власної зони відповідальності, що дозволяє зменшити міжсервісні залежності, усунути потребу в централізованій координації та підвищити автономність і масштабованість системи.

Розроблена концептуальна модель включає механізми керування кешем на основі декларативних правил, винесених за межі бізнес-логіки, а також централізований доступ до інфраструктури кешування через API-шлюз. Така організація забезпечує уніфікацію взаємодії з кешем і спрощує еволюцію логіки інвалідації без необхідності суттєвих змін архітектури системи.

Для підтримки практичної реалізації запропонованого підходу розроблено набір спеціалізованих алгоритмів, що охоплюють ключові операції роботи з кешем, зокрема доступ до даних, створення нових елементів, пакетне оновлення та каскадне очищення кешованих представлень. Сукупне застосування цих алгоритмів спрямоване на забезпечення консистентності стану кешу, зменшення кількості застарілих об'єктів і зниження затримок при доступі до даних.

Порівняльний аналіз із традиційними імперативними підходами до кешування, включаючи рішення на основі Redis та апаратні моделі кешування, дозволяє окреслити потенційні переваги декларативного підходу в аспектах автоматизації інвалідації, зниження зв'язності між компонентами та підвищення

гнучкості масштабування. Водночас встановлено, що додаткові накладні витрати, пов'язані з інтерпретацією декларативних правил, є прийнятним компромісом з огляду на покращення керованості та передбачуваності поведінки системи.

Отримані результати свідчать про перспективність подальшого розвитку декларативного підходу до інвалідації кешу в розподілених вебсистемах. Подальші дослідження доцільно спрямувати на експериментальну оцінку ефективності моделі за різних сценаріїв навантаження, а також на інтеграцію механізмів конфігурації правил, моніторингу оновлень і взаємодії з подієвими шинами в умовах промислової експлуатації.

4 ПОБУДОВА МАСШТАБОВАНОЇ ІНФРАСТРУКТУРИ ТА ВПРОВАДЖЕННЯ ОПТИМІЗАЦІЙ У ВЕБСИСТЕМАХ З ВИСОКИМ НАВАНТАЖЕННЯМ

У сучасних високонавантажених вебсистемах, особливо тих, що орієнтовані на мультибрендові B2B-рішення, питання ефективного розгортання фронтенд-застосунків набуває критичного значення. З розвитком підходів до компонування інтерфейсів, все частіше використовуються мультифронтендні архітектури, які дозволяють масштабувати рішення за рахунок повторного використання бібліотек, адаптивного дизайну та централізованого керування кодом. Водночас така гнучкість супроводжується низкою складнощів, зокрема у конфігурації CI/CD (безперервна доставка / безперервне розгортання) пайплайнів, підтримці незалежності компонентів і зниженні часу розгортання.

У цьому контексті, побудова ефективної інфраструктури розгортання базується не лише на технічній організації репозиторіїв, а й на застосуванні оптимізаційних стратегій, таких як кешування, паралельне виконання задач і використання модульних збірок. Особливої актуальності набуває перехід від традиційної послідовної обробки до паралелізації процесів у межах CI/CD пайплайнів, що дозволяє суттєво скоротити затримки та підвищити ефективність використання обчислювальних ресурсів.

У даному розділі розглядається формалізована методика оптимізації процесу розгортання мультифронтендної архітектури, яка базується на використанні монорепозиторіїв, багаторазово використовуваних бібліотек та автоматизованих пайплайнів з підтримкою кешування і паралельного виконання.

4.1 Архітектура мульти-фронтендного середовища у монорепозиторіях

Поява мульти-фронтендних проєктів як реакція на потребу обслуговування кількох брендів або сегментів ринку в межах однієї платформи зумовила зміну

підходів до організації структури фронтенд-коду. В умовах зростання кількості спільних і брендових компонентів, виникла потреба у створенні централізованого середовища розробки, що забезпечує модульність, узгодженість і керованість змін [98].

Монорепозиторна модель стала поширеним рішенням для таких проєктів, оскільки дозволяє розміщувати всі застосунки, бібліотеки та утиліти в єдиному репозиторії з наскрізною системою контролю версій. Такий підхід забезпечує зручну синхронізацію змін, спрощує інтеграцію та підтримує повторне використання коду. Водночас він висуває нові вимоги до побудови архітектури, що охоплює і структурування залежностей, і підтримку незалежності брендів [99].

4.1.1 Визначення концепції мульти-фронтендного проєкту: спільні та брендові компоненти

Мульти-фронтендний проєкт передбачає підтримку кількох клієнтських інтерфейсів, які поділяють спільну функціональну основу, але реалізують індивідуальні стилістичні або поведінкові особливості відповідно до потреб різних брендів чи бізнес-одиниць. Такий підхід дозволяє створити централізоване середовище розробки, де повторно використовувані компоненти (shared components) забезпечують єдине джерело істини для логіки, API-інтеграцій і базових UI-елементів, а брендові компоненти (brand-specific components) реалізують візуальну та функціональну диференціацію [100].

Спільні компоненти формують ядро проєкту, що охоплює загальні бібліотеки, утиліти, теми оформлення, сервіси доступу до даних, а також типові шаблони для сторінок або блоків інтерфейсу. Їхня повторна використання сприяє уніфікації користувацького досвіду, скороченню дублювання коду та зменшенню витрат на підтримку. Оновлення таких компонентів відбувається централізовано, що дозволяє водночас поширювати зміни на всі залежні фронтенди [5].

Натомість брендові компоненти реалізуються як розширення або перевизначення базових, забезпечуючи індивідуальний стиль, контент або поведінку відповідно до специфіки кожного бренду. Вони можуть охоплювати кастомні теми, стилі, специфічну верстку окремих сторінок або адаптовану логіку взаємодії. У контексті монорепозиторної архітектури така ізоляція реалізується за допомогою просторової організації (наприклад, окремих директорій для кожного бренду) або механізмів умовного імпорту, які дозволяють динамічно підключати потрібні реалізації залежно від конфігурації [101].

Визначення чіткого поділу між спільними та брендовими компонентами є критично важливим для масштабованості та стабільності системи. Такий поділ дозволяє уникнути надмірної зв'язності між частинами проєкту, забезпечує гнучкість у кастомізації, а також спрощує тестування й оновлення. З точки зору CI/CD-процесів це також відкриває можливість для незалежного збирання та розгортання окремих фронтендів із використанням спільного коду як бібліотечного пакета [102].

Таким чином, концепція мульти-фронтендного проєкту передбачає баланс між уніфікацією та індивідуалізацією, де спільні компоненти забезпечують стабільність і повторюваність, а брендові – адаптивність і варіативність інтерфейсів.

4.1.2 Роль монорепо-архітектури у підтримці масштабованості та модульності

Монорепозиторій (monorepo) – це підхід до організації кодової бази, при якому всі пов'язані проєкти, бібліотеки та інструменти розміщуються в межах одного репозиторію з єдиною історією змін і загальною структурою управління версіями. У контексті мульти-фронтендних систем такий підхід забезпечує структурну цілісність і сприяє підтримці масштабованості та модульності архітектури [103].

Однією з ключових переваг монорепо є можливість централізованого контролю над спільними залежностями та бібліотеками. Це особливо важливо у мультибрендових проєктах, де кілька інтерфейсів мають спільне ядро компонентів, утиліт або тем оформлення. Завдяки спільному контексту розробки оновлення бібліотеки відображається на всіх пов'язаних фронтендах без потреби дублювання змін або управління окремими версіями [99].

Монорепо також спрощує підтримку ізольованих брендових застосунків у межах єдиної кодової бази. Використання незалежних директорій, умовного імпорту або конфігурацій на рівні CI/CD дозволяє розробникам працювати над окремими брендами, не порушуючи стабільності інших. Це підтримує модульність системи та зменшує ймовірність побічних ефектів при внесенні змін [100].

Ще одним важливим аспектом є підвищення прозорості та спрощення автоматизації. Загальний репозиторій дозволяє легко відслідковувати залежності, інтегрувати інструменти аналізу коду, лінтингу, тестування та збірки на рівні всієї системи. Це створює передумови для впровадження масштабованих CI/CD пайплайнів, які можуть паралельно збирати пакети, застосунки та генерувати Docker-образи для розгортання, що доведено експериментально в межах дослідження [100].

У той самий час, монорепо-модель висуває вимоги до структурної дисципліни та архітектурного планування. Зростання кількості пакетів і застосунків підвищує ризики появи прихованих залежностей, неочікуваного поширення змін та ускладнення тестування. Для запобігання цим проблемам необхідно впроваджувати чіткі принципи модульності, а також автоматизовані механізми перевірки залежностей, контроль версій спільних бібліотек і кешування артефактів [102].

Монорепозиторна архітектура відіграє ключову роль у забезпеченні структурної узгодженості та повторюваності процесів у мульти-фронтендних проєктах. Вона створює технічну основу для масштабування команди, прискорення доставки оновлень і централізованого управління всіма компонентами системи.

4.1.3 Проблеми повторного використання бібліотек і управління залежностями

Повторне використання спільних бібліотек є фундаментальним принципом ефективної розробки у мульти-фронтендному середовищі. Воно дозволяє скоротити дублювання коду, забезпечити уніфіковану поведінку компонентів і пришвидшити впровадження змін. Проте в умовах зростання масштабів проєкту та кількості залежних фронтендів виникають суттєві виклики, пов'язані з контролем сумісності, ізоляцією змін та забезпеченням стабільності всього середовища.

Однією з ключових проблем є висока зв'язність між модулями. Зміни у спільній бібліотеці можуть мати неконтрольований вплив на всі залежні застосунки, особливо у випадках, коли відсутня жорстка фіксація версій або інтеграційні тести для перевірки зворотної сумісності. У монорепозіторії така проблема часто посилюється тим, що зміни потрапляють до всіх фронтендів одночасно, незалежно від готовності до їх адаптації [103].

Ще одним викликом є поява прихованих залежностей – ситуацій, коли один брендовий застосунок непрямо використовує модуль, створений для іншого бренду. Це призводить до ерозії модульності, ускладнює рефакторинг і створює технічний борг. Такі ситуації виникають у разі відсутності чіткої структури експорту та контролю за точками входу до спільних пакетів.

Особливу увагу слід приділяти механізмам управління версіями спільних бібліотек. У середовищі, де CI/CD пайплайни працюють із кешованими артефактами, відсутність контрольованих версій може призводити до кешування застарілих або несумісних залежностей. Це ускладнює діагностику помилок і знижує повторюваність збірки. Впровадження стратегій тегування, семантичного версіонування або використання внутрішніх прм-реєстрів дозволяє частково розв'язати цю проблему, але потребує додаткової автоматизації в пайплайнах [98].

Також варто враховувати проблему надмірної генералізації спільних бібліотек. У прагненні зробити модулі універсальними розробники часто створюють абстракції, які складно підтримувати або тестувати. Це знижує

швидкість розробки й ускладнює адаптацію компонентів до конкретних бізнес-вимог брендів.

Для подолання зазначених проблем у межах дослідження були впроваджені структурні й процесні обмеження: чіткий поділ бібліотек за призначенням, контроль імпортів на рівні лінтингу, кешування збірок за хешами комітів, а також автоматизоване очищення застарілих артефактів. Експериментальні результати продемонстрували, що такі заходи сприяють зменшенню помилок, покращенню контрольованості залежностей і підвищенню стабільності розгортань [101].

Ефективне повторне використання бібліотек потребує не лише спільної кодової бази, а й чітко визначених практик управління залежностями, ізоляції змін та інфраструктурної підтримки процесів версіонування, тестування і кешування.

4.2 Оптимізація CI/CD пайплайнів за допомогою кешування та паралелізації

У мультифронтендних системах, що розробляються в межах монорепозіторіїв, ефективність CI/CD пайплайнів безпосередньо впливає на швидкість і стабільність розгортання. Для забезпечення масштабованості процесу розробки ключове значення мають кешування результатів збірки та паралельне виконання незалежних задач [36].

Кешування між етапами пайплайна реалізується на рівнях встановлення залежностей, збереження артефактів компіляції та побудови Docker-образів. Це дозволяє уникнути повторного виконання незмінних обчислень і скоротити загальний час збірки. Паралельна обробка незалежних бібліотек і фронтенд-застосунків знижує затримки в пайплайні та забезпечує рівномірне використання обчислювальних ресурсів [37].

Використання GitLab CI у поєднанні з конфігураціями у форматі YAML забезпечує гнучке описання етапів збірки з урахуванням залежностей, правил кешування та ізоляції завдань. Такий підхід формує надійну технічну основу для

стабільного, контрольованого та прискореного розгортання у складних мультифронтендних середовищах [101].

4.2.1 Реалізація кешування між етапами збірки: залежності, артефакти, Docker-образи

У багатокомпонентних системах із загальними бібліотеками та залежностями навіть незначні зміни в коді можуть призводити до повної повторної збірки, що помітно збільшує тривалість пайплайна. Щоб уникнути цього, сучасні CI/CD системи передбачають реалізацію кешування на кількох рівнях, яке дозволяє повторно використовувати вже оброблені ресурси без необхідності їхньої повторної компіляції чи повторного завантаження.

Перший рівень – кешування зовнішніх залежностей. У типовому пайплайні встановлення залежностей є одним із найбільш повторюваних процесів, особливо при використанні JavaScript-фреймворків з великою кількістю пакетів. У GitLab CI або подібних системах кешування реалізується через ключі, побудовані на хешах файлів «package.json» або «yarn.lock». Це дозволяє зберігати «node_modules» або аналогічні директорії у внутрішньому кеш-сховищі та використовувати їх повторно доти, доки не відбуваються зміни в конфігураційних файлах, значно зменшуючи навантаження на файлову систему та мережеві ресурси [36].

Другий рівень стосується кешування артефактів збірки. На цьому етапі створюються проміжні результати компіляції, які використовуються на наступних кроках пайплайна або при подальших запусках CI. Наприклад, зібрані пакети або загальні бібліотеки можна зберігати як артефакти, прив'язані до конкретного коміту або гілки. Це дозволяє швидко відновлювати стан збірки без необхідності її повторного запуску в повному обсязі.

Додатково, при розгалуженій структурі з декількома фронтенд-додатками, артефакти окремих пакетів можуть використовуватися не лише в одному пайплайні, а й як частина міжпроектної інтеграції.

На завершальному етапі використовується кешування при створенні Docker-образів. Завдяки особливостям шарової архітектури Docker, зміни в кодовій базі не призводять до повної регенерації образу, якщо нижні шари (наприклад, базовий образ або встановлення залежностей) залишилися незмінними. Для цього застосовуються механізми локального кешування або використання віддалених реєстрів із можливістю збереження та повторного використання шарів, що не змінилися. Це дозволяє суттєво скоротити час на складання та публікацію контейнерів, особливо у хмарних середовищах з обмеженим часом виконання завдань.

Комплексне застосування кешування на кожному з етапів забезпечує не лише пришвидшення процесів, а й стабільність пайплайна, підвищує повторюваність збірки та полегшує відкат у разі виникнення помилок. У сукупності ці підходи становлять необхідну інфраструктурну базу для реалізації високопродуктивного CI/CD у складних вебсистемах.

4.2.2 Впровадження паралельної збірки пакетів і фронтенд-додатків

У масштабованих мультифронтендних системах із великою кількістю незалежних пакетів та застосунків значну частину часу виконання CI/CD пайплайна займає процес збирання. Для зменшення тривалості пайплайна доцільним є впровадження паралельного виконання незалежних задач, що дозволяє максимально використати обчислювальні ресурси та уникнути зайвих очікувань між етапами.

На рис. 4.1 представлено узагальнену модель побудови CI/CD пайплайна у мультифронтендному середовищі з реалізацією паралелізації на всіх можливих етапах. Пайплайн структуровано у п'ять ключових фаз, які умовно поділяються як горизонтально, так і по потоках залежностей.

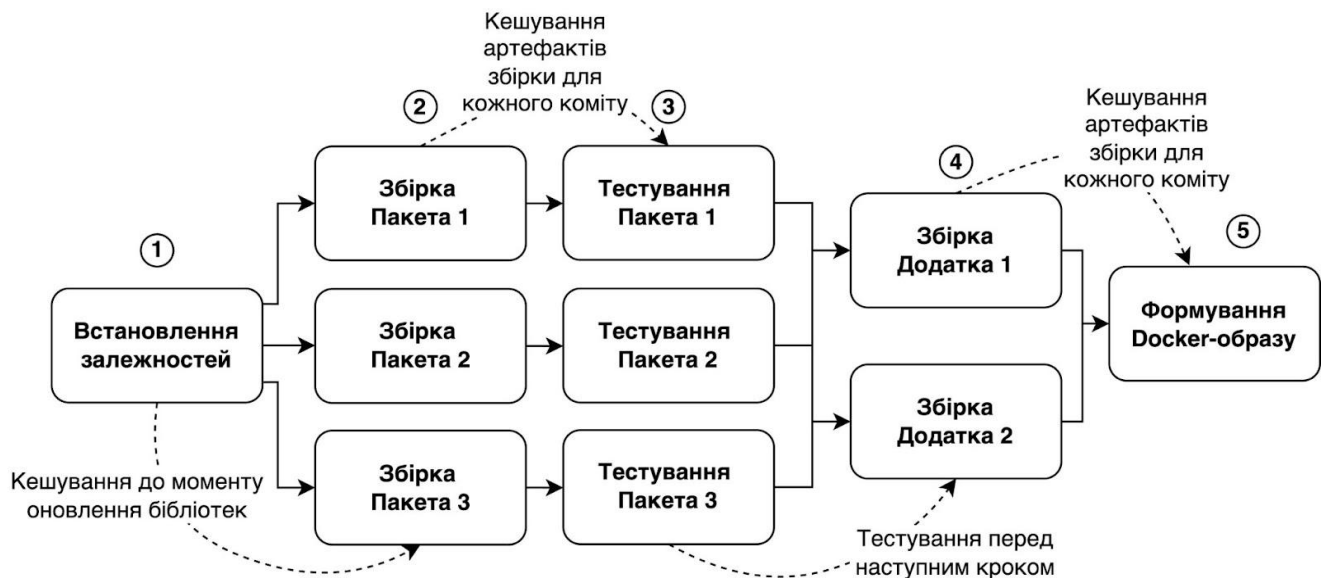


Рисунок 4.1 – Кроки CI пайплайнів та завдання (jobs)

Встановлення залежностей – початковий етап, що забезпечує середовище для подальших дій. Сторонні бібліотеки встановлюються один раз для всіх подальших процесів. При цьому можуть використовуватися механізми кешування, що дозволяють зберігати залежності до моменту внесення змін у відповідні конфігураційні файли. Це суттєво знижує навантаження та прискорює підготовку.

Збірка пакетів – паралельна компіляція окремих незалежних бібліотек або внутрішніх модулів. Пакети, які не мають міжсобойних залежностей, можуть збиратися одночасно у вигляді окремих завдань. Після успішної збірки артефакти кожного пакета кешуються з прив'язкою до хешу коміту, забезпечуючи повторюваність у наступних пайплайнах або етапах [36].

Тестування пакетів – після компіляції запускаються модульні тести, які також виконуються паралельно. Це дозволяє локалізувати потенційні помилки на ранньому етапі, не блокуючи подальші дії у суміжних частинах системи. Важливо, що перехід до наступного етапу дозволено лише за умови проходження всіх перевірок.

Збірка застосунків – побудова брендових або продуктових фронтенд-додатків, які залежать від результатів попередніх пакетів. Ці застосунки зазвичай розробляються як ізольовані проєкти в межах монорепозіторію, тому можуть

збиратися незалежно, також у паралельному режимі. Вихідні артефакти кешуються за кожним комітом для прискорення подальших розгортань [36].

Фінальна збірка являє собою завершальний етап пайплайна, на якому формуються інсталяційні або розгортальні артефакти, зокрема Docker-образи, архіви з компілятором, статичні бандли чи інші пакети, необхідні для деплоюменту. На цьому кроці також застосовуються механізми кешування на рівні образів, що додатково оптимізує час публікації або доставки.

4.2.3 Оптимізація використання ресурсів: артефакти та GitLab CI Runners

В межах даного дослідження для демонстрації оптимізації CI/CD пайплайнів використано платформу GitLab та декларативну конфігурацію у форматі YAML. Хоча на практиці можуть застосовуватись інші системи (наприклад, GitHub Actions, Jenkins або CircleCI), GitLab CI обрано як зручний і поширений інструмент для наочності реалізації основних принципів кешування, побудови артефактів та управління ресурсами [103].

У прикладі дослідження проєкт має структуру монорепозиторію, де:

- спільні бібліотеки й пакети розміщено у директорії «packages»;
- окремі фронтенд-додатки – у директорії «apps».

Один із способів зменшення дублювання дій між завданнями – використання глобального кешу. У прикладі кешуються (лістинг 4.1):

- зовнішні залежності «node_modules»;
- результати попередньої збірки пакетів «packages/*/build»;
- проміжні файли додатків «apps/*».
- «CI_COMMIT_SHORT_SHA» – короткий хеш коміту, який гарантує унікальність кешу для кожної зміни коду;

– «policy: pull-push» – політика кешування, означає, що кеш буде завантажено на початку завдання «pull» і оновлено після виконання «push», забезпечуючи узгодженість між стадіями пайплайна.

```
default:
  cache: &global_cache
    key: ${CI_COMMIT_SHORT_SHA}
    paths:
      - "node_modules"
      - "packages/*/node_modules"
      - "packages/*/build"
      - "apps/*"
    policy: pull-push
```

Лістинг 4.1– Глобальна конфігурація кешу на етапах неперервної інтеграції (CI/CD)

На початку виконується «yarn install» – команда для інсталяції усіх залежностей проєкту. У прикладі також використовується прапорець «--cache-folder», який зберігає кеш Yarn у спеціальній директорії «.yarn» для прискорення наступних встановлень сторонніх бібліотек (лістинг 4.2).

```
install_dependencies:
  stage: install_dependencies
  image: $NODE_IMAGE
  before_script:
    - yarn install --cache-folder .yarn
  resource_group: $CI_COMMIT_REF_SLUG
  interruptible: true
  script:
    - yarn install
  only:
    changes:
```

- `package.json`
- `yarn.lock`
- `.gitlab-ci.yml`

Лістинг 4.2 – Встановлення необхідних бібліотек на етапі встановлення залежностей

«`resource_group: $CI_COMMIT_REF_SLUG`» – групує завдання за гілкою, щоб уникнути одночасного запуску конфліктних джобів для однієї гілки.

«`interruptible: true`» – дозволяє перервати джоби, якщо з'явиться новий запуск тієї ж гілки (наприклад, при пуші нових оновлень).

«`only: changes:`» – джоба виконується лише при зміні файлів, що впливають на залежності (наприклад, `package.json`), що економить ресурси.

Кожен пакет у «`packages/*`» збирається через окреме завдання із використанням «`Yarn workspaces`» (лістинг 4.3). Завдання, залежні від встановлення залежностей (`needs`), але виконуються паралельно, що дозволяє істотно зменшити час пайплайна:

```
build_package1:
  image: $NODE_IMAGE
  needs:
    - job: install_dependencies
  resource_group: $CI_COMMIT_REF_SLUG
  interruptible: true
  script:
    - yarn workspace @project/package1 run build
  stage: build_packages
```

Лістинг 4.3 – Збірка пакета 1 на етапі збірки пакетів

Завдання тестування виконується після збірки відповідного пакета (послідовно), але незалежно від інших пакетів – отже, також може бути запущено паралельно (лістинг 4.4).

```
test_package1:
  image: $NODE_IMAGE
  needs:
    - job: build_package1
  resource_group: $CI_COMMIT_REF_SLUG
  interruptible: true
  script:
    - yarn workspace @project/package1 run test
  stage: test_packages
```

Лістинг 4.4 – Тестування пакета 1 на етапі тестування пакетів

Додатки з директорії «apps/*» збираються лише після успішного проходження тестів пакетів, від яких вони залежать. Це забезпечується через needs, і дані завдання також можуть виконуватись паралельно (лістинг 4.5).

Фінальний етап «assemble» перевіряє, чи наявні потрібні ресурси (залежності, збірки пакетів, файли додатків). Якщо їх немає – завдання не запускається, що запобігає створенню неповного або порожнього Docker-образа. Для побудови образу використовується Kaniko, який працює без потреби root-доступу (лістинг 4.6).

Дане завдання є послідовним, оскільки залежить від результатів збірки обох додатків. На завершення середовище очищується після виконання основного скрипту у блоці «after_script», щоб уникнути впливу попередніх збірок на наступні запуски та оптимізувати розмір кешу між етапами виконання пайплайнів [36].

Наведена конфігурація демонструє один із можливих підходів до ефективної організації CI/CD у монорепозиторіях, який поєднує принципи повторного

використання артефактів, паралельного виконання задач і контрольованого управління ресурсами.

```
build_app1:
  image: $NODE_IMAGE
  needs:
    - job: test_package1
    - job: test_package2
  resource_group: $CI_COMMIT_REF_SLUG
  interruptible: true
  script:
    - yarn workspace @project/apps/app1 run build
  stage: build_apps
```

Лістинг 4.5 – Збірка додатку 1 на етапі збірки додатків

```
assemble:
  stage: assemble
  needs:
    - job: build_app1
    - job: build_app2
  resource_group: $CI_COMMIT_REF_SLUG
  interruptible: true
  before_script:
    - |
      IFS=$'\n'
      check_dirs="**/node_modules
      packages/*/build
      apps/*"
      for dir in $check_dirs; do
        if ! ls $dir &> /dev/null; then
          exit 1
        fi
      done
```

```

- mkdir -p /kaniko/.docker
script:
- /kaniko/executor
  --build-arg "platform='linux/amd64'"
  --context $CI_PROJECT_DIR
  --use-new-run
  --dockerfile $CI_PROJECT_DIR/Dockerfile
  --destination${FINAL_PATH}:${CI_COMMIT_REF_NAME}
after_script:
- rm -rf **/node_modules apps/* packages/*/build

```

Лістинг 4.6 – Створення та завантаження Docker-образу на етапі збірки

Завдяки структурованій побудові залежностей між завданнями, система здатна точно визначити мінімально необхідний обсяг робіт при кожному запуску, що дозволяє уникати зайвих етапів збірки та тестування.

Поетапне кешування (зовнішніх залежностей, артефактів та результатів попередніх збірок) істотно скорочує час виконання пайплайнів і мінімізує навантаження на CI/CD середовище [36].

4.3 Аналіз продуктивності та напрями оптимізації CI/CD

Підвищення ефективності CI/CD процесів є критичним чинником для стабільної та швидкої доставки змін у багатокомпонентних вебсистемах. У контексті мультифронтендної архітектури, де взаємозалежні пакети та додатки збираються в межах спільного репозиторію, тривалість виконання пайплайнів суттєво впливає на продуктивність розробки та інфраструктурні витрати.

Порівняння послідовної та паралельної моделі виконання дозволяє оцінити реальний ефект від впровадження оптимізацій, зокрема кешування та паралельної обробки задач. Аналіз експериментальних вимірювань демонструє, які етапи є критичними для прискорення процесу, і допомагає визначити найбільш доцільні напрями подальшого вдосконалення CI/CD-пайплайнів.

4.3.1 Порівняння продуктивності: послідовне та паралельне виконання (експериментальні результати)

Для оцінки ефективності запропонованих оптимізацій CI/CD пайплайнів проведено серію експериментів із використанням двох сценаріїв виконання: традиційного послідовного та оптимізованого паралельного. Метою експерименту було кількісне визначення впливу паралелізації на загальну тривалість пайплайна, а також виявлення етапів, що мають найбільший потенціал для прискорення [36].

Пайплайн складався з п'яти основних етапів:

- встановлення зовнішніх залежностей;
- збірка бібліотек (пакетів);
- запуск тестів;
- компіляція фронтенд-додатків;
- формування контейнерного образу для розгортання.

Щоб забезпечити відтворюваність і достовірність результатів, були дотримані такі умови:

- усі тести запускались в межах одного часово узгодженого проміжку, щоб мінімізувати зовнішні коливання в навантаженні;
- версії пакетів і конфігурацій були ідентичними;
- усі запуски виконувалися на одній віртуальній машині з постійними характеристиками;
- використовувалися безкоштовні GitLab CI Runners, що відповідає умовам реального хмарного середовища;
- перед кожним запуском очищувалися кеші та артефакти для виключення впливу залишкових даних.

Приклад Gitlab CI конфігурації, за яким проводився експеримент наведено у додатку Г.

У традиційному (послідовному) режимі проходження повного CI/CD пайплайна займало 1661 секунду (27 хв 41 с). Після впровадження паралельного

виконання незалежних завдань, цей час скоротився до 1150 секунд (19 хв 10 с), що дало економію 30,7% від початкової тривалості. Результати експериментів наведено у додатках Д, Е, Є.

Розподіл виграшу за етапами (таблиця 4.1):

- встановлення залежностей (час зменшився незначно – з 47 до 45 секунд);
- збірка пакетів (у послідовному режимі цей етап займав 614 секунд, оскільки пакети будувались один за одним. У паралельному виконанні тривалість визначалась найдовшим із трьох одночасних процесів – 278 секунд. Скорочення часу становить 54,7%);
- тестування (у послідовному виконанні етап тривав 283 секунди. При паралельному запуску трьох завдань максимальний час склав 246 секунд, що дало 13,1% виграш у продуктивності);
- компіляція застосунків (послідовна компіляція тривала 564 секунди. При паралельній побудові двох застосунків етап скоротився до 428 секунд. Покращення становить 24,1%);
- збірка Docker-образу (тривалість залишилася незмінною – 153 секунди, оскільки цей етап не був оптимізований або поділений на підзадачі).

Таблиця 4.1– Порівняння проходження CI/CD пайплайнів у послідовному та паралельному режимах

Крок	Послідовні завд. (с)	Паралельні завд. (с)
(1) Встановлення залежностей	47с	45с
(2) Збірка 3 пакетів	614с	232с / 278с / 224с
(3) Тестування 3 пакетів	283с	104с / 246с / 115с
(4) Збірка 2 додатків	564с	291с / 428с
(5) Збірка Docker образу	153с	153с
Загальний час очікування	1661с (27м 41с)	1150с (19м 10с)

Експериментально доведено, що найбільший виграш у продуктивності досягається на етапах, де можливе незалежне виконання завдань, тобто:

- збірка ізольованих пакетів;
- паралельне тестування;
- компіляція окремих застосунків.

Особливо ефективною виявилась паралельна збірка бібліотек, що дозволила скоротити час більш ніж удвічі. Це свідчить про доцільність виокремлення малозалежних модулів для незалежного збирання, що також позитивно впливає на масштабованість архітектури.

Крім зменшення тривалості, оптимізований сценарій забезпечує кращу повторюваність і контроль над пайплайном – завдяки розділенню задач, кешуванню артефактів і чітко визначеним залежностям між завданнями.

Хоча не всі етапи піддаються паралелізації (наприклад, інсталяція залежностей або збірка образу), результат експерименту доводить, що навіть часткова оптимізація здатна забезпечити суттєву економію часу при розгортанні мультифронтендних систем. Такий підхід може масштабуватись на більші проєкти та різні інфраструктурні платформи.

4.3.2 Перспективи подальшої оптимізації: баланс кешу, гібридні стратегії розгортання

Подальша оптимізація CI/CD у мультифронтендних системах на основі монорепозіторію передбачає не лише пришвидшення збірки, а й раціоналізацію процесу формування та доставки артефактів. Один із ключових напрямів – оптимізація складу контейнерних образів: замість включення всього коду з моногеро до кожного образу, варто формувати ізольовані збірки, що містять лише необхідні залежності та код конкретного застосунку. Такий підхід дозволяє зменшити обсяг образів, прискорити їх створення і завантаження.

Ще один важливий аспект – селективне включення бібліотек. У великих проєктах спільні пакети можуть використовуватись лише частиною застосунків. Включення всіх бібліотек у кожен збірку призводить до надлишкового обсягу даних і зайвого використання ресурсів. Оптимізація полягає в аналізі фактичних залежностей на етапі побудови і включенні до образу лише тих компонентів, які дійсно використовуються. Це знижує навантаження на CI/CD ранери, покращує час виконання завдань і спрощує відладку у разі помилок [98].

Розгортання також вимагає уваги до мережових аспектів. Якщо між реєстром образів (Docker Registry) та інфраструктурою виконання деплою існує затримка – наприклад, через географічну віддаленість або повільні канали зв'язку – це може негативно впливати на час масштабування чи аварійного відновлення. Для мінімізації цих затримок доцільно використовувати edge-сервери, кешуючі дзеркала або CDN, які дозволяють швидше доставляти образи у цільове середовище [100].

Крім технічної оптимізації складу та доставки, важливим напрямом розвитку є використання гібридних стратегій розгортання, які адаптуються до характеру змін та їхнього ризику. Наприклад, canary-реліз дозволяє розгортати нову версію лише на обмеженій кількості серверів чи для певної частини користувачів – це забезпечує раннє виявлення проблем без впливу на всю систему. Для стабільних частин системи доцільно застосовувати blue-green deployment, при якому одночасно підтримуються дві версії середовища (активна та резервна), що дає змогу миттєво перемикатися між ними без простоїв.

У CI/CD-системах на кшталт GitLab CI ці стратегії підтримуються через умовні завдання, які активуються за певних умов – наприклад, лише для основної гілки (main), тільки при зміні важливих конфігурацій або за певних тегів. Це дозволяє будувати динамічні пайплайни, що не лише підвищують гнучкість доставки, а й економлять ресурси [38].

Таким чином, наступний рівень оптимізації – це перехід від уніфікованих процедур до адаптивних, контекстно-залежних конфігурацій, орієнтованих на:

- специфіку застосунків і бібліотек;

- частоту і характер змін;
- доступну мережеву інфраструктуру;
- вимоги до стабільності та безпеки.

4.3.3 Обмеження оптимізацій та виклики масштабування

Хоча оптимізація CI/CD пайплайнів і впровадження монорепозиторної архітектури надають значні переваги у вигляді повторного використання коду, спрощеної синхронізації змін та централізованого управління, із масштабуванням таких рішень виникають нові обмеження та складнощі, які потребують окремого аналізу [36].

Одним з основних викликів є зростання кількості застосунків і бібліотек у межах одного репозиторію. Зі збільшенням кількості компонентів у моногеро зростає складність підтримки цілісності коду, контроль за залежностями ускладнюється, а оновлення спільних пакетів може мати непередбачувані наслідки для кількох продуктів одночасно. У випадку помилки у загальній бібліотеці її вплив може поширитися на всі залежні застосунки, що значно ускладнює відкат та пошук джерела проблеми.

Ще одна важлива проблема – надмірний доступ розробників до всього репозиторію. У традиційних багаторепозиторних підходах доступ обмежується межами конкретного продукту або мікросервісу, тоді як у моногеро всі учасники команди можуть змінювати загальні ресурси. Це підвищує ризики неконтрольованих змін, конфліктів та помилок у спільному коді, а також ускладнює аудит.

Зростання масштабів також призводить до ускладнення пайплайнів CI/CD. Попри кешування, паралелізацію і умовність виконання завдань, загальний час виконання пайплайнів зростає через збільшення кількості компонентів і залежностей. Це вимагає дедалі глибшої кастомізації логіки виконання: вибіркове виконання завдань, фільтрація змін, динамічне формування списків залежних

компонентів. Без таких механізмів ресурси CI можуть бути перевантажені, а ефективність – знижена.

Також слід враховувати організаційні аспекти масштабування:

- централізована модель ускладнює делегування відповідальності за окремі частини системи;
- складніше проводити рев'ю, якщо pull request охоплює кілька продуктів або зміни в загальних бібліотеках;
- онбординг нових розробників потребує значно глибшого занурення у структуру репозиторію, а також розуміння архітектурних залежностей між пакетами та застосунками.

Окрім цього, зростає складність підтримки документації та тестового покриття, оскільки оновлення одного компонента може потребувати перегляду інтеграційних тестів у багатьох інших.

Нарешті, обмеження безкоштовних CI/CD ранерів або локальних ресурсів стають більш відчутними в умовах активного масштабування. Із кожним новим додатком або бібліотекою зростає кількість завдань, що конкурують за ресурси, і навіть при правильному кешуванні та паралельному виконанні може виникати затримка в чергах на виконання або нестача доступних середовищ.

Таким чином, незважаючи на переваги, монорепозиторій і оптимізації CI/CD потребують чіткого стратегічного планування, впровадження політик доступу, модульної ізоляції та автоматизації керування залежностями. Лише за таких умов можна забезпечити масштабованість без втрати контрольованості та якості.

Висновки до розділу 4

У межах дослідження даного розділу проведено аналіз архітектурних і інфраструктурних рішень для мультифронтендних систем, реалізованих на основі монорепозиторіїв. Дослідження охоплює аспекти організації коду, оптимізації CI/CD та виклики масштабування.

Виявлено, що використання монорепозіторіїв є доцільним для централізованого управління спільними і брендовими компонентами. Запропоновано структуру, що сприяє повторному використанню коду, синхронізації змін і модульності. Разом із тим досліджено ризики, пов'язані з міжпроектними залежностями та потребою ізоляції середовищ.

Встановлено, що неконтрольоване повторне використання бібліотек у спільному середовищі може призводити до порушення стабільності системи. Запропоновано впровадження контролю версій, автоматичних перевірок сумісності та ізоляції критичних компонентів для зменшення впливу помилок.

Реалізацію кешування на рівні залежностей, артефактів і контейнерних образів виявлено як ключовий фактор скорочення тривалості пайплайнів. Особливу ефективність продемонстровано при використанні хеш-контролю залежностей і проміжних результатів збірки.

У результаті експериментального порівняння доведено, що паралельна збірка і тестування дає до 30% вигрaш у часі, а найбільший приріст продуктивності спостерігається при паралельній збірці незалежних пакетів (до 54,7%). Паралельне виконання рекомендовано як базова стратегія оптимізації для великих систем.

Проаналізовано можливості використання «needs», «resource_group», «interruptible», умовних завдань і політик кешування у GitLab CI. Виявлено, що комбінація цих інструментів дозволяє суттєво знизити навантаження на CI-інфраструктуру та підвищити керованість пайплайнів.

Наголошено на доцільності формування легковагих образів, які містять лише необхідні залежності. Рекомендовано застосовувати гібридні стратегії розгортання («canary», «blue-green») з урахуванням важливості змін. Також запропоновано географічну оптимізацію доставки артефактів для зменшення затримок у продакшн-середовищах.

Зі зростанням кількості застосунків і команд зростає складність доступів, контроль змін і технічна заборгованість. Досліджено потребу у впровадженні політик прав доступу, розподілу відповідальності та автоматизованих інструментів для підтримки цілісності спільного середовища.

ВИСНОВКИ

У дисертаційній роботі вирішено наукову проблему розробки методології проєктування високонавантажених вебсистем з урахуванням архітектурних патернів, підходів до кешування та оптимізації інфраструктурних процесів. Відповідно до сформульованої мети дослідження виконано всі поставлені задачі та отримано науково обґрунтовані результати.

У межах дослідження проаналізовано застосування архітектурних патернів проєктування у високонавантажених вебдодатках з точки зору вирішення задач агрегації даних, уніфікації бізнес-логіки та реалізації механізмів контролю доступу. Показано, що традиційні підходи до взаємодії клієнтських і серверних компонентів є недостатньо гнучкими в умовах мікросервісної архітектури, особливо у B2B-системах з підвищеними вимогами до продуктивності, безпеки, масштабованості та керованості.

Отримав подальший розвиток архітектурний патерн BFF, який, на відміну від існуючих архітектурних рішень, поєднує фабрики API-провайдерів, централізований API-шлюз та підтримує одночасну взаємодію з REST і GraphQL. Запропонована архітектурна модель забезпечує агрегацію даних з різноманітних джерел, централізовану обробку запитів і розмежування прав доступу. Практичне застосування моделі дозволяє скоротити обсяг шаблонного коду, інкапсулювати зони відповідальності та підвищити продуктивність клієнтської частини вебзастосунків.

Виконано аналіз методів і підходів до кешування даних у вебсистемах з мікросервісною архітектурою, за результатами якого виявлено обмеження централізованих та імперативних механізмів інвалідації кешу, зокрема підвищену міжсервісну зв'язаність і складність забезпечення консистентності даних. На цій основі вперше розроблено концепцію декларативної інвалідації кешу, що передбачає автономне управління кешем кожним мікросервісом на основі формалізованих правил керування кешованими об'єктами. Запропонований підхід

дозволяє зменшити міжсервісні залежності, забезпечити консистентність даних і підвищити ефективність доступу до інформації.

Розроблено архітектурні підходи до побудови мультифронтендного середовища в межах монорепозиторіїв. Досліджено практики структуризації спільних і брендових компонентів, проаналізовано ризики міжпроектних залежностей і запропоновано підходи до контролю залежностей та ізоляції середовищ. Обґрунтовано доцільність використання монорепозиторної організації для підвищення стабільності, масштабованості та керованості мультифронтендних вебсистем в умовах розподіленої розробки.

Отримали подальший розвиток методики оптимізації CI/CD-процесів у багатокомпонентних вебсистемах. Запропоновано застосування кешування артефактів і залежностей, паралельного виконання завдань, умовного запуску етапів конвеєра та використання легковагових контейнерних образів. Проведене експериментальне дослідження підтвердило можливість скорочення часу виконання CI/CD-пайплайнів на 30–50 %, зниження навантаження на інфраструктуру та підвищення керованості процесів розгортання.

Отримані результати мають теоретичну та прикладну цінність і можуть бути використані при проєктуванні високонавантажених мікросервісних вебсистем з мультифронтендною організацією, а також у практиці розробки й оптимізації інфраструктурних процесів у сучасних вебдодатках.

Результати дослідження пройшли апробацію на профільних наукових конференціях, а основні положення дисертаційної роботи опубліковано у рецензованих фахових та міжнародних наукових виданнях.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Savani, N., 2023. The future of web development: An in-depth analysis of micro-frontend approaches. *International Journal of Computer Trends and Technology*. 2023. T. 71. № 11. С. 65–69.
2. Abgaz Y., McCarren A., Elger P., Solan D., Lapuz N., Bivol M., Jackson G., Yilmaz M., Buckley J., Clarke P. Decomposition of monolith applications into microservices architectures: A systematic review. *IEEE Transactions on Software Engineering*. 2023. T. 49. № 8. С. 4213–4242.
3. Singh S. Micro-Frontends vs. Monolith Frontends : дис. ... / University of Applied Sciences. 2024.
4. Laigner R., Zhou Y., Salles M. A. V., Liu Y., Kalinowski M. Data management in microservices: State of the practice, challenges, and research directions. *arXiv*. 2021. URL: <https://arxiv.org/abs/2103.00170>.
5. Gashi E., Hyseni D., Shabani I., Çiço B. The advantages of micro-frontend architecture for developing web applications. *Mediterranean Conference on Embedded Computing : матеріали міжнар. конф. (MECO 2024)*. 2024. С. 1–5.
6. Velepucha V., Flores P. A survey on microservices architecture: Principles, patterns and migration challenges. *IEEE Access*. 2023. T. 11. С. 339–358.
7. Falkevych V. G., Lisniak A. O. Client state management using backend for frontend pattern architecture in B2B segment. *Artificial Intelligence*. 2024. № 2. С. 49–60.
8. Waseem M., Liang P., Shahin M., Di Salle A., Márquez G. Design, monitoring, and testing of microservices systems: The practitioners' perspective. *Journal of Systems and Software*. 2021. T. 182.
9. Alkhodary S. The evaluation of using Backend-for-Frontend in a microservices environment : кваліфікаційна робота. LU-CS-EX. 2022.
10. von Allwörden T. M., Sosnowski M. Content and API acceleration using content delivery networks. *Seminar IITM SS 23*. 2023. С. 61–65.

11. Zhao J. T., Jing S. Y., Jiang L. Z. Management of API gateway based on micro-service architecture. *Journal of Physics: Conference Series*. 2018. T. 1087. № 3. C. 032032.
12. Oyeniran O. C., Adewusi A. O., Adeleke A. G., Akwawa L. A., Azubuko C. F. Microservices architecture in cloud-native applications: Design patterns and scalability. *International Journal of Advanced Research and Interdisciplinary Scientific Endeavours*. 2024. T. 1. № 2. C. 92–106.
13. Pandiya D. K., Charankar N. Testing strategies for microservices and APIs. *AI and Microservices: Integrating AI into API Design and Distributed Microservice Architecture*. Berkeley, CA : Apress, 2025. C. 87–120.
14. Rattanukul P., Makaranond C., Watanakulcharus P., Ragkhitwetsagul C., Nearunchorn T., Visoottiviseth V., Choetkiertikul M., Sunetnanta T. Microuisity: A testing tool for backends for frontends (BFF) microservice systems. *International Conference on Program Comprehension : матеріали міжнар. конф. (ICPC 2023)*. 2023. C. 74–78.
15. Abdelfattah A., Cerny T. Filling the gaps in microservice frontend communication: Case for new frontend patterns. *International Conference on Cloud Computing and Services Science : матеріали міжнар. конф. (CLOSER 2023)*. 2023.
16. Hidayat D. C., Atmaja I. K. J., Sarasvananda I. B. G. Analysis and comparison of micro frontend and monolithic architecture for web applications. *Jurnal Galaksi*. 2024. T. 1. № 2. C. 92–100.
17. Wanjala S. T. A framework for implementing micro frontend architecture. *International Journal of Web Engineering and Technology*. 2022. T. 17. № 4. C. 337–352.
18. Falkevych V. G., Lisniak A. O. Methodology of cache invalidation in microservices architecture of web applications. *Вчені записки Таврійського національного університету імені В. І. Вернадського. Серія: Технічні науки*. 2023.
19. Falkevych V. G., Lisniak A. O. Cache invalidation based on a declarative approach for separating business logic of microservices from cache update rules. *Eastern-European Journal of Enterprise Technologies*. 2025. № 2. C. 68–74.

20. Kunštnár V., Podhorský P. Micro frontend architecture. Zooming Innovation in Consumer Technologies Conference : матеріали міжнар. конф. (ZINC 2024). 2024. С. 124–129.
21. Faridi M. T., Singh K., Soni K., Negi S. Memcached vs Redis caching optimization comparison using machine learning. International Conference on Automation, Computing and Renewable Systems : матеріали міжнар. конф. (ICACRS 2023). 2023. С. 1153–1159.
22. Carra D., Michiardi P. Memory partitioning in memcached: An experimental performance analysis. IEEE International Conference on Communications : матеріали міжнар. конф. (ICC 2014). 2014. С. 1154–1159.
23. Zhang H., Kallas K., Pavlatos S., Alur R., Angel S., Liu V. MuCache: A general framework for caching in microservice graphs. USENIX Symposium on Networked Systems Design and Implementation : матеріали міжнар. симпозіуму (NSDI 24). 2024. С. 221–238.
24. Chen S., Tang X., Wang H., Zhao H., Guo M. Towards scalable and reliable in-memory storage system: A case study with Redis. IEEE Trustcom/BigDataSE/ISPA : матеріали міжнар. конф. 2016. С. 1660–1667.
25. Liu Q., Yuan H. A high performance memory key-value database based on Redis. Journal of Computers. 2019. Т. 14. № 3. С. 170–183.
26. Fukuda E. S., Inoue H., Takenaka T., Kim D., Sadahisa T., Asai T., Motomura M. Caching memcached at reconfigurable network interface. International Conference on Field Programmable Logic and Applications : матеріали міжнар. конф. (FPL 2014). 2014. С. 1–6.
27. Ji Z., Ganchev I., O'Droma M., Ding T. A distributed Redis framework for use in the UCWW. International Conference on Cyber-Enabled Distributed Computing and Knowledge Discovery : матеріали міжнар. конф. 2014. С. 241–244.
28. Ji Y., Zhou X., Zhou Y., Wang K. The case of transparent cache invalidation in web applications. arXiv. 2023. URL: <https://arxiv.org/abs/2311.02384>.
29. Zhu M., Shahab A., Katsarakis A., Grot B. Invalidate or update? Revisiting coherence for tomorrow's cache hierarchies. International Conference on Parallel

Architectures and Compilation Techniques : матеріали міжнар. конф. (PACT 2021). 2021. С. 226–241.

30. Zhu M. Adaptive invalidate/update coherence protocol with accurate prediction for all-private cache hierarchies : магістерська робота / University of Edinburgh. Edinburgh, 2023. С. 1–79.

31. Łopuszański J. Algorithm for invalidation of cached results of queries to a single table. arXiv. 2023. URL: <https://arxiv.org/abs/2310.15360>.

32. Kumar S., Dash Y. Caching techniques in edge computing and challenges. International Conference on Advancement in Electronics and Communication Engineering : матеріали міжнар. конф. (AECE 2023). 2023. С. 360–367.

33. Kim Y., Chen Y., Lee Y., Peng L., Hong S. Proactively invalidating dead blocks to enable fast writes in STT-MRAM caches. IEEE Access. 2022. Т. 10. С. 29419–29431.

34. Sun X., Wang D., Zhang W., Lou G., Wang J., Yadav R. Minimizing service latency through image-based microservice caching and randomized request routing in mobile edge computing. IEEE Internet of Things Journal. 2024. Т. 11. № 18. С. 30054–30068.

35. Li L., Ye C., Zhou H. Cache replacement algorithm based on dynamic constraints in microservice platform. International Conference on Service Science : матеріали міжнар. конф. (ICSS 2022). 2022. С. 167–174.

36. Falkevych V. G., Lisniak A. O. Optimization of infrastructure deployment for multi-frontends in monorepo. Artificial Intelligence. 2025. № 2. С. 63–70.

37. Deepak R. D., Swarnalatha P. Continuous integration–continuous security–continuous deployment pipeline automation for application software. International Journal of Computer Science and Software Engineering. 2019. Т. 8. № 10. С. 247–253.

38. Brousse N. The issue of monorepo and polyrepo in large enterprises. International Conference on the Art, Science, and Engineering of Programming : матеріали міжнар. конф. 2019. С. 1–4.

39. Zampetti F., Geremia S., Bavota G., Di Penta M. CI/CD pipelines evolution and restructuring: A qualitative and quantitative study. International Conference on

Software Maintenance and Evolution : матеріали міжнар. конф. (ICSME 2021). 2021. С. 471–482.

40. Mustyala A. CI/CD pipelines in Kubernetes: Accelerating software development and deployment. EPH-International Journal of Science and Engineering. 2022. Т. 8. № 3. С. 1–11.

41. Ghaleb T. A., Abduljalil O., Hassan S. CI/CD configuration practices in open-source Android apps: An empirical study. arXiv. 2024. URL: <https://arxiv.org/abs/2411.06077>.

42. Reddy S., Catharine A., Shanthamalar J. J. Efficient application deployment: GitOps for faster and secure CI/CD cycles. Advances in Modern Age Technologies for Health and Engineering Science : матеріали міжнар. конф. (AMATHE 2024). 2024. С. 1–7.

43. Chittala S. Enhancing developer productivity through automated CI/CD pipelines: A comprehensive analysis. International Journal of Computer Engineering and Technology. 2024. Т. 15. № 5. С. 882–891.

44. Slagsvold A. F. Exploring CI/CD pipelines for cloud infrastructure deployment: Can one increase efficiency through amalgamation? : магістерська робота / Oslo Metropolitan University. Oslo, 2023.

45. Singireddy S. R. Analysis of continuous integration/continuous deployment (CI/CD) pipelines for automated cloud infrastructure management. International Journal of Core Engineering & Management. 2024. Т. 7. № 10. С. 45–57.

46. Bournassenko G. On queueing theory for large-scale CI/CD pipelines optimization. arXiv. 2025. URL: <https://arxiv.org/abs/2504.18705>.

47. Загорулько А. В., Павленко В. І. Архітектура ефективної CI/CD-системи для програмних рішень, заснованих на MSA. Інфокомунікаційні та комп'ютерні технології. 2024. № 1(07). С. 56–60.

48. Cowell C., Lotz N., Timberlake C. Automating DevOps with GitLab CI/CD pipelines. Birmingham : Packt Publishing, 2023.

49. Detti A. Microservices from cloud to edge: An analytical discussion on risks, opportunities and enablers. IEEE Access. 2023. Т. 11. С. 49924–49942.

50. Iyer R., De V., Illikkal R., Koufaty D., Chitlur B., Herdrich A., Khellah M., Hamzaoglu F., Karl E. Advances in microprocessor cache architectures over the last 25 years. *IEEE Micro*. 2021. T. 41. № 6. С. 78–88.
51. Raharjo A. B., Andyartha P. K., Wijaya W. H., Purwananto Y., Purwitasari D., Juniarta N. Reliability evaluation of microservices and monolithic architectures. *Computer Engineering, Network, and Intelligent Multimedia : матеріали міжнар. конф. (CENIM 2022)*. 2022. С. 1–7.
52. Stefanovska E., Trajkovik V. Evaluating micro frontend approaches for code reusability. *ICT Innovations : матеріали міжнар. конф.* 2022. С. 93–106.
53. Seth D., Singh A., Panyam S. Distributed caching challenges and strategies in enterprise applications. *International Research Journal of Modernization in Engineering Technology and Science*. 2023. T. 6.
54. Stünkel P., von Bargaen O., Rutle A., Lamo Y. GraphQL federation: A model-based approach. *Journal of Object Technology*. 2020. T. 19. № 2.
55. Vohra N., Manuaba I. B. K. Implementation of REST API vs GraphQL in microservice architecture. *Information Management and Technology : матеріали міжнар. конф. (ICIMTech 2022)*. 2022. С. 45–50.
56. Andersson T., Reinholdsson H. REST API vs GraphQL: A literature and experimental study. 2021.
57. What is WunderGraph. GitHub repository. 2023. URL: <https://github.com/wundergraph/wundergraph>.
58. Bergvall A. Performance in Apollo Federation: A controlled experiment evaluating the effects of execution strategies and number of subgraphs. 2022.
59. Queirós L. D. O. Explorar performance com Apollo Federation. 2023.
60. Bergvall A. Performance in Apollo Federation: A controlled experiment evaluating the effects of execution strategies and number of subgraphs. 2022.
61. Scese M. D. Design and implementation of a GraphQL mesh gateway: Federating API endpoints based on a defined data model. 2023.

62. Kolczynski N. Developing a GraphQL mesh federated API gateway: Rapid integration of new endpoints into a predefined schema : магістерська робота / Michigan Technological University. 2025.
63. Chaudhary V., Sharma M., Rella B. P. R., Sharma G., Jaiswal A., Singh Y. A comparative analysis of REST and GraphQL. Data Analytics & Management : матеріали міжнар. конф. 2025. С. 173–192.
64. Khan R., Noor Mian A. Sustainable IoT sensing applications development through GraphQL-based abstraction layer. Electronics. 2020. Т. 9. № 4. С. 564.
65. Erigha E. D., Obuse E., Okare B. P., Uzoka A. C., Owoade S., Ayanbode N. Optimizing GraphQL server performance with intelligent request batching, query deduplication, and caching mechanisms. 2021.
66. Dos Santos J. S., Azevedo L. G., Soares E. F., Thiago R. M., da Silva V. T. Analysis of tools for REST contract specification in Swagger/OpenAPI. ICEIS : матеріали міжнар. конф. 2020. С. 201–20.
67. Wittern E., Cha A., Laredo J. A. Generating GraphQL wrappers for REST-like APIs. International Conference on Web Engineering : матеріали міжнар. конф. 2018. С. 65–83.
68. Sferruzza D. Top-down model-driven engineering of web services from extended OpenAPI models. Automated Software Engineering : матеріали міжнар. конф. 2018. С. 940–943.
69. Carvalho T. E. A. D. Generation of web API definition files using a single platform according to the design-first approach : дис. ... / 2021.
70. Oleshchenko L., Burchak P. Web application state management performance optimization methods. Computer Science, Engineering and Education Applications : матеріали міжнар. конф. 2023. С. 59–74.
71. Le T. Comparison of state management solutions between Context API and Redux Hook in ReactJS. 2021.
72. Kato D. Micro state management with React hooks. Birmingham : Packt Publishing, 2022.

73. Marco V., Farias K. Exploring the technologies and architectures used to develop micro-frontend applications: A systematic mapping and emerging perspectives. SSRN. 2024.
74. Genne S. Micro frontend architecture: Engineering modular solutions for enterprise web applications. *Journal of Engineering and Computer Sciences*. 2025. T. 4. № 7. C. 754–760.
75. Moraes F., Campos G., Almeida N., Affonso F. Micro frontend-based development: Concepts, motivations, implementation principles, and an experience report. *Enterprise Information Systems : матеріали міжнар. конф.* 2024.
76. Parab A. Micro frontends architecture: Breaking down monolithic frontend applications. *Journal of Computer Science and Technology Studies*. 2025. T. 7. № 6. C. 1014–1023.
77. Petcu A., Frunzete M., Stoichescu D. A. Benefits, challenges, and performance analysis of a scalable web architecture based on micro-frontends. *Scientific Bulletin, Series C*. 2023. T. 85. № 3. C. 319–334.
78. Antunes F., Lima M. J. D., Araújo M. A. P., Taibi D., Kalinowski M. Investigating benefits and limitations of migrating to a micro-frontends architecture. *arXiv*. 2024. URL: <https://arxiv.org/abs/2407.15829>.
79. Nikulina O., Khatsko K. Method of converting the monolithic architecture of a front-end application to microfrontends. *Bulletin of National Technical University “KhPI”*. 2023. № 2(10). C. 79–84.
80. Nurzhan M., Amandyk K. Micro-frontends integration for enhanced user experience in web applications. *Universum: технічні науки*. 2025. № 8(6). C. 38–42.
81. McFarlane T. Managing state in React applications with Redux. 2019.
82. Yao Y., Xia J. Analysis and research on the performance optimization of web application systems in high concurrency environments. *Information Technology, Networking, Electronic and Automation Control : матеріали міжнар. конф.* 2016. C. 321–326.
83. Afonso D., Mestre R. State management with React Query. Birmingham : Packt Publishing, 2023.

84. Yang J., Yue Y., Vinayak R. Segcache: A memory-efficient and scalable in-memory key-value cache for small objects. *USENIX Symposium on Networked Systems Design and Implementation*. 2021. C. 503–518.
85. Almeida D., Lopes M., Saraiva L., Abbasi M., Martins P., Silva J., Váz P. Performance comparison of Redis, Memcached, MySQL, and PostgreSQL. *Smart Technologies for Smart Nation : матеріали міжнар. конф.* 2023. C. 902–907.
86. Chopade R., Pachghare V. A data recovery technique for Redis using internal dictionary structure. *Forensic Science International: Digital Investigation*. 2021. T. 38. C. 301218.
87. Kanthed S. Redis vs. Memcached in microservices architectures: Caching strategies. 2023.
88. Joshi P. K. Redis cache optimization for payment gateways in the cloud. *International Journal of Emerging Trends in Computer Science and Information Technology*. 2023. T. 4. № 2. C. 28–36.
89. Iurchenko A. Optimization of microservices architecture performance in high-load systems. *The American Journal of Engineering and Technology*. 2025. T. 7. № 5. C. 123–132.
90. Fitzpatrick B. Distributed caching with memcached. *Linux Journal*. 2004. № 124. C. 5.
91. Harris A. Distributed caching via memcached. *Pro ASP.NET 4 CMS*. Berkeley, CA : Apress, 2010. C. 165–196.
92. Salhi H., Odeh F., Nasser R., Taweel A. Open source in-memory data grid systems: Benchmarking Hazelcast and Infinispan. *Performance Engineering : матеріали міжнар. конф.* 2017. C. 163–164.
93. Gencer C., Topolnik M., Ďurina V., Demirci E., Kahveci E. B., Lukáš A., Bartók J., Gierlach G., Hartman F., Yılmaz U., Doğan M. Hazelcast Jet: Low-latency stream processing at the 99.99th percentile. *arXiv*. 2021. URL: <https://arxiv.org/abs/2103.10169>.

94. Maxim B., Tetiana K. Hazelcast vs Ignite: Opportunities for Java programmers. *International Journal of Computer Science & Network Security*. 2022. T. 22. № 2. С. 406–412.
95. Tian K. L., Cai J., Ooi B. C. An evaluation of cache invalidation strategies in wireless environments. *IEEE Transactions on Parallel and Distributed Systems*. 2001. T. 12. № 8. С. 789–807.
96. Bai X., Junqueira F. P. Online result cache invalidation for real-time web search. *SIGIR : матеріали міжнар. конф.* 2012. С. 641–650.
97. Sun H., Cui Q., Huang J., Qin X. NCache: A machine-learning cache management scheme for computational SSDs. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*. 2022. T. 42. № 6. С. 1810–1823.
98. Jani Y. Implementing continuous integration and continuous deployment in modern software development. *International Journal of Science and Research*. 2023. T. 12. № 6. С. 2984–2987.
99. Thatikonda V. K. Beyond the buzz: A journey through CI/CD principles and best practices. *European Journal of Theoretical and Applied Sciences*. 2023. T. 1. № 5. С. 334–340.
100. Koppanati P. K. Best practices for version control and release management using GitLab CI/CD. *European Journal of Advances in Engineering and Technology*. 2022. T. 9. № 11. С. 141–146.
101. Chintale P. DevOps design pattern: Implementing DevOps best practices for secure and reliable CI/CD pipeline. New Delhi : BPB Publications, 2023.
102. Putra A. M., Kabetta H. Implementation of DevSecOps by integrating static and dynamic security testing in CI/CD pipelines. *Computer Science and Information Technology : матеріали міжнар. конф.* 2022. С. 1–6.
103. Duranti E. Microservice oriented pipeline architectures : дис. ... / Politecnico di Torino. 2020.
104. Brown K., Woolf B., Yoder J. Cloud application architecture patterns. Sebastopol, CA : O'Reilly Media, 2025. 650 с.
105. Guide to caching in Java. URL: <https://www.baeldung.com/caching-series>.

106. Tabunshchik G., Parkhomenko A., Subbotin S., Zeleneva I., Holub T., Kapliienko T. Framework for the online education with the distributed educational resources. *Interactive Collaborative Learning* : матеріали міжнар. конф. 2023. С. 247–254.

107. Plecinski P., Bokla N., Klymkovych T., Melnyk M., Zabierowski W. Comparison of representative microservices technologies in terms of performance. *Sensors*. 2022. Т. 22. № 20. С. 7759.

108. Дмитренко О. А. Мікросервісна архітектура та її задачі на прикладах з реального життя. *Перспективи телекомунікацій* : матеріали міжнар. наук.-техн. конф. 2023. С. 239–242.

109. Burlachenko I., Zavorotnii D. Мікросервісна архітектура системи адаптивної автентифікації користувачів фінансових установ. *Computer-Integrated Technologies: Education, Science, Production*. 2025. № 60. С. 429–440.

ДОДАТКИ

ДОДАТОК А.

Приклад реалізації арі шлюза

```
// Реєстрація бізнес-сервісів у модулі API Шлюза
@Module({
  imports: [
    ClientsModule.register([
      {
        name: 'BOOK_SERVICE',
        transport: Transport.REDIS,
        options: {
          host: HOST_REDIS,
          port: PORT_REDIS,
        },
      },
    ]),
  ],
  controllers: [GatewayController],
})

// Контролер API Шлюза для обробки запитів від бізнес-сервісів
@Controller()
export class GatewayController {
  private serviceClients: Record<string, ClientProxy>;

  constructor(
    // Реєстрація бізнес-сервісів
    @Inject('BOOK_SERVICE') private readonly bookServiceClient:
ClientProxy,
  ) {
    this.serviceClients = {
```

```

        book: this.bookServiceClient,
    };
}
// Наприклад: POST запит /book/get для отримання списку книг
(сервіс та
// тип операції)
// Наприклад: POST request /book/create для створення нової
книги
// Наприклад: POST request /book/getOne для отримання однієї
книги за
// вибраними фільтрами
@Post('/:service/:operation')
async gateway(
    @Param('service') service: string,
    @Param('operation') operation: string,
    @Body() payload: any,
) {
    if (!this.serviceClients?.[service])
        throw new HttpException(
            exceptions.OPERATION_IS_NOT_ALLOWED,
            HttpStatus.UNPROCESSABLE_ENTITY,
        );

    return firstValueFrom(
        this.serviceClients?.[service].send(
            { cmd: `${service}_${operation}` },
            { payload },
        ),
    );
}
}

```

ДОДАТОК Б

Приклад реалізації кеш-сервіса

```

// Модуль Кеш-сервісу
@Module({
  imports: [RedisModule.forRoot(HOST_REDIS, PORT_REDIS)],
  controllers: [CacheController],
  providers: [CacheService],
  exports: [CacheService],
})
class CacheModule {}
// Реєстрація Кеш модуля з типом транспорту Redis
async function bootstrap() {
  const app = await
NestFactory.createMicroservice<MicroserviceOptions>(
  CacheModule,
  {
    transport: Transport.REDIS,
    options: {
      host: HOST_REDIS,
      port: PORT_REDIS,
    },
  },
);
  await app.listen();
}

```

ДОДАТОК В

Приклад реалізації бізнес-сервіса

```
// Модуль Бізнес-сервісу
@Module({
  imports: [
    ClientsModule.register([
      {
        name: 'CACHE_SERVICE',
        transport: Transport.REDIS,
        options: {
          host: HOST_REDIS,
          port: PORT_REDIS,
        },
      },
    ]),
  ],
  providers: [BookService],
  controllers: [BookController],
})
class BookModule {}

// Реєстрація модуля Бізнес-сервіса з типом транспорту Redis
async function bootstrap() {
  const app = await
NestFactory.createMicroservice<MicroserviceOptions>(
  BookModule,
  {
    transport: Transport.REDIS,
    options: {
      host: HOST_REDIS,
      port: PORT_REDIS,
    },
  },
),
```

```

    );
    await app.listen();
}

// Контролер Бізнес-сервіса
@Controller()
export class BookController {
    constructor(
        @Inject('CACHE_SERVICE')          private          readonly
cacheServiceClient: ClientProxy,
        private readonly bookService: BookService,
    ) {}

    @MessagePattern({ cmd: BOOK_EVENTS.BOOK_GET })
    // Опис методу для отримання списку книг
    async getBooks({ payload }: PayloadDto<GetBooksDto>):
Promise<Book[]> {
        // Отримання декларативного опису для інвалідації кешу для даного
сервіса
        const cacheConfig = this.bookService.getBookConfig();
        // Перевірка даних з кешу
        const dataFromCache = await firstValueFrom(
            this.cacheServiceClient.send(
                { cmd: CACHE_EVENTS.CACHE_CHECK },
                {
                    key: BOOK_EVENTS.BOOK_GET,
                    payload,
                },
            ),
        );
        if (dataFromCache) return dataFromCache;
        // Отримання даних з бази даних через сервіс для роботи з моделью
        const books = await this.bookService.getBooks();

        // Виклик сервісу для роботи з кешем з подальшою передачею
інформації
    }
}

```

```
// щодо особливостей оновлення даних у кеші для даного сервісу та
запиту
```

```
    this.cacheServiceClient
      .send(
        { cmd: CACHE_EVENTS.CACHE_RUN_INSTRUCTIONS },
        {
          cmd: INSTRUCTION_OPERATIONS.GET,
          instructions: cacheConfig,
          payload,
          data: books,
        },
      )
      .subscribe((res) => {
        console.log('res', res);
      });
    return books;
  }
}
```

```
@MessagePattern({ cmd: BOOK_EVENTS.BOOK_CREATE })
// Метод створення нового запису у базі
async createBook({
  payload,
}: PayloadDto<Book>): Promise<SuccessWithoutDataResponseDto> {
  // Отримання декларативного опису для інвалідації кешу для даного
сервіса
```

```
    const cacheConfig = this.bookService.getBookConfig();
  // Створення нового запису у базі
    const newBook = await this.bookService.createBook(payload);
  // Виклик сервісу для роботи з кешем з подальшою передачею
інформації
  // щодо особливостей оновлення даних у кеші для даного сервісу та
запиту
```

```
    this.cacheServiceClient.send(
      { cmd: CACHE_EVENTS.CACHE_RUN_INSTRUCTIONS },
      {
        cmd: INSTRUCTION_OPERATIONS.CREATE,
```

```

        instructions: cacheConfig,
        data: newBook,
    },
);
return successEmptyResponse;
}

@MessagePattern({ cmd: BOOK_EVENTS.BOOK_REMOVE })
// Метод видалення запису з бази
async removeBook({
    payload,
}: PayloadDto<{
    slug: string;
}>): Promise<SuccessWithoutDataResponseDto> {
    // Отримання декларативного опису для інвалідації кешу для даного
сервіса
    const cacheConfig = this.bookService.getBookConfig();
    // Видалення запису з бази
    await this.bookService.removeBookBySlug(payload.slug);
    // Виклик сервісу для роботи з кешем з подальшою передачею
інформації
    // щодо особливостей оновлення даних у кеші для даного сервісу та
запиту
    this.cacheServiceClient.send(
        { cmd: CACHE_EVENTS.CACHE_RUN_INSTRUCTIONS },
        {
            cmd: INSTRUCTION_OPERATIONS.REMOVE,
            instructions: cacheConfig,
        },
    );
    return successEmptyResponse;
}
}

```

ДОДАТОК Г

Приклад декларативного опису інвалідації кешу

```

# config.yaml
title: 'Інструкції інвалідації кешу'
service: 'Book'
cache_keys:
  book_get: "book_get"
  book_get_one: "book_get_one"
operations:
  get:
    description: "Додати книги до кешу"
    actions:
      # Операція додавання/оновлення даних
      - operation: "add"
        # Ціль у кеші - масив даних
        target: "list"
        description: "Додати книги до кешу як список"
        # Базовий ключ, який ідентифікує сутність
        key: "book_get"
        # Тип дії встановлення з повним перезаписом наявних даних
        action_type: "set"
      # Операція додавання/оновлення даних
      - operation: "add"
        # Ціль у кеші - елемент з унікальним слагом
        target: "slug"
        description: "Додати книги до кешу за слагом"
        # Базовий ключ, який ідентифікує сутність
        key: "book_get_one"
        # Тип дії встановлення з повним перезаписом наявних даних
        action_type: "set"
    getOne:
      description: "Додати книгу до кешу"
      actions:

```

```

# Операція додавання/оновлення даних
- operation: "add"
# Ціль у кеші - елемент з унікальним слагом
target: "slug"
description: "Додати книгу до кешу у список"
# Базовий ключ, який ідентифікує сутність
key: "book_get_one"
# Тип дії встановлення з повним перезаписом наявних даних
action_type: "set"
create:
description: "Add a book to cache upon creation"
actions:
# Операція додавання/оновлення даних
- operation: "add"
# Ціль у кеші - елемент з унікальним слагом
target: "slug"
description: "Додати до кешу запис книги за слагом"
# Базовий ключ, який ідентифікує сутність
key: "book_get_one"
# Тип дії встановлення або оновлення наявних даних
action_type: "update"
remove:
description: "Прибрати книгу з кешу після операції видалення"
actions:
# Операція видалення даних
- operation: "remove"
# Ціль у кеші - елемент з унікальним слагом
target: "slug"
description: "Видалити запис книги з кешу за слагом"
# Базовий ключ, який ідентифікує сутність
key: "book_get_one"
# Тип дії оновленням наявних даних
action_type: "update"
notes:
- "'book_list' ключ зберігає список всіх книг за слагом"

```

- "Кожна книга зберігається окремо як 'book:{slug}' з детальною інформацією."

- "Під час додавання книги оновлюється список книг та додається книга за слагом у кеші"

- "Під час видалення книги, також оновлюється список книг у кеші та видаляється за слагом"

ДОДАТОК І

Приклад приклад налаштування Gitlab CI

```
stages:
  - install_dependencies
  - build_packages
  - test_packages
  - build_apps
  - assemble

variables:
  NODE_ENV: "production"
  NODE_IMAGE: dockerhub.timeweb.cloud/node:20.9.0
  FF_USE_FASTZIP: "true"
  ARTIFACT_COMPRESSION_LEVEL: "slowest"
  CACHE_COMPRESSION_LEVEL: "slowest"

default:
  cache: &global_cache
    key: ${CI_COMMIT_SHORT_SHA}
    paths:
      - node_modules
      - "packages/*/node_modules"
      - "packages/*/dist"
      - "apps/*"
    policy: pull-push

install_dependencies:
  image: $NODE_IMAGE
  needs: []
  variables:
    NODE_ENV: $NODE_ENV
  resource_group: $CI_COMMIT_REF_SLUG
  interruptible: true
  script:
    - yarn install
  stage: install_dependencies
```

```
build_package1:
  image: $NODE_IMAGE
  needs:
    - job: install_dependencies
  variables:
    NODE_ENV: $NODE_ENV
  resource_group: $CI_COMMIT_REF_SLUG
  interruptible: true
  script:
    - yarn workspace package1 run build
  stage: build_packages
build_package2:
  image: $NODE_IMAGE
  needs:
    - job: install_dependencies
  variables:
    NODE_ENV: $NODE_ENV
  resource_group: $CI_COMMIT_REF_SLUG
  interruptible: true
  script:
    - yarn workspace package2 run build
  stage: build_packages
build_package3:
  image: $NODE_IMAGE
  needs:
    - job: install_dependencies
  variables:
    NODE_ENV: $NODE_ENV
  resource_group: $CI_COMMIT_REF_SLUG
  interruptible: true
  script:
    - yarn workspace package3 run build
  stage: build_packages
test_package1:
  image: $NODE_IMAGE
  needs:
```

```

    - job: build_package1
variables:
    NODE_ENV: $NODE_ENV
    resource_group: $CI_COMMIT_REF_SLUG
    interruptible: true
script:
    - yarn workspace package1 run test
stage: test_packages
test_package2:
    image: $NODE_IMAGE
    needs:
        - job: build_package2
    variables:
        NODE_ENV: $NODE_ENV
        resource_group: $CI_COMMIT_REF_SLUG
        interruptible: true
    script:
        - yarn workspace package2 run test
    stage: test_packages
test_package3:
    image: $NODE_IMAGE
    needs:
        - job: build_package3
    variables:
        NODE_ENV: $NODE_ENV
        resource_group: $CI_COMMIT_REF_SLUG
        interruptible: true
    script:
        - yarn workspace package3 run test
    stage: test_packages
build_app1:
    stage: build_apps
    needs:
        - job: test_package1
        - job: test_package2
        - job: test_package3

```

```
image: $NODE_IMAGE
resource_group: $CI_COMMIT_REF_SLUG
interruptible: true
script:
  - yarn workspace app1 run build
build_app2:
  stage: build_apps
  needs:
    - job: test_package1
    - job: test_package2
    - job: test_package3
  image: $NODE_IMAGE
  resource_group: $CI_COMMIT_REF_SLUG
  interruptible: true
  script:
    - yarn workspace app2 run build
assemble:
  stage: assemble
  needs:
    - job: build_app1
    - job: build_app2
  resource_group: $CI_COMMIT_REF_SLUG
  interruptible: true
  script:
    - sleep 120 && echo "Image is uploaded!"
```

ДОДАТОК Д

Результати експерименту з послідовною збіркою пакетів та додатків

Vitaliy Falkevich / deploy-optimization / Pipelines

All 1FinishedBranchesTags

View analyticsClear runner cachesNew pipeline

Status = Running X

Show Pipeline ID

Status	Pipeline	Created by	Stages	Actions
Running	default CI #1734956634 main e4e4fa56 latestbranch		Stage: build_packages build_packages	

Vitaliy Falkevich / deploy-optimization / Jobs / #9521255827

Search visible log output

00:04

20 Failed to extract cache

21 Executing "step_script" stage of the job script

22 Using docker image sha256:03501b0671a4e8450444678df0aad9c78ecce5fe2f4455daae571bd879f6d965 for dockerhub.timeweb.cloud/node:20.9.0 with digest dockerhub.timeweb.cloud/node@sha256:146bbe4eae99ae885be2a0a767f63a4b96032141d70d80e590f10e0d7ebabcb ...

23 \$ yarn install

24 yarn install v1.22.19

25 [1/4] Resolving packages...

26 [2/4] Fetching packages...

27 [3/4] Linking dependencies...

28 [4/4] Building fresh packages...

29 Done in 2.18s.

30 Saving cache for successful job

31 Creating cache e4e4fa56-2-protected...

32 node_modules: found 640 matching artifact files and directories

33 packages/*/node_modules: found 15 matching artifact files and directories

34 WARNING: packages/*/dist: no matching files. Ensure that the artifact path is relative to the working directory (/builds/vitaliifalkevich/deploy-optimization)

35 apps/*: found 24 matching artifact files and directories

36 Uploading cache.zip to https://storage.googleapis.com/gitlab-com-runners-cache/project/68361904/e4e4fa56-2-protected

37 Created cache

38 Cleaning up project directory and file based variables

39 Job succeeded

Duration: 47 seconds

Finished: 2 minutes ago

Queued: 0 seconds

Timeout: 1h (from project)

Runner: #12270807 (j1aLDqxS) 1-blue.saas-linux-small-amd64.runners-manager.gitlab.com/default

Source: Web

Commit e4e4fa56 default CI

Pipeline #1734956634 Running for main

install_dependencies

Related jobs

→ install_dependencies

Vitaliy Falkevich / deploy-optimization / Jobs / #9521255830

Search visible log output





```
23 $ yarn workspace package1 run build
24 yarn workspace v1.22.19
25 yarn run v1.22.19
26 $ echo 'start building package 1...' && sleep 180 && echo 'package 1 is built!'
27 start building package 1...
28 package 1 is built!
29 Done in 180.06s.
30 Done in 180.31s.
31 $ yarn workspace package2 run build
32 yarn workspace v1.22.19
33 yarn run v1.22.19
34 $ echo 'start building package 2...' && sleep 180 && echo 'package 2 is built!'
35 start building package 2...
36 package 2 is built!
37 Done in 180.05s.
38 Done in 180.30s.
39 $ yarn workspace package3 run build
40 yarn workspace v1.22.19
41 yarn run v1.22.19
42 $ echo 'start building package 3...' && sleep 180 && echo 'package 3 is built!'
43 start building package 3...
44 package 3 is built!
45 Done in 180.06s.
46 Done in 180.31s.
47 Saving cache for successful job
48 Creating cache e4e4fa56-2-protected...
49 node_modules: found 640 matching artifact files and directories
50 packages/*/node_modules: found 15 matching artifact files and directories
51 WARNING: packages/*/dist: no matching files. Ensure that the artifact path is relative to the working di
52 rectory (/builds/vitaliifalkevich/deploy-optimization)
53 apps/*: found 24 matching artifact files and directories
54 Uploading cache.zip to https://storage.googleapis.com/gitlab-com-runners-cache/project/68361904/e4e4fa5
55 6-2-protected
56 Created cache
57 Cleaning up project directory and file based variables
58 Job succeeded
```

Duration: 10 minutes 14 seconds
Finished: 17 seconds ago
Queued: 0 seconds
Timeout: 1h (from project) ⓘ
Runner: #12270807 (j1aLDqxS) 1-blue.saas-linux-small-amd64.runners-manager.gitlab.com/default
Source: Web

Commit e4e4fa56 
default CI

Pipeline #1734956634  Running for main 
build_packages

Related jobs
→  build_packages

Vitaliy Falkevich / deploy-optimization / Pipelines

All 1 Finished Branches Tags

View analytics Clear runner caches New pipeline

Status =  Running X

Show Pipeline ID

Status	Pipeline	Created by	Stages	Actions
	default CI #1734956634  main  e4e4fa56  latest branch		     Stage: test_packages  test_packages	 

Search visible log output

```
23 $ yarn workspace package1 run test
24 yarn workspace v1.22.19
25 yarn run v1.22.19
26 $ echo 'start testing package 1...' && sleep 60 && echo 'testing for package 1 completed successfully'
27 start testing package 1...
28 testing for package 1 completed successfully
29 Done in 60.05s.
30 Done in 60.29s.
31 $ yarn workspace package2 run test
32 yarn workspace v1.22.19
33 yarn run v1.22.19
34 $ echo 'start testing package 2...' && sleep 60 && echo 'testing for package 2 completed successfully'
35 start testing package 2...
36 testing for package 2 completed successfully
37 Done in 60.05s.
38 Done in 60.29s.
39 $ yarn workspace package3 run test
40 yarn workspace v1.22.19
41 yarn run v1.22.19
42 $ echo 'start testing package 3...' && sleep 60 && echo 'testing for package 3 completed successfully'
43 start testing package 3...
44 testing for package 3 completed successfully
45 Done in 60.05s.
46 Done in 60.29s.
47 Saving cache for successful job
48 Creating cache e4e4fa56-2-protected...
49 node_modules: found 640 matching artifact files and directories
50 packages/*/node_modules: found 15 matching artifact files and directories
51 WARNING: packages/*/dist: no matching files. Ensure that the artifact path is relative to the working di
52 rectorry (/builds/vitaliifalkevich/deploy-optimization)
53 apps/*: found 24 matching artifact files and directories
54 Uploading cache.zip to https://storage.googleapis.com/gitlab-com-runners-cache/project/68361984/e4e4fa5
55 6-2-protected
56 Created cache
57 Cleaning up project directory and file based variables
58 Job succeeded
```

Duration: 4 minutes 43 seconds

Finished: 16 seconds ago

Queued: 0 seconds

Timeout: 1h (from project)

Runner: #12270837 (J2nyww-s) 4-blue.saas-linux-small-amd64.runners-manager.gitlab.com/default

Source: Web

Commit e4e4fa56 default CI

Pipeline #1734956634 Running for main

test_packages

Related jobs

→ test_packages

All 1 Finished Branches Tags

View analytics Clear runner caches New pipeline

Status = Running

Show Pipeline ID

Status	Pipeline	Created by	Stages	Actions
Running	default CI #1734956634 main e4e4fa56 latest branch		Stage: build_apps build_apps	

Search visible log output

19

Downloading cache from <https://storage.googleapis.com/gitlab-com-runners-cache/project/68361904/e4e4fa56-2-protected> ETag="988f2be91470efd746e1421affe201e2"

20

Successfully extracted cache

21

Executing "step_script" stage of the job script

22

Using docker image sha256:03501b0671a4e8450444678df0aad9c78ecce5fe2f4455daae571bd879fd965 for dockerhub.b.timeweb.cloud/node:20.9.0 with digest dockerhub.timeweb.cloud/node@sha256:146bbe4eae99ae885be2a0a767f63a4b96032141d70d80e590f10e0d7ebabcb ...

23

\$ yarn workspace app1 run build

24

yarn workspace v1.22.19

25

yarn run v1.22.19

26

\$ echo 'start building app1...' && sleep 240 && echo 'app1 is built!'

27

start building app1...

28

app1 is built!

29

Done in 240.05s.

30

Done in 240.30s.

31

\$ yarn workspace app2 run build

32

yarn workspace v1.22.19

33

yarn run v1.22.19

34

\$ echo 'start building app2...' && sleep 240 && echo 'app2 is built!'

35

start building app2...

36

app2 is built!

37

Done in 240.06s.

38

Done in 240.30s.

39

Saving cache for successful job

40

Creating cache e4e4fa56-2-protected...

41

node_modules: found 640 matching artifact files and directories

42

packages/*/node_modules: found 15 matching artifact files and directories

43

WARNING: packages/*/dist: no matching files. Ensure that the artifact path is relative to the working directory (/builds/vitaliifalkevich/deploy-optimization)

44

apps/*: found 24 matching artifact files and directories

45

Uploading cache.zip to <https://storage.googleapis.com/gitlab-com-runners-cache/project/68361904/e4e4fa56-2-protected>

46

Created cache

47

Cleaning up project directory and file based variables

48

Job succeeded

08:02

08:03

08:00

Duration: 9 minutes 24 seconds

Finished: just now

Queued: 0 seconds

Timeout: 1h (from project)

Runner: #12270837 (J2nyww-s) 4-blue.saas-linux-small-amd64.runners-manager.gitlab.com/default

Source: Web

Commit [e4e4fa56](#) default CI

Pipeline #1734956634 Running for main

build_apps

Related jobs

→ build_apps

All 1 Finished Branches Tags

View analytics Clear runner caches New pipeline

Status = Running

Show Pipeline ID

Status	Pipeline	Created by	Stages	Actions
Running	default CI #1734956634 main e4e4fa56 latest branch			
			Stage: assemble assemble	

Search visible log output

🗑️ ↺ ⬆️ ⬇️ ↗️

🔄

6 Pulling docker image ruby:3.1 ...
7 Using docker image sha256:62d2ffd1ba2a4f9df123711e18c1cbb361bdf212ce3bee3d6ebfd769961be79c for ruby:3.1 with digest ruby@sha256:aaf664741b8de45efe7c5db430430784c11af7d069027b2f008b74a53b8882ae ...
8 **Preparing environment** 00:05
9 Running on runner-xxurkrix-project-68361904-concurrent-0 via runner-xxurkrix-s-l-s-amd64-1742932482-337d2246...
10 **Getting source from Git repository** 00:01
11 Fetching changes with git depth set to 20...
12 Initialized empty Git repository in /builds/vitaliifalkevich/deploy-optimization/.git/
13 Created fresh repository.
14 Checking out e4e4fa56 as detached HEAD (ref is main)...
15 Skipping Git submodules setup
16 \$ git remote set-url origin "\${CI_REPOSITORY_URL}"
17 **Restoring cache** 00:01
18 Checking cache for e4e4fa56-2-protected...
19 Downloading cache from https://storage.googleapis.com/gitlab-com-runners-cache/project/68361904/e4e4fa56-2-protected ETag="7b6792d5678c6010bea95f2781a7b135"
20 Successfully extracted cache
21 **Executing "step_script" stage of the job script** 02:01
22 Using docker image sha256:62d2ffd1ba2a4f9df123711e18c1cbb361bdf212ce3bee3d6ebfd769961be79c for ruby:3.1 with digest ruby@sha256:aaf664741b8de45efe7c5db430430784c11af7d069027b2f008b74a53b8882ae ...
23 \$ sleep 120 && echo "Image is uploaded!"
24 Image is uploaded!
25 **Saving cache for successful job** 00:02
26 **Creating cache e4e4fa56-2-protected...**
27 node_modules: found 640 matching artifact files and directories
28 packages/*/node_modules: found 15 matching artifact files and directories
29 WARNING: packages/*/dist: no matching files. Ensure that the artifact path is relative to the working directory (/builds/vitaliifalkevich/deploy-optimization)
30 apps/*: found 24 matching artifact files and directories
31 Uploading cache.zip to https://storage.googleapis.com/gitlab-com-runners-cache/project/68361904/e4e4fa56-2-protected
32 **Created cache**
33 **Cleaning up project directory and file based variables** 00:01
34 **Job succeeded**

Duration: 2 minutes 33 seconds
Finished: just now
Queued: 0 seconds
Timeout: 1h (from project) ⓘ
Runner: #12270831 (XxUrkriX) 2-blue.saas-linux-small-amd64.runners-manager.gitlab.com/default
Source: Web

Commit e4e4fa56 🔄
default CI

Pipeline #1734956634 ✅ Passed for main 🔄
assemble

Related jobs
→ ✅ assemble

Vitaliy Falkevich / deploy-optimization / Pipelines

All 1 Finished Branches Tags

View analytics Clear runner caches New pipeline

Status = 🟢 Passed X

🔍 Show Pipeline ID ▾

Status	Pipeline	Created by	Stages	Actions
🟢 Passed 🕒 00:27:43 🕒 1 minute ago	default CI #1734956634 📁 main ➡ e4e4fa56 🔄 latest branch	👤	🟢 🟢 🟢 🟢 🟢	📄 ▾

ДОДАТОК Е

Результати експерименту з послідовною збіркою пакетів та додатків

Vitaliy Falkevich / deploy-optimization / Pipelines

All 1FinishedBranchesTags

View analyticsClear runner cachesNew pipeline

StatusRunning

Show Pipeline ID

Status	Pipeline	Created by	Stages	Actions
Running	Deploy optimization #1736653243 optimization 3aa73e7b latestbranch		Stage: install_dependencies install_dependencies	

Vitaliy Falkevich / deploy-optimization / Jobs / #9532304519

Search visible log output

14 Checking out 3aa73e7b as detached HEAD (ref is optimization)...

15 Skipping Git submodules setup

16 \$ git remote set-url origin "\${CI_REPOSITORY_URL}"

17 Restoring cache

18 Checking cache for 3aa73e7b-2-non_protected...

19 WARNING: file does not exist

20 Failed to extract cache

21 Executing "step_script" stage of the job script

22 Using docker image sha256:83501b0671a4e8450444678df0aad9c78ecce5fe2f4455daae571bd879f6d965 for dockerhub b.timeweb.cloud/node:20.9.0 with digest dockerhub.timeweb.cloud/node@sha256:146bbe4eaae99ae885be2a0a767f63a4b96832141d70d80e590f10e0d7ebabcb ...

23 \$ yarn install

24 yarn install v1.22.19

25 [1/4] Resolving packages...

26 [2/4] Fetching packages...

27 [3/4] Linking dependencies...

28 [4/4] Building fresh packages...

29 Done in 2.09s.

30 Saving cache for successful job

31 Creating cache 3aa73e7b-2-non_protected...

32 node_modules: found 640 matching artifact files and directories

33 packages/*/node_modules: found 15 matching artifact files and directories

34 WARNING: packages/*/dist: no matching files. Ensure that the artifact path is relative to the working directory (/builds/vitaliifalkevich/deploy-optimization)

35 apps/*: found 24 matching artifact files and directories

36 Uploading cache.zip to https://storage.googleapis.com/gitlab-com-runners-cache/project/68361904/3aa73e7b-2-non_protected

37 Created cache

38 Cleaning up project directory and file based variables

39 Job succeeded

Duration: 45 seconds

Finished: just now

Queued: 0 seconds

Timeout: 1h (from project)

Runner: #12270835 (zxwgkAP) 3-blue.saas-linux-small-amd64.runners-manager.gitlab.com/default

Source: Push

Commit 3aa73e7b

Deploy optimization

Pipeline #1736653243 Running for optimization

install_dependencies

Related jobs

→ install_dependencies

Vitaliy Falkevich / deploy-optimization / Pipelines

View analytics Clear runner caches New pipeline

Status = Running X Show Pipeline ID v

Status	Pipeline	Created by	Stages	Actions
Running	Deploy optimization #1736653243 optimization 3aa73e7b latest branch		Stage: build_packages - build_package1 - build_package2 - build_package3	

Vitaliy Falkevich / deploy-optimization / Pipelines

View analytics Clear runner caches New pipeline

Status = Running X Show Pipeline ID v

Status	Pipeline	Created by	Stages	Actions
Running	Deploy optimization #1736653243 optimization 3aa73e7b latest branch		Stage: test_packages - test_package1 - test_package3 - test_package2	

Vitaliy Falkevich / deploy-optimization / Pipelines

View analytics Clear runner caches New pipeline

Status = Running X Show Pipeline ID

Status	Pipeline	Created by	Stages	Actions
Running	Deploy optimization #1736653243 optimization 3aa73e7b latest branch		Stage: build_packages - build_package2 - build_package1 - build_package3	

Vitaliy Falkevich / deploy-optimization / Jobs / #9532304568

Search visible log output

```
16 $ git remote set-url origin "${CI_REPOSITORY_URL}"
17 Restoring cache
18 Checking cache for 3aa73e7b-2-non_protected...
19 Downloading cache from https://storage.googleapis.com/gitlab-com-runners-cache/project/68361904/3aa73e7b-2-non_protected ETag="ef2f715802504aaed0e3ef09298f6f2c"
20 Successfully extracted cache
21 Executing "step_script" stage of the job script
22 Using docker image sha256:03501b0671a4e845044678df0aad9c78ecce5fe2f445daae571bd879f6d965 for dockerhub.b.timeweb.cloud/node:20.9.0 with digest dockerhub.timeweb.cloud/node@sha256:146bbe4eae99ae885be2a0a767f63a4b96032141d70d80e590f10e0d7ebabcb ...
23 $ yarn workspace package1 run build
24 yarn workspace v1.22.19
25 yarn run v1.22.19
26 $ echo 'start building package 1...' && sleep 180 && echo 'package 1 is built!'
27 start building package 1...
28 package 1 is built!
29 Done in 180.04s.
30 Done in 180.28s.
31 Saving cache for successful job
32 Creating cache 3aa73e7b-2-non_protected...
33 node_modules: found 640 matching artifact files and directories
34 packages/*/node_modules: found 15 matching artifact files and directories
35 WARNING: packages/*/dist: no matching files. Ensure that the artifact path is relative to the working directory (/builds/vitaliifalkevich/deploy-optimization)
36 apps/*: found 24 matching artifact files and directories
37 Uploading cache.zip to https://storage.googleapis.com/gitlab-com-runners-cache/project/68361904/3aa73e7b-2-non_protected
38 Created cache
39 Cleaning up project directory and file based variables
40 Job succeeded
```

Duration: 3 minutes 52 seconds
Finished: 45 seconds ago
Queued: 0 seconds
Timeout: 1h (from project)
Runner: #12270835 (zxwgkjAP) 3-blue.saas-linux-small-amd64.runners-manager.gitlab.com/default
Source: Push

Commit 3aa73e7b
Deploy optimization

Pipeline #1736653243 Running for optimization
build_packages

Related jobs

- build_package2
- build_package1
- build_package3

Vitaliy Falkevich / deploy-optimization / Jobs / #9532304597

Search visible log output

16 \$ git remote set-url origin "\${CI_REPOSITORY_URL}"

17 Restoring cache 00:01

18 Checking cache for 3aa73e7b-2-non_protected...

19 Downloading cache from https://storage.googleapis.com/gitlab-com-runners-cache/project/68361984/3aa73e7b-2-non_protected ETag="ef2f715802504aaed0e3ef09298f6f2c"

20 Successfully extracted cache

21 Executing "step_script" stage of the job script 03:01

22 Using docker image sha256:03501b0671a4e8450444678df0aad9c78eccc5fe2f4455daae571bd879f6d965 for dockerhub.b.timeweb.cloud/node:20.9.0 with digest dockerhub.timeweb.cloud/node@sha256:146bbe4eae99ae885be2a0a767f63a4b96032141d70d80e590f10e0d7ebabcb ...

23 \$ yarn workspace package2 run build

24 yarn workspace v1.22.19

25 yarn run v1.22.19

26 \$ echo 'start building package 2...' && sleep 180 && echo 'package 2 is built!'

27 start building package 2...

28 package 2 is built!

29 Done in 180.05s.

30 Done in 180.30s.

31 Saving cache for successful job 00:03

32 Creating cache 3aa73e7b-2-non_protected...

33 node_modules: found 640 matching artifact files and directories

34 packages/*/node_modules: found 15 matching artifact files and directories

35 WARNING: packages/*/dist: no matching files. Ensure that the artifact path is relative to the working directory (/builds/vitaliifalkevich/deploy-optimization)

36 apps/*: found 24 matching artifact files and directories

37 Uploading cache.zip to https://storage.googleapis.com/gitlab-com-runners-cache/project/68361984/3aa73e7b-2-non_protected

38 Created cache

39 Cleaning up project directory and file based variables 00:01

40 Job succeeded

Duration: 4 minutes 38 seconds

Finished: 12 seconds ago

Queued: 0 seconds

Timeout: 1h (from project) ?

Runner: #12270835 (zxwgkjAP) 3-blue.saas-linux-small-amd64.runners-manager.gitlab.com/default

Source: Push

Commit 3aa73e7b

Deploy optimization

Pipeline #1736653243 Running for optimization

build_packages

Related jobs

✓ build_package1

→ ✓ build_package2

✓ build_package3

Vitaliy Falkevich / deploy-optimization / Jobs / #9532304636

Search visible log output

16 \$ git remote set-url origin "\${CI_REPOSITORY_URL}"

17 Restoring cache 00:01

18 Checking cache for 3aa73e7b-2-non_protected...

19 Downloading cache from https://storage.googleapis.com/gitlab-com-runners-cache/project/68361984/3aa73e7b-2-non_protected ETag="ef2f715802504aaed0e3ef09298f6f2c"

20 Successfully extracted cache

21 Executing "step_script" stage of the job script 03:01

22 Using docker image sha256:03501b0671a4e8450444678df0aad9c78eccc5fe2f4455daae571bd879f6d965 for dockerhub.b.timeweb.cloud/node:20.9.0 with digest dockerhub.timeweb.cloud/node@sha256:146bbe4eae99ae885be2a0a767f63a4b96032141d70d80e590f10e0d7ebabcb ...

23 \$ yarn workspace package3 run build

24 yarn workspace v1.22.19

25 yarn run v1.22.19

26 \$ echo 'start building package 3...' && sleep 180 && echo 'package 3 is built!'

27 start building package 3...

28 package 3 is built!

29 Done in 180.05s.

30 Done in 180.28s.

31 Saving cache for successful job 00:03

32 Creating cache 3aa73e7b-2-non_protected...

33 node_modules: found 640 matching artifact files and directories

34 packages/*/node_modules: found 15 matching artifact files and directories

35 WARNING: packages/*/dist: no matching files. Ensure that the artifact path is relative to the working directory (/builds/vitaliifalkevich/deploy-optimization)

36 apps/*: found 24 matching artifact files and directories

37 Uploading cache.zip to https://storage.googleapis.com/gitlab-com-runners-cache/project/68361984/3aa73e7b-2-non_protected

38 Created cache

39 Cleaning up project directory and file based variables 00:00

40 Job succeeded

Duration: 3 minutes 44 seconds

Finished: 1 minute ago

Queued: 0 seconds

Timeout: 1h (from project) ?

Runner: #12270840 (-AzERasQ) 5-blue.saas-linux-small-amd64.runners-manager.gitlab.com/default

Source: Push

Commit 3aa73e7b

Deploy optimization

Pipeline #1736653243 Running for optimization

build_packages

Related jobs

✓ build_package1

✓ build_package2

→ ✓ build_package3

[All](#) **1** [Finished](#) [Branches](#) [Tags](#)
[View analytics](#) [Clear runner caches](#) [New pipeline](#)

Status = Running × Q Show Pipeline ID ▼

Status	Pipeline	Created by	Stages	Actions
Running	Deploy optimization #1736653243 optimization 3aa73e7b latest branch		Stage: test_packages test_package2 ⌛ test_package1 ✓ test_package3 ✓	▼

Search visible log output Q 📄 ↶ ↷ ↕ ↗

```

16 $ git remote set-url origin "${CI_REPOSITORY_URL}"
17 Restoring cache
18 Checking cache for 3aa73e7b-2-non_protected...
19 Downloading cache from https://storage.googleapis.com/gitlab-com-runners-cache/project/68361904/3aa73e7b-2-non_protected ETag="83c436bc039c7085b42b490ec08cbb5"
20 Successfully extracted cache
21 Executing "step_script" stage of the job script
22 Using docker image sha256:03501b0671a4e845044678df0aad9c78ecce5fe2f4455daae571bd879f6d965 for dockerhub
   b.timeweb.cloud/node:20.9.0 with digest dockerhub.timeweb.cloud/node@sha256:146bbe4eae99ae885be2a0a767f63a4b96032141d70d80e590f10e0d7ebabcb ...
23 $ yarn workspace package1 run test
24 yarn workspace v1.22.19
25 yarn run v1.22.19
26 $ echo 'start testing package 1...' && sleep 60 && echo 'testing for package 1 completed successfully'
27 start testing package 1...
28 testing for package 1 completed successfully
29 Done in 60.05s.
30 Done in 60.28s.
31 Saving cache for successful job
32 Creating cache 3aa73e7b-2-non_protected...
33 node_modules: found 640 matching artifact files and directories
34 packages/*/node_modules: found 15 matching artifact files and directories
35 WARNING: packages/*/dist: no matching files. Ensure that the artifact path is relative to the working directory (/builds/vitaliifalkevich/deploy-optimization)
36 apps/*: found 24 matching artifact files and directories
37 Uploading cache.zip to https://storage.googleapis.com/gitlab-com-runners-cache/project/68361904/3aa73e7b-2-non_protected
38 Created cache
39 Cleaning up project directory and file based variables
40 Job succeeded

```

00:01

01:01

00:03

00:01

Duration: 1 minute 44 seconds

Finished: 26 seconds ago

Queued: 1 second

Timeout: 1h (from project) ?

Runner: #32976687 (nN8vMRS9) 6-blue.saas-linux-small-amd64.runners-manager.gitlab.com/default

Source: Push

Commit 3aa73e7b 🔗
 Deploy optimization

Pipeline #1736653243 Running for optimization 🔗
 test_packages ▼

Related jobs
🔄 test_package2
→ ✓ test_package1
✓ test_package3

Vitaliy Falkevich / deploy-optimization / Jobs / #9532304715

Search visible log output

00:01

16 \$ git remote set-url origin "\${CI_REPOSITORY_URL}"

17 Restoring cache

18 Checking cache for 3aa73e7b-2-non_protected...

19 Downloading cache from https://storage.googleapis.com/gitlab-com-runners-cache/project/68361984/3aa73e7b-2-non_protected ETag="83c436bc839c7085b42b490ec08cfbb5"

20 Successfully extracted cache

21 Executing "step_script" stage of the job script

22 Using docker image sha256:03501b0671a4e8450444678df0aad9c78ecce5fe2f4455daae571bd879f6d965 for dockerhub b.timeweb.cloud/node:20.9.0 with digest dockerhub.timeweb.cloud/node@sha256:146bbe4eae99ae885be2a0a767f63a4b96032141d70d80e590f10e0d7ebabcb ...

23 \$ yarn workspace package3 run test

24 yarn workspace v1.22.19

25 yarn run v1.22.19

26 \$ echo 'start testing package 3...' && sleep 60 && echo 'testing for package 3 completed successfully'

27 start testing package 3...

28 testing for package 3 completed successfully

29 Done in 60.05s.

30 Done in 60.28s.

31 Saving cache for successful job

32 Creating cache 3aa73e7b-2-non_protected...

33 node_modules: found 640 matching artifact files and directories

34 packages/*/node_modules: found 15 matching artifact files and directories

35 WARNING: packages/*/dist: no matching files. Ensure that the artifact path is relative to the working directory (/builds/vitaliifalkevich/deploy-optimization)

36 apps/*: found 24 matching artifact files and directories

37 Uploading cache.zip to https://storage.googleapis.com/gitlab-com-runners-cache/project/68361984/3aa73e7b-2-non_protected

38 Created cache

39 Cleaning up project directory and file based variables

40 Job succeeded

Duration: 1 minute 55 seconds

Finished: 38 seconds ago

Queued: 0 seconds

Timeout: 1h (from project)

Runner: #12270807 (j1aLDqXS) 1-blue.saas-linux-small-amd64.runners-manager.gitlab.com/default

Source: Push

Commit 3aa73e7b

Deploy optimization

Pipeline #1736653243 Running for optimization

test_packages

Related jobs

test_package2

test_package1

test_package3

Vitaliy Falkevich / deploy-optimization / Jobs / #9532304704

Search visible log output

00:01

12 Initialized empty Git repository in /builds/vitaliifalkevich/deploy-optimization/.git/

13 Created fresh repository.

14 Checking out 3aa73e7b as detached HEAD (ref is optimization)...

15 Skipping Git submodules setup

16 \$ git remote set-url origin "\${CI_REPOSITORY_URL}"

17 Restoring cache

18 Checking cache for 3aa73e7b-2-non_protected...

19 Downloading cache from https://storage.googleapis.com/gitlab-com-runners-cache/project/68361984/3aa73e7b-2-non_protected ETag="818b6c729fd3742b109e182706553962"

20 Successfully extracted cache

21 Executing "step_script" stage of the job script

22 Using docker image sha256:03501b0671a4e8450444678df0aad9c78ecce5fe2f4455daae571bd879f6d965 for dockerhub b.timeweb.cloud/node:20.9.0 with digest dockerhub.timeweb.cloud/node@sha256:146bbe4eae99ae885be2a0a767f63a4b96032141d70d80e590f10e0d7ebabcb ...

23 \$ yarn workspace package2 run test

24 yarn workspace v1.22.19

25 yarn run v1.22.19

26 \$ echo 'start testing package 2...' && sleep 60 && echo 'testing for package 2 completed successfully'

27 start testing package 2...

28 testing for package 2 completed successfully

29 Done in 60.05s.

30 Done in 60.29s.

31 Saving cache for successful job

32 Creating cache 3aa73e7b-2-non_protected...

33 node_modules: found 640 matching artifact files and directories

34 packages/*/node_modules: found 15 matching artifact files and directories

35 WARNING: packages/*/dist: no matching files. Ensure that the artifact path is relative to the working directory (/builds/vitaliifalkevich/deploy-optimization)

36 apps/*: found 24 matching artifact files and directories

37 Uploading cache.zip to https://storage.googleapis.com/gitlab-com-runners-cache/project/68361984/3aa73e7b-2-non_protected

38 Created cache

39 Cleaning up project directory and file based variables

40 Job succeeded

Duration: 4 minutes 6 seconds

Finished: just now

Queued: 0 seconds

Timeout: 1h (from project)

Runner: #12270835 (zxwgkJP) 3-blue.saas-linux-small-amd64.runners-manager.gitlab.com/default

Source: Push

Commit 3aa73e7b

Deploy optimization

Pipeline #1736653243 Running for optimization

test_packages

Related jobs

test_package1

test_package2

test_package3

View analytics Clear runner caches New pipeline

Status = Running X Q Show Pipeline ID v

Status	Pipeline	Created by	Stages	Actions
Running	Deploy optimization #1736653243 optimization 3aa73e7b latest branch		Stage: build_apps build_app1 build_app2	

Search visible log output

```
16 $ git remote set-url origin "${CI_REPOSITORY_URL}"
17 Restoring cache
18 Checking cache for 3aa73e7b-2-non_protected...
19 Downloading cache from https://storage.googleapis.com/gitlab-com-runners-cache/project/68361984/3aa73e7b-2-non_protected ETag="b6f8d8dc42a5535a625aba811ab8ea0"
20 Successfully extracted cache
21 Executing "step_script" stage of the job script
22 Using docker image sha256:03501b0671a4e8450444678df0aad9c78ecce5fe2f4455daae571bd879f6d965 for dockerhub
b.timeweb.cloud/node:20.9.0 with digest dockerhub.timeweb.cloud/node@sha256:146bbe4eae99ae885be2a8a767f63a4b96032141d78d80e598f10e0d7ebabcb ...
23 $ yarn workspace app1 run build
24 yarn workspace v1.22.19
25 yarn run v1.22.19
26 $ echo 'start building app1...' && sleep 240 && echo 'app1 is built!'
27 start building app1...
28 app1 is built!
29 Done in 240.05s.
30 Done in 240.28s.
31 Saving cache for successful job
32 Creating cache 3aa73e7b-2-non_protected...
33 node_modules: found 640 matching artifact files and directories
34 packages/*/node_modules: found 15 matching artifact files and directories
35 WARNING: packages/*/dist: no matching files. Ensure that the artifact path is relative to the working directory (/builds/vitaliifalkevich/deploy-optimization)
36 apps/*: found 24 matching artifact files and directories
37 Uploading cache.zip to https://storage.googleapis.com/gitlab-com-runners-cache/project/68361984/3aa73e7b-2-non_protected
38 Created cache
39 Cleaning up project directory and file based variables
40 Job succeeded
```

Duration: 4 minutes 51 seconds
Finished: 17 seconds ago
Queued: 0 seconds
Timeout: 1h (from project)
Runner: #12270840 (-AzERasQ) 5-blue.saas-linux-small-amd64.runners-manager.gitlab.com/default
Source: Push
Commit 3aa73e7b
Deploy optimization
Pipeline #1736653243 Running for optimization
build_apps
Related jobs
build_app2
build_app1

[All](#) 1 [Finished](#) [Branches](#) [Tags](#)
[View analytics](#) [Clear runner caches](#) [New pipeline](#)

Status = Running × Q Show Pipeline ID ▼

Status	Pipeline	Created by	Stages	Actions
Running	Deploy optimization #1736653243 optimization 3aa73e7b latest branch		Stage: build_apps Running build_app2 ⌵ Completed build_app1 ↺	⛔ 📄 ▼

Search visible log output Q 📄 ⇐ ⇨ ⬆ ⬇ ⬇ ⬆

```

16 $ git remote set-url origin "${CI_REPOSITORY_URL}"
17 Restoring cache 00:02
18 Checking cache for 3aa73e7b-2-non_protected...
19 Downloading cache from https://storage.googleapis.com/gitlab-com-runners-cache/project/68361984/3aa73e7b-2-non_protected ETag="b6f8d8dc42a5535a625aba811ab8ea0"
20 Successfully extracted cache
21 Executing "step_script" stage of the job script 00:01
22 Using docker image sha256:03501b0671a4e8450444678df0aad9c78ecce5fe2f4455daae571bd879f6d965 for dockerhub
23 $ yarn workspace app2 run build
24 yarn workspace v1.22.19
25 yarn run v1.22.19
26 $ echo 'start building app2...' && sleep 240 && echo 'app2 is built!'
27 start building app2...
28 app2 is built!
29 Done in 240.06s.
30 Done in 240.31s.
31 Saving cache for successful job 00:03
32 Creating cache 3aa73e7b-2-non_protected...
33 node_modules: found 640 matching artifact files and directories
34 packages/*/node_modules: found 15 matching artifact files and directories
35 WARNING: packages/*/dist: no matching files. Ensure that the artifact path is relative to the working directory (/builds/vitaliifalkevich/deploy-optimization)
36 apps/*: found 24 matching artifact files and directories
37 Uploading cache.zip to https://storage.googleapis.com/gitlab-com-runners-cache/project/68361984/3aa73e7b-2-non_protected
38 Created cache
39 Cleaning up project directory and file based variables 00:00
40 Job succeeded
  
```

Duration: 7 minutes 8 seconds
 Finished: just now
 Queued: 0 seconds
 Timeout: 1h (from project) ?
 Runner: #12270837 (J2nyww-s) 4-blue.saas-linux-small-amd64.runners-manager.gitlab.com/default
 Source: Push

Commit 3aa73e7b 🔗
 Deploy optimization

Pipeline #1736653243 Running for optimization 🔗
 build_apps ▼

Related jobs

- ✓ build_app1
- ✓ build_app2

All 1 Finished Branches Tags

View analytics

Clear runner caches

New pipeline

Status = Running X

X Q

Show Pipeline ID v

Status	Pipeline	Created by	Stages	Actions
Running	Deploy optimization #1736653243 optimization ~ 3aa73e7b latest branch		    	 

Stage: assemble

assemble

Vitaliy Falkevich / deploy-optimization / Jobs / #9532304808

Search visible log output

Q



```

10 Getting source from Git repository
11 Fetching changes with git depth set to 20...
12 Initialized empty Git repository in /builds/vitaliifalkevich/deploy-optimization/.git/
13 Created fresh repository.
14 Checking out 3aa73e7b as detached HEAD (ref is optimization)...
15 Skipping Git submodules setup
16 $ git remote set-url origin "${CI_REPOSITORY_URL}"
17 Restoring cache
18 Checking cache for 3aa73e7b-2-non_protected...
19 Downloading cache from https://storage.googleapis.com/gitlab-com-runners-cache/project/68361904/3aa73e7b-2-non_protected ETag="b9ebfd8dd31d39dc5948bf0e5b1607cb"
20 Successfully extracted cache
21 Executing "step_script" stage of the job script
22 Using docker image sha256:62d2ffd1ba2a4f9df123711e18c1cbb361bdf212ce3bee3d6ebfd769961be79c for ruby:3.1 with digest ruby@sha256:aaf664741b8de45efe7c5db430430784c11af7d069027b2f080b74a53b8882ae ...
23 $ sleep 120 && echo "Image is uploaded!"
24 Image is uploaded!
25 Saving cache for successful job
26 Creating cache 3aa73e7b-2-non_protected...
27 node_modules: found 640 matching artifact files and directories
28 packages/*/node_modules: found 15 matching artifact files and directories
29 WARNING: packages/*/dist: no matching files. Ensure that the artifact path is relative to the working directory (/builds/vitaliifalkevich/deploy-optimization)
30 apps/*: found 24 matching artifact files and directories
31 Uploading cache-cache.zip to https://storage.googleapis.com/gitlab-com-runners-cache/project/68361904/3aa73e7b-2-non_protected
32 Created cache
33 Cleaning up project directory and file based variables
34 Job succeeded

```

Duration: 2 minutes 33 seconds

Finished: 14 seconds ago

Queued: 0 seconds

Timeout: 1h (from project) ?

Runner: #12270837 (J2nyww-s) 4-blue.saas-linux-small-amd64.runners-manager.gitlab.com/default

Source: Push

Commit 3aa73e7b 

Deploy optimization

 Pipeline #1736653243  Passed for optimization 

assemble

Related jobs

→  assemble

ДОДАТОК Є

Порівняння результатів експерименту до оптимізації збірки та після

Vitaliy Falkevich / deploy-optimization / Pipelines

All2FinishedBranchesTags

View analyticsClear runner cachesNew pipeline

Status = Passed X

Show Pipeline ID

Status	Pipeline	Created by	Stages	Actions
Passed 00:19:16 36 seconds ago	Deploy optimization #1736653243 optimization 3aa73e7b latestbranch		✓✓✓✓✓	Download
Passed 00:27:43 20 hours ago	default CI #1734956634 main e4e4fa56 latestbranch		✓✓✓✓✓	Download