КІБЕРБЕЗПЕКА: освіта, наука, техніка
CYBERSECURITY: EDUCATION, SCIENCE, TECHNIQUE
№ 3 (31), 2025
ISSN 2663 - 4023
DOI 10.28925/2663-4023.2025.31.1057
УДК 004.41:004.75

Трофименко Олена Григорівна
кандидат технічних наук, доцент,
доцент кафедри інформаційних технологій,
Національний університет «Одеська юридична академія», м. Одеса, Україна,
ORCID: 0000-0001-7626-0886
trofymenko@onua.edu.ua

Манаков Сергій Юрійович
кандидат технічних наук, доцент,
доцент кафедри інформаційних технологій,
Національний університет «Одеська юридична академія», м. Одеса, Україна,
ORCID: 0000-0001-5930-4592
manakov_serhii@onua.edu.ua

Чикунов Павло Олександрович
кандидат технічних наук, доцент,
доцент кафедри інформаційних технологій,
Національний університет "Одеська юридична академія", м. Одеса, Україна,
ORCID: 0000-0003-4959-774412:27
pavel@onua.edu.ua

Лобода Юлія Геннадіївна
кандидат технічних наук, доцент,
доцент кафедри інформаційних технологій,
Національний університет «Одеська юридична академія», м. Одеса, Україна,
ORCID: 0000-0001-7083-552X
jul.loboda@gmail.com

Гурін Євген Павлович
здобувач вищої освіти,
Національний університет "Одеська юридична академія", м. Одеса, Україна,
ORCID: 0009-0000-4640-6581
egurin443@gmail.com

МАРШРУТИЗАЦІЯ В МІКРОФРОНТЕНДНИХ ЗАСТОСУНКАХ: АРХІТЕКТУРНІ ПІДХОДИ, ВИКЛИКИ ТА ШЛЯХИ ЇХ ВИРІШЕННЯ

Анотація. Стаття присвячена організації маршрутизації у мікрофронтендній архітектурі, яка є одним із ключових напрямів сучасної веброзробки. Актуальність теми зумовлена високою динамікою розвитку технологій та постійним зростанням вимог до масштабованості, модульності й гнучкості клієнтських систем. Мікрофронтендна архітектура розглядається як фронтенд-еквівалент мікросервісної парадигми, який забезпечує автономність розробки окремих модулів, їх незалежне тестування та оновлення, але водночас породжує низку специфічних викликів, серед яких маршрутизація займає центральне місце. Вона визначає узгодженість взаємодії між підсистемами, впливає на продуктивність і якість користувацького досвіду та формує основу для подальшої масштабованості системи. У статті здійснено аналіз підходів до організації маршрутизації у мікрофронтендній архітектурі, зокрема глобального, локального та гібридного. Показано, що глобальний підхід забезпечує централізований контроль і узгодженість навігації, але знижує автономність модулів і створює ризик централізованих відмов. Локальний підхід, навпаки, гарантує незалежність команд і швидкість розробки, проте призводить до конфліктів маршрутів, проблем із синхронізацією історії переходів та зниження узгодженості користувацького досвіду. Гібридні моделі намагаються поєднати переваги обох підходів, проте залишають відкритими питання синхронізації між різними фреймворками та підтримки SSR/SSG. Окрему увагу

приділено аналізу ключових викликів, що виникають у процесі маршрутизації: конфліктів URL, координації історії навігації, проблем SEO та індексації, труднощів при lazy loading модулів, залежності від конкретних фреймворків, узгодженості UI/UX, надмірних витрат продуктивності, обмеженості ресурсів команд та невідповідності масштабу застосування мікрофронтендів у малих проєктах. Додатково розглянуто проблему синхронізації стану між модулями, яка є критичною для забезпечення консистентності бізнес-логіки та даних. У роботі запропоновано низку практичних рішень, спрямованих на подолання зазначених викликів.

Ключові слова: мікрофронтендна архітектура; маршрутизація; виклики маршрутизації; продуктивність; масштабованість; фреймворк; веброзробка; UI/UX.

ВСТУП

Постановка проблеми. Сфера веброзробки характеризується надзвичайно високою динамікою розвитку: щоденно з’являються нові бібліотеки, фреймворки та інструменти, спрямовані на оптимізацію процесів створення клієнт-серверних систем. Така інтенсивна еволюція технологій зумовлює постійне зростання вимог до сучасних вебзастосунків, які мають відповідати критеріям масштабованості, модульності, гнучкості та зручності підтримки. У цьому контексті вибір архітектурного підходу стає одним із ключових рішень, що визначає не лише технічну якість продукту, а й ефективність командної роботи та довгострокову життєздатність проєкту.

Традиційні архітектури, такі як моноліти чи монорепозиторії, історично забезпечували базові потреби розробників, проте зі зростанням складності та обсягів програмних систем вони все частіше демонструють обмеження. Зокрема, у великих проєктах ускладнюється координація між розробниками, знижується продуктивність розробки, а внесення змін до окремих компонентів може призводити до значних витрат часу та ресурсів. Ці фактори актуалізували пошук нових архітектурних рішень, здатних забезпечити незалежність модулів, швидке оновлення та масштабування без втрати цілісності системи.

У відповідь на ці виклики сформувався підхід мікрофронтендної архітектури, який можна розглядати як фронтенд-еквівалент мікросервісної парадигми, що вже довела свою ефективність у бекенд-розробці. Мікрофронтенди (Microfrontend, MFE) дозволяють розділяти застосунок на автономні частини, кожна з яких може розроблятися, тестуватися та оновлюватися незалежно, що значно підвищує гнучкість та стійкість системи. Водночас саме питання маршрутизації у таких застосунках постає як одне з найскладніших і найважливіших, адже воно визначає узгодженість взаємодії між окремими модулями та впливає на користувацький досвід. Саме тому дослідження викликів і перспектив розвитку механізмів маршрутизації в мікрофронтендних застосунках є актуальним.

Аналіз останніх досліджень та публікацій. Попри те, що створення та налаштування архітектури мікросервісів залишається складним і тривалим завданням для розробників, різні підходи до мікрофронтенд та мікросервісного архітектурного стиля у бекенді все частіше привертають увагу дослідників та науковців. Так, у статті [1] розглянуто маршрутизацію запитів у мікросервісній архітектурі та запропоновано архітектурну модель на API Spring Framework. У роботі [2] досліджено застосування принципів MFE у розробці вебзастосунків на базі Flutter. Дослідження [3] спрямоване на огляд публікацій з інтеграції мікросервісів і мікрофронтендів та їх порівняння. У статті [4] вказано на важливість трансформації застарілих систем з монолітної архітектури на мікросервісну задля покращення продуктивності, портативності та інших важливих

характеристик, наголошено на тому, що такий процес є складним і дорогим, а також запропоновано триетапний метод трансформації архітектури монолітних односторінкових застосунків (Single Page Applications, SPA) на мікросервісній архітектурі (Microservice Architecture, MSA). Автори роботи [5] розробили інструмент для створення архітектур мікросервісів, який має три компоненти: графічний редактор для архітектурного подання та внутрішнього опису кожного мікросервісу, проміжний рушій трансформації для перетворення графічних елементів у модель коду та модуль для уточнення коду відповідно до архітектури мікросервісу. Дослідження [6] присвячено системному огляду 162 публікацій з архітектури мікросервісів, розподіливши їх на п’ять груп: декомпозиція, розміщення, планування ресурсів, міграція та виявлення аномалій. У статті [7] проаналізовано та розроблено методи переходу від монолітної архітектури до мікросервісної архітектури в традиційних системах планування ресурсів підприємства (Enterprise Resource Planning, ERP) для забезпечення їхньої масштабованості, гнучкості та безпеки. У дослідженні [8] розглядаються деталі архітектурних проблем міжмікросервісних комунікацій. У статті [9] проаналізовано відомі недоліки та вразливості, з якими може стикатися архітектура мікросервісів. У роботі [10] досліджено основні архітектурні принципи і шаблони мікросервісів, які використовуються для кіберфізичних систем. Загалом аналіз наявних публікацій свідчить про значну зацікавленість дослідників у перевагах MSA щодо масштабованості, автономного розгортання та командної незалежності. Але, попри активну індустріальну апробацію мікрофронтендних підходів, характер академічних та прикладних досліджень маршрутизації в архітектурах мікрофронтендів залишається дещо фрагментарним.

Метою статті є комплексний аналіз підходів до маршрутизації в мікрофронтендній архітектурі, дослідження ключових технічних викликів та формування практичних рекомендацій для їх вирішення, що враховують сучасні тенденції та потреби веброзробки.

РЕЗУЛЬТАТИ ДОСЛІДЖЕННЯ

Мікрофронтендна архітектура є фронтенд-аналогом мікросервісної парадигми, проте застосовується на рівні клієнтського інтерфейсу. Вона ґрунтується на принципі декомпозиції системи на незалежні підмодулі, кожен з яких може мати власну програмну логіку, користувацький інтерфейс (UI) та стан, а маршрутизація реалізується залежно від вибраного підходу – глобального, локального чи гібридного. Така організація забезпечує автономність розробки, тестування та оновлення окремих модулів без критичного впливу на інші частини системи. У практиці MFE часто застосовується концепція «lazy loading», яка дозволяє завантажувати лише ті модулі, що потрібні користувачеві в конкретний момент часу, тим самим зменшуючи час початкового завантаження застосунку та оптимізуючи використання ресурсів [11]. Водночас впровадження мікрофронтендів породжує низку специфічних викликів, серед яких ключовими є ефективна маршрутизація між модулями, уникнення конфліктів URL, синхронізація історії переходів, забезпечення узгодженості користувацького досвіду та підтримка SEO-оптимізації.

Розбиття застосунку на підмодулі вимагає від системи ефективної взаємодії між відповідними компонентами. У мікрофронтенд-архітектурі маршрутизація є одним з головних факторів, який визначає загальну якість проєкту. Так, ефективна маршрутизація визначає узгодженість частин, масштабованість та зручність навігації

між модулями у проєкту. У сучасній веброзробці вирізняють декілька основних підходів до реалізації архітектури маршрутизації: глобальний (він же централізований, host-managed), локальний (self-contained) та гібридний (рис. 1). Підходи відрізняються рівнем централізації маршрутизатора, механізмом керування та способом інтеграції між компонентами.

Глобальний підхід до маршрутизації в мікрофронтендній архітектурі характеризується централізацією механізмів навігації у кореневому застосунку, який виконує функції координатора для всіх підмодулів. У такій моделі саме host-компонент містить центральний роутер, що відповідає за аналіз URL-адрес, визначення активного мікрофронтенду та його відображення у клієнтському інтерфейсі (рис. 2). Це забезпечує єдиний контроль над конфігурацією маршрутів, що створює передумови для узгодженості користувацького досвіду, спрощення аналітики та підвищення ефективності SEO-оптимізації, оскільки всі переходи обробляються централізовано. Централізована маршрутизація дозволяє уникати конфліктів між модулями та забезпечує єдину політику роботи з History API, проте водночас вона суттєво знижує автономність окремих мікрофронтендів і ускладнює масштабування системи. Висока залежність від логіки центрального ядра створює ризик так званих «централізованих відмов», коли збій у host-роутері може паралізувати роботу всієї платформи.

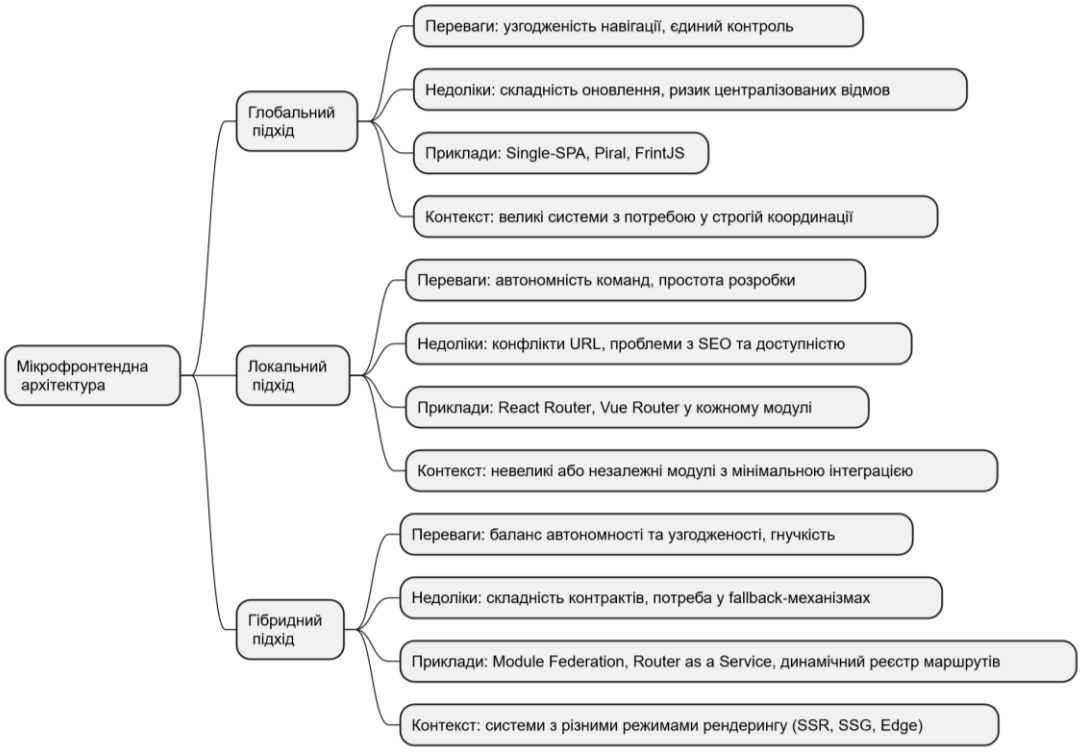

Рис. 1. Маршрутизація в мікрофронтендній архітектурі

Приклади реалізації цього підходу можна знайти у фреймворку Single-SPA, який забезпечує централізоване керування життєвим циклом модулів, а також у Webpack Module Federation, що дозволяє динамічно імпортувати мікрофронтенди з різних джерел, залишаючи контроль маршрутів у компетенції кореневого застосунку [12]. Попри наявність таких технологій, глобальна маршрутизація залишається складною у підтримці при розширенні великих систем, особливо у випадках динамічного імпорту модулів, що потребує додаткових механізмів узгодження та моніторингу.

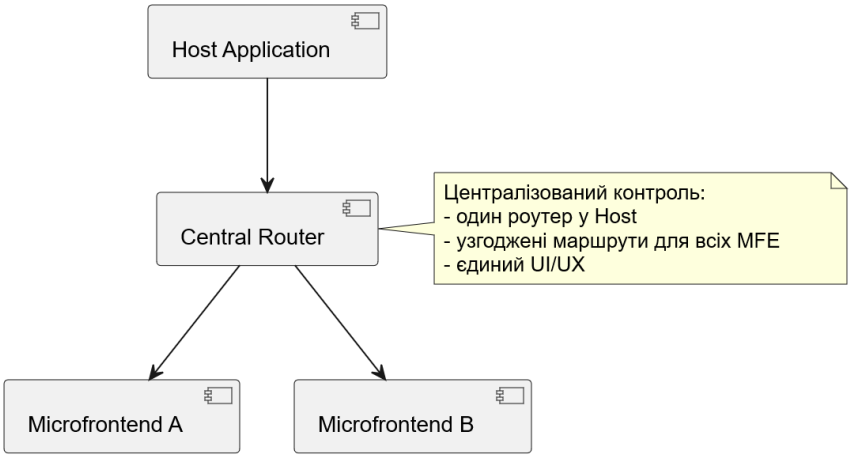

Рис. 2. Глобальний (централізований) підхід до маршрутизації

Локальний підхід до маршрутизації в мікрофронтендній архітектурі, який часто позначають як self-contained, полягає у реалізації логіки навігації безпосередньо всередині кожного окремого MFE незалежно від кореневого застосунку. У такій моделі кожен модуль є самодостатнім і володіє власним роутером, що відповідає за обробку переходів, визначення активних компонентів та їх відображення у клієнтському інтерфейсі (рис. 3). Це забезпечує високу автономність і гнучкість, що особливо важливо у випадках, коли над різними частинами системи працюють незалежні команди, оскільки дозволяє розробляти та розгортати MFE ізольовано. Водночас локальна маршрутизація створює низку проблем, пов'язаних із інтеграцією у спільному середовищі. Відсутність глобального контролю ускладнює формування єдиної історії переходів та централізованої аналітики, що потребує додаткової синхронізації з браузерним API, зокрема window.history [13]. Попри те що сучасні фреймворки, такі як React Router чи Vue Router, надають абстракції над History API і спрощують роботу з навігацією, використання кількох різних роутерів у межах одного застосунку часто призводить до конфліктів маршрутів, непередбачуваної поведінки та зниження узгодженості користувацького досвіду. Отже, локальний підхід забезпечує значну гнучкість і швидкість розробки, але водночас потребує додаткових механізмів координації для уникнення проблем при інтеграції кількох автономних MFE в єдину систему.

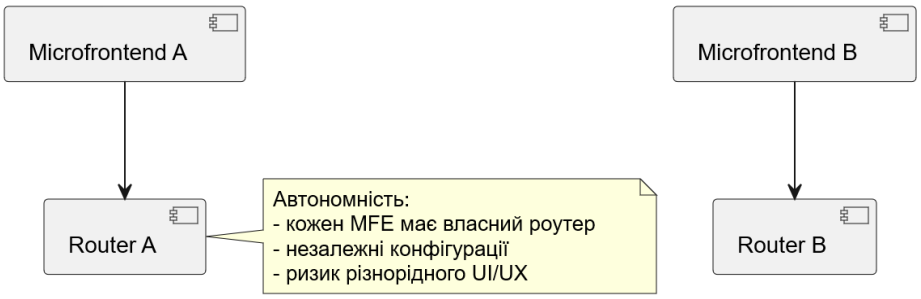

Рис. 3. Локальний (self-contained) підхід до маршрутизації

Гібридний підхід до маршрутизації в мікрофронтендній архітектурі поєднує властивості глобальної та локальної моделей і прагне забезпечити баланс між централізованим управлінням та автономністю окремих мікрофронтендів (рис. 4). У такій конфігурації host-застосунок виконує роль координатора, формуючи реєстр

маршрутів, що містить метадані про навігаційні правила кожного MFE. Цей реєстр може оновлюватися при додаванні чи видаленні модулів, що підвищує гнучкість системи та зменшує ризик конфліктів між різними частинами застосунку. Водночас кожен MFE зберігає власний локальний роутер, який відповідає за внутрішню навігацію, що дозволяє поєднувати централізовану координацію з модульною самостійністю [14]. Запровадження гібридної моделі значно полегшує управління складними маршрутами у великих застосунках, проте створює низку викликів. Однією з ключових проблем є синхронізація локальних маршрутизаторів, особливо у випадках використання різних фреймворків у межах одного проєкту, наприклад React у host-застосунку та Vue у окремому MFE. Додатковим завданням стає реалізація fallback-маршрутів, які забезпечують стійкість системи у разі відмови окремих модулів, а також підтримка різних режимів рендерингу, включно з SSR, SSG та Edge Rendering. Ці аспекти мають критичне значення для продуктивності та SEO-оптимізації, оскільки визначають узгодженість роботи всієї системи та її здатність адаптуватися до змінних умов виконання.

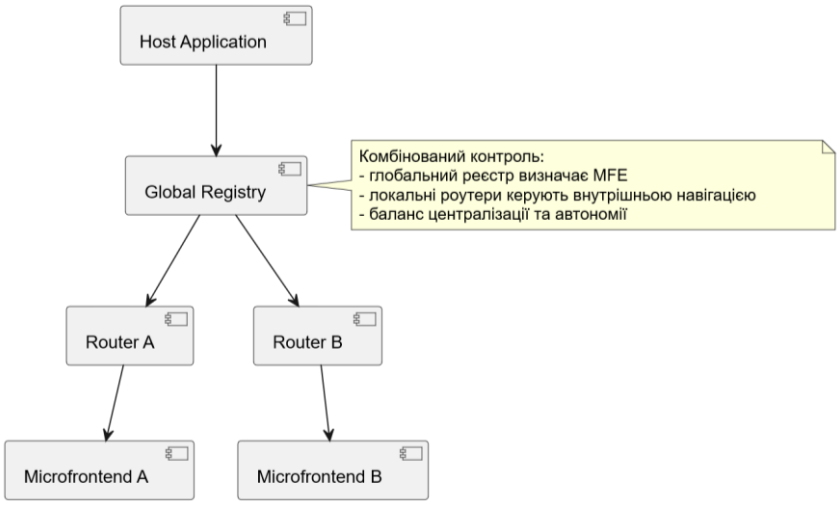

Рис. 4. Гібридний підхід до маршрутизації

У межах гібридних моделей маршрутизації перспективним інструментом виступає GraphQL, який може виконувати функцію динамічного управління маршрутами у складних клієнтських застосунках. Його використання дозволяє формувати запити до навігаційної логіки у спосіб, що забезпечує вибіркове отримання лише тих маршрутів, які є актуальними для користувача в конкретний момент часу. Такий підхід знижує навантаження на клієнтську частину та водночас створює передумови для централізованого контролю над навігаційними даними. На відміну від традиційних REST-запитів, які часто потребують обробки надлишкової інформації, GraphQL є більш адаптивним до складних ієрархій маршрутів та здатний забезпечити узгодженість роботи системи в умовах постійного масштабування [15]. Саме тому його застосування у сфері маршрутизації розглядається як перспективний напрям, що поєднує управління даними з оптимізацією навігаційних процесів у мікрофронтендній архітектурі. Ефективною також є *автоматизована реєстрація маршрутів* на основі метаданих, що дозволяє централізовано відстежувати конфігурації та уникати дублювання.

Розробка клієнтських застосунків із використанням мікрофронтендної архітектури супроводжується появою комплексних викликів, які охоплюють організацію

маршрутизації, координацію автономних модулів, формування єдиного інтеграційного середовища та узгодження різнорідних технологій. Сукупність цих проблем ілюструє рис. 5, де систематизовано ключові виклики маршрутизації у мікрофронтендній архітектурі, що охоплюють технічні, продуктивні та організаційні аспекти інтеграції автономних модулів у єдине середовище.

– ***Конфлікти маршрутів між модулями*** є однією з найпоширеніших проблем у мікрофронтендній архітектурі [16]. Оскільки окремі MFE часто розробляються незалежними командами, виникають ситуації, коли різні частини застосунку використовують однакові шляхи або намагаються звертатися до тих самих маршрутів для різних цілей. За відсутності узгодженої політики іменування чи централізованого реєстру маршрутів це призводить до непередбачуваної поведінки, порушення навігаційної логіки та навіть до повного виходу з ладу окремих MFE. Проблема ускладнюється у випадках, коли модулі створюються на різних фреймворках, кожен з яких має власну маршрутизаційну модель, що підвищує ризик конфліктів. *Причиною* таких конфліктів є відсутність єдиної стратегії управління простором URI-адрес та недостатня координація між командами. *Наслідки* проявляються у вигляді дублювання маршрутів, некоректного відображення інтерфейсу та зниження узгодженості користувацького досвіду. Для *усунення* цієї проблеми доцільним є впровадження *стандартизованих схем іменування маршрутів* із використанням префіксування. Принцип префіксування полягає у поділі простору URL/URI-адрес на ієрархічні сегменти, які відповідають окремим MFE: наприклад, усі маршрути модуля керування користувачами можуть починатися з /users, а модулів звітності – з /reports. Такий підхід забезпечує чітку ізоляцію маршрутів і зменшує ймовірність їх перетинання.

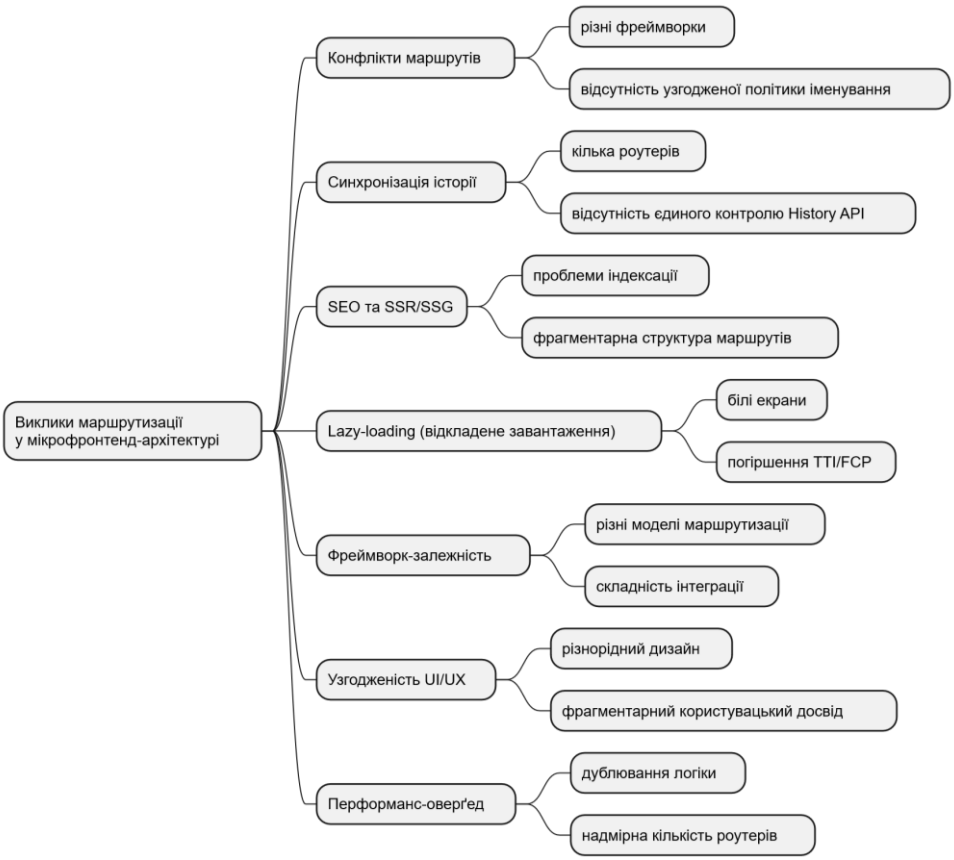

Рис. 5. Ключові виклики маршрутизації у мікрофронтендній архітектурі

Важливим інструментом є також *runtime-валідатори маршрутів*, які перевіряють унікальність та коректність конфігурації ще на етапі завантаження MFE. У сучасних практиках рекомендується інтегрувати такі валідатори у CI/CD-конвеєри, що забезпечує раннє виявлення конфліктів і знижує ризик їхнього потрапляння у production [17]. Поєднання префіксування, централізованої реєстрації маршрутів та runtime-валідації створює комплексний механізм запобігання конфліктам, що підвищує стабільність і передбачуваність роботи мікрофронтенд-системи.

– ***Координація історії навігації між підсистемами*** є суттєвим викликом у мікрофронтендній архітектурі. У класичних SPA-застосунках, що використовують глобальний маршрутизатор, історія переходів централізовано керується єдиним екземпляром роутера, що забезпечує узгодженість навігаційної логіки. Натомість у мікрофронтенд-системах кожен модуль може мати власний роутер, що створює потребу у синхронізованому доступі до механізмів навігаційної історії, зокрема до window.history та History API [18]. Відсутність такої синхронізації призводить до аномальної поведінки модулів, розриву єдиної навігаційної сесії та зниження узгодженості користувацького досвіду. *Причиною проблеми* є відсутність єдиного координатора історії переходів, що ускладнює інтеграцію незалежних модулів у спільне середовище (табл. 1). *Наслідки* проявляються у вигляді некоректної роботи кнопок «назад» та «вперед», втрати стану при переходах між модулями та неможливості централізованої аналітики користувацьких дій. Для *усунення* цієї проблеми застосовуються кілька підходів. Найбільш поширеним є використання *спільного об'єкта-обгортки (History singleton)*, який забезпечує єдиний доступ до глобального History API для всіх модулів. Такий механізм дозволяє централізовано зберігати навігаційну сесію та синхронізувати переходи незалежно від походження запиту. Ефективним рішенням також є застосування спеціалізованих бібліотек, зокрема *Single-SPA*, які надають абстракції для розділення доступу до History API між MFE та підтримують узгоджену навігаційну модель. Додатковим інженерним підходом є впровадження так званого *«брокера навігації»*, який виступає посередником між окремими MFE та root-застосунком. Такий брокер інкапсулює логіку переходів, забезпечує уніфіковане навігаційне API та послаблює зв'язки між модулями, що підвищує стійкість системи до змін та спрощує інтеграцію модулів, створених на різних фреймворках.

Таблиця 1

Аналіз проблем маршрутизації в мікрофронтендах і можливі шляхи до їх усунення

Проблема	Причина	Наслідки	Можливі рішення
Конфлікти маршрутів	Відсутність узгодженої політики іменування, різні фреймворки	Непередбачувана поведінка, вихід з ладу модулів	Префіксування, централізований реєстр маршрутів, runtime валідатори
Синхронізація історії	Кілька роутерів у різних модулях; відсутність єдиного контролю над History API	Аномальна поведінка, розрив логіки переходів	History singleton, бібліотеки (Single-SPA), брокер навігації
SEO та SSR/SSG	Клієнтський рендеринг, фрагментарна структура маршрутів	Недоступність контенту для пошукових роботів, погане ранжування	Ізоморфний SSR, рендеринг-шар, статична генерація сторінок
Lazy loading	Відкладене завантаження великих модулів	«Білі екрани», погіршення TTI та FCP	Skeleton компоненти, preload/prefetch, адаптивна стратегія завантаження
Фреймворк-залежність	Різні моделі маршрутизації (React, Angular, Vue, Svelte)	Неможливість єдиного навігаційного шару, складність інтеграції	Абстрактний API маршрутизації, адаптерний патерн, інтерфейсні мости

Узгодженість UI/UX	Автономна розробка модулів різними командами	Різнорідний дизайн, фрагментарний користувацький досвід	Уніфіковані рекомендації з дизайну, спільні UI-бібліотеки, централізовані style-tokens
Надмірні витрати продуктивності (performance overhead)	Надмірна кількість роутерів, дублювання логіки, зайві переходи	Зниження продуктивності, збільшення часу відгуку	Оптимізація маршрутизаторів, кешування, мінімізація дублювання логіки
Обмеженість ресурсів команд	Невеликі команди, брак досвіду	Повільна інтеграція, помилки у конфігурації	Використання готових бібліотек, шаблонів, централізованих інструментів
Невідповідність масштабу	Використання мікрофронтендів у малих проєктах	Надмірна складність, необґрунтовані витрати	Вибір архітектури відповідно до розміру проєкту; застосування спрощених SPA-підходів
Синхронізація стану між MFE	Автономне управління станом у кожному модулі (Redux, Vuex, Zustand тощо)	Неконсистентність даних, дублювання бізнес-логіки, розриви у взаємодії між модулями	Використання централізованого state-management API; впровадження спільного «state-gateway»; застосування подієвої шини (event bus) або брокера стану для узгодження даних між MFE

Загалом координація історії навігації у мікрофронтенд-архітектурі потребує впровадження централізованих механізмів доступу до History API, що дозволяє уникнути аномальної поведінки модулів, зберегти єдину навігаційну сесію та забезпечити узгодженість користувацького досвіду.

– ***SEO та SSR/SSG*** є одним із найбільш проблемних аспектів мікрофронтендної архітектури. При клієнтському рендерингу, характерному для більшості сучасних застосунків, контент окремих мікрофронтендів стає недоступним для пошукових роботів. Якщо маршрути MFE генеруються виключно на клієнтській стороні, індексація таких сторінок у пошукових системах значно ускладнюється [4]. Особливо гостро проблема постає у випадках фрагментарної або динамічної структури маршрутів, де пошукові алгоритми не можуть коректно відтворити повний контент. *Причиною* є відсутність серверного рендерингу, який забезпечує попереднє формування HTML-контенту для пошукових систем. *Наслідки* проявляються у зниженні доступності сторінок, погіршенні їхнього ранжування та втраті органічного трафіка. При застосуванні SSR (Server-Side Rendering) складність зростає через потребу правильної ініціалізації стану, завантаження маршруту та відображення UI кожного MFE на сервері [19]. Якщо кожен модуль має власну систему SSR, виникає проблема координації, що ускладнює рендеринг цільових маршрутів і негативно позначається на продуктивності та SEO. Для *усунення* цих проблем ефективним рішенням є впровадження *ізоморфного SSR-підходу*, який забезпечує єдину модель рендерингу як на клієнтській, так і на серверній стороні. Такий підхід дозволяє здійснювати попередній рендеринг контенту кожного MFE ще на сервері, що робить його доступним для пошукових роботів і підвищує якість індексації [20]. Практичним інструментом реалізації є створення спеціального *«рендеринг-шару»*, який виступає проміжною абстракцією між логікою та UI, асинхронно ініціалізує та рендерить MFE на сервері перед передачею результату клієнту. У випадках, коли SSR є недоцільним або технічно неможливим, альтернативним шляхом є *Static Site Generation (SSG).* Цей підхід стосується генерації статичних сторінок для ключових маршрутів, які підлягають індексації, з подальшим редиректом на клієнтський рендеринг. SSG забезпечує базову SEO-оптимізацію та дозволяє зберегти доступність критично важливих сторінок навіть у складних мікрофронтенд-системах. Тим самим забезпечення SEO у мікрофронтендній архітектурі потребує впровадження

SSR або SSG-рішень, які дозволяють узгодити інтереси пошукових систем із вимогами до продуктивності та модульної автономності.

– ***Проблема «лінивого завантаження» (lazy loading) модулів*** є одним із поширених технічних бар'єрів у мікрофронтендній архітектурі. При переході до маршруту, який відповідає модулю, що ще не завантажений у проєкт, може виникати затримка відображення контенту або навіть його повна відсутність. У таких випадках користувач стикається з «порожньою сторінкою», що негативно впливає на ключові показники продуктивності застосунку, зокрема Time to Interactive (TTI) [21] та First Contentful Paint (FCP) [22]. *Причиною* проблеми є відкладене завантаження значних обсягів коду при першій ініціалізації модуля, що створює додаткове навантаження на клієнтську частину та затримує відображення інтерфейсу. *Наслідки* проявляються у зниженні швидкості реакції системи, погіршенні користувацького досвіду та потенційному відтоку користувачів через низьку продуктивність. Для *усунення* цих проблем застосовуються кілька інженерних стратегій. Найбільш поширеним рішенням є використання *«скелетон-компонентів»*, тобто заздалегідь визначених placeholder-інтерфейсів, які відображаються під час очікування завантаження модуля. Це дозволяє уникнути «білих екранів» і зберегти відчуття безперервності взаємодії. Додатково ефективним є впровадження механізмів *пріоритетного попереднього завантаження (preload/prefetch)* для модулів, до яких очікується перехід користувача. Такий підхід зменшує затримку при першому зверненні до маршруту та покращує показники TTI і FCP. У системах з обмеженими ресурсами доцільним є застосування *адаптивної стратегії lazy loading*, яка враховує поведінкові шаблони користувачів і пріоритизує завантаження найбільш ймовірних маршрутів. Це дозволяє оптимізувати використання ресурсів та забезпечити баланс між продуктивністю і економією трафіку. Загалом проблема «лінивого завантаження» у мікрофронтенд-архітектурі може бути ефективно розв'язана шляхом поєднання скелетон-компонентів, механізмів пріоритетного preload/prefetch та адаптивних стратегій завантаження, що забезпечуватиме стабільність роботи системи та покращить користувацький досвід.

– ***Фреймворк-залежність*** є окремим викликом у мікрофронтендній архітектурі, який полягає у залежності від вибраного фреймворка та його маршрутизатора. Сучасні фронтенд-фреймворки – Angular, React, Vue, Svelte – демонструють суттєві відмінності у моделюванні маршрутизації. Наприклад, React Router реалізує декларативну модель навігації, Angular Router ґрунтується на імперативних конфігураціях із розширеними механізмами guard-ів, resolve-рів та preload-ерів, тоді як Vue Router орієнтується на вкладені маршрути та забезпечує гнучку інтеграцію з компонуванням сторінок. У межах мікрофронтенд-застосунку, який об'єднує модулі з різних фреймворків, виникає проблема несумісності навігаційних моделей, що ускладнює реалізацію єдиного навігаційного шару без додаткових адаптерів чи інтерфейсних мостів [23]. *Причиною* цієї проблеми є відсутність стандартизованого підходу до маршрутизації між фреймворками, що призводить до фрагментації навігаційної логіки. *Наслідки* проявляються у зростанні складності інтеграції, підвищенні ризику конфліктів між модулями та зниженні узгодженості користувацького досвіду. Для *усунення* фреймворк-залежності застосовується побудова *абстрактного шару маршрутизації*, який реалізується у вигляді спільного API. Усі мікрофронтенди, незалежно від фреймворка, взаємодіють із маршрутизатором через цей API, що стандартизує виклики навігаційних функцій та забезпечує єдину модель роботи з маршрутами. Додатковим рішенням є використання *адаптерного патерна*, при якому host-застосунок інкапсулює специфіку роботи з конкретним фреймворком і надає уніфікований інтерфейс для інтеграції. Це дозволяє

підтримувати технологічну різноманітність системи без втрати контролю над маршрутизаційним ядром та забезпечує масштабованість архітектури. Тим самим подолання фреймворк-залежності у мікрофронтенд-архітектурі потребує впровадження абстрактних шарів й адаптерних механізмів, які дозволяють стандартизувати навігаційну логіку та зберегти узгодженість роботи системи при використанні різних технологій.

– ***Узгодженість UI/UX*** є критичною проблемою у мікрофронтендній архітектурі, яка виникає внаслідок автономної розробки модулів різними командами. Кожен мікрофронтенд може мати власні підходи до дизайну та стилізації, що призводить до різнорідності інтерфейсних рішень. *Причиною* цього є відсутність єдиних стандартів оформлення та централізованих механізмів контролю стилів, які б забезпечували узгодженість між незалежними модулями. *Наслідки* проявляються у фрагментарному користувацькому досвіді: інтерфейс системи виглядає неоднорідним, окремі модулі відрізняються за стилем, поведінкою та інтерактивними елементами. Це знижує довіру користувачів, ускладнює навігацію та негативно впливає на сприйняття цілісності застосунку. У великих системах така різнорідність може призвести до зростання витрат на підтримку та ускладнити масштабування, оскільки кожна команда змушена вирішувати проблеми узгодженості самостійно. Для *усунення* цієї проблеми доцільним є впровадження *уніфікованих рекомендацій з дизайну*, які визначають базові принципи стилізації, компонування та інтерактивності. Ефективним рішенням є *використання спільних UI-бібліотек*, які забезпечують повторне використання компонентів та стандартизують їхню поведінку [24]. Додатково важливим інструментом є застосування централізованих style tokens – узгоджених змінних для кольорів, шрифтів, відступів та інших параметрів, які дозволяють підтримувати єдиний візуальний стиль у всіх MFE. Такий підхід забезпечує цілісність користувацького досвіду, зменшує ризик фрагментації та спрощує інтеграцію нових модулів у систему.

– ***Надмірні витрати продуктивності*** *(performance overhead)* є поширеним викликом у мікрофронтендній архітектурі, який виникає внаслідок надмірної кількості роутерів, дублювання логіки та зайвих переходів між модулями. У системах, де кожен мікрофронтенд має власний маршрутизатор, відсутність узгодженої стратегії призводить до паралельної обробки навігаційних подій, що створює додаткове навантаження на клієнтську частину. Причиною проблеми є фрагментація навігаційної логіки та відсутність централізованих механізмів оптимізації, що ускладнює інтеграцію модулів у спільне середовище. *Наслідки* проявляються у зниженні продуктивності системи, збільшенні часу відгуку та погіршенні ключових показників взаємодії, як-от TTI та First Input Delay (FID) [25]. У великих застосунках це може призвести до накопичення затримок при переходах між модулями, зростання споживання ресурсів та зниження узгодженості користувацького досвіду. Для *усунення* проблеми доцільним є впровадження механізмів *оптимізації маршрутизаторів*, що передбачає зменшення кількості незалежних роутерів та їхню інтеграцію у спільний навігаційний шар. Ефективним рішенням є застосування *кешування маршрутів та стану*, що дозволяє повторно використовувати результати попередніх переходів і знижує навантаження на систему. Додатково важливим є *мінімізація дублювання логіки*, яка може бути досягнута шляхом винесення спільних функцій у централізовані утиліти або сервісні модулі. Такий підхід забезпечує скорочення обчислювальних витрат, підвищує узгодженість роботи системи та покращує продуктивність при масштабуванні. Подолання надмірних витрат продуктивності у мікрофронтенд-архітектурі потребує комплексної оптимізації навігаційної логіки, що поєднує інтеграцію роутерів, застосування кешування та

усунення дублювання функціоналу, що у сукупності забезпечує стабільність і ефективність роботи системи.

– ***Обмеженість ресурсів команд*** є поширеним викликом у мікрофронтендній архітектурі, зумовленим невеликим розміром команд та браком досвіду у розробників. За таких умов інтеграція окремих мікрофронтендів у спільну систему відбувається повільніше, а ризик помилок у конфігурації значно зростає. *Причиною* проблеми є недостатня кількість людських та технічних ресурсів, що обмежує можливості команд у впровадженні складних механізмів маршрутизації, синхронізації стану та оптимізації продуктивності. *Наслідки* проявляються у затримках під час інтеграції нових модулів, зниженні стабільності системи та виникненні конфігураційних помилок, які можуть призвести до порушення навігаційної логіки або некоректного відображення інтерфейсу. У довгостроковій перспективі це негативно впливає на масштабованість архітектури та ускладнює підтримку застосунку. Для *усунення* цієї проблеми доцільним є використання *готових бібліотек та шаблонів*, які стандартизують процеси інтеграції та зменшують кількість ручних налаштувань [26]. Ефективним рішенням також є впровадження *централізованих інструментів управління конфігурацією*, які дозволяють автоматизувати перевірку сумісності модулів та знижують ризик помилок. Крім того, застосування спільних практик та узгоджених інженерних стандартів сприяє підвищенню продуктивності команд і забезпечує більш стабільну інтеграцію MFE у загальну систему. Тим самим подолання обмеженості ресурсів команд у мікрофронтенд-архітектурі потребує використання готових рішень, централізованих інструментів та стандартизованих шаблонів, що дозволяє компенсувати брак досвіду та забезпечити ефективну інтеграцію модулів.

– ***Невідповідність масштабу*** є характерним викликом у застосуванні мікрофронтендної архітектури, що виникає у випадках використання цього підходу для малих проєктів. Мікрофронтенди розроблялися як інженерне рішення для великих систем із високим рівнем складності, де необхідна автономність команд та модульність архітектури. У невеликих застосунках впровадження MFE призводить до надмірної складності, оскільки потребує додаткових механізмів маршрутизації, синхронізації стану та інтеграції модулів, які не є критично необхідними для невеликих команд чи обмежених функціональних вимог. *Причиною* проблеми є невиправдане використання мікрофронтендів у контекстах, де їхня архітектурна потужність не відповідає реальним потребам проєкту. *Наслідки* проявляються у необґрунтованих витратах часу та ресурсів, зростанні складності конфігурації та підтримки, а також у зниженні ефективності розробки. У таких випадках команди стикаються з труднощами інтеграції, які перевищують вигоди від модульності, що негативно позначається на продуктивності та швидкості розгортання. Для *усунення* цієї проблеми доцільним є *вибір архітектури відповідно до масштабу проєкту*. Якщо система є невеликою та не потребує високого рівня модульності, ефективним рішенням стає застосування спрощених підходів, зокрема *Single Page Application (SPA)* без поділу на окремі MFE [27]. Це дозволяє уникнути надмірної складності, зменшити витрати на підтримку та забезпечити швидшу інтеграцію нових функцій. У випадках, коли проєкт поступово зростає, можливим є поступове впровадження мікрофронтендів лише для тих частин системи, які дійсно потребують автономності та масштабованості. Тим самим невідповідність масштабу у використанні мікрофронтендів може бути подолана шляхом критичного аналізу потреб проєкту та вибору архітектурних рішень, що відповідають його розміру й складності. Це забезпечує оптимальний баланс між витратами та вигодами, зберігаючи ефективність розробки та підтримки системи.

– ***Синхронізація стану між MFE*** є одним із ключових викликів мікрофронтендної архітектури. Кожен мікрофронтенд може мати власну систему управління станом (наприклад, Redux, Vuex, Zustand чи інші локальні сховища), що забезпечує автономність його роботи [3]. Однак у межах комплексного застосунку така автономність призводить до проблеми узгодженості даних, оскільки відсутність єдиного механізму синхронізації створює ризик дублювання бізнес-логіки та розривів у взаємодії між модулями. *Причиною* проблеми є фрагментація state-management, коли кожен модуль незалежно управляє власним станом без координації з іншими. Це ускладнює обмін даними між MFE та призводить до неконсистентності, особливо коли кілька модулів взаємодіють зі спільними бізнес-об'єктами (наприклад, користувачами, кошиком покупок чи налаштуваннями профіля). *Наслідки* призводять до дублювання даних, розбіжностях у бізнес-логіці та непередбачуваній поведінці системи. Користувач може отримати різні результати при взаємодії з різними модулями, що знижує довіру до застосунку та ускладнює його підтримку. У великих системах це також негативно впливає на масштабованість та продуктивність, оскільки кожна команда змушена вирішувати проблему узгодженості самостійно. Для *усунення* цієї проблеми застосовуються кілька інженерних стратегій. Найбільш ефективним є впровадження *централізованого state-management API*, який виступає спільним інтерфейсом для всіх MFE та забезпечує узгодженість даних. Додатковим рішенням є створення *«state-gateway»*, що інкапсулює логіку управління станом і надає уніфікований доступ до бізнес-даних. У випадках, коли централізація є недоцільною, застосовується *подієва шина (event bus)* або *брокер стану*, які синхронізують дані між модулями через події, забезпечуючи слабке зв'язування та гнучкість інтеграції. Отже, синхронізація стану між MFE є критичною умовою стабільності та узгодженості мікрофронтенд-архітектури. Використання централізованих API, state-gateway або подієвих механізмів дозволяє уникнути дублювання логіки, забезпечити консистентність даних та підвищити масштабованість системи.

Мікрофронтенд-архітектура відкриває нові можливості для масштабування та автономності команд, проте створює низку викликів у сфері маршрутизації, продуктивності, узгодженості UI/UX та організації процесів. Впровадження вищеописаних рекомендацій дозволяє суттєво підвищити надійність, стабільність і масштабованість маршрутизаційної моделі у MFE-середовищі. Важливо підкреслити, що більшість описаних механізмів реалізуються не у вигляді ізольованих рішень, а як частина загальної інфраструктури архітектури застосунку, що потребує стратегічного планування ще на ранніх етапах розробки.

ВИСНОВКИ ТА ПЕРСПЕКТИВИ ПОДАЛЬШИХ ДОСЛІДЖЕНЬ

Маршрутизація в мікрофронтендній архітектурі постає не лише як технічний механізм, а як системоутворюючий фактор, який визначає узгодженість, масштабованість і якість користувацького досвіду. Її виклики охоплюють як інженерні, так і організаційні аспекти, а запропоновані рішення демонструють необхідність балансу між автономністю модулів та централізованим управлінням. Сформовані рекомендації створюють методологічну основу для побудови стійких і гнучких клієнтських систем, здатних адаптуватися до динаміки технологічного розвитку. Подальші дослідження у цій сфері відкривають перспективи формалізації стандартів та інтеграції новітніх інструментів, що забезпечать новий рівень зрілості мікрофронтендних застосунків.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Pereiaslavska, S. & Smahina, O. (2023). Designing the routing level in microservice architectures on the Spring platform. *Innovative technologies and scientific solutions for industries*, 3(25), 64–78. https://doi.org/10.30837/ITSSI.2023.25.064
2. Shruthi, A. (2025). Building Microfrontend Architecture with Flutter for Modular Apps. *The American Journal of Engineering and Technology*, 07, 142-150. https://doi.org/10.37547/tajet/Volume07Issue05-12.
3. Kurian, G. & Sanjeev, K. (2024). Integrating Microservices and Microfrontends: A Comprehensive Literature Review on Architecture, Design Patterns, and Implementation Challenges. *Journal of Scientific Research and Technology*, 1-12. https://doi.org/10.61808/jsrt115.
4. Nikulina, O. & Khatsko, N. (2023). Method of converting the monolithic architecture of a front-end application to microfrontends. *Bulletin of National Technical University KhPI Series System Analysis Control and Information Technologies*, 79-84. https://doi.org/10.20998/2079-0023.2023.02.12.
5. Kungne, W., Kouamou, G., & Ayang, P. (2025). GenMicro: A Tool for Generating Microservice Architectures with In-Depth Microservice Design. *Journal of Computer Science*, 21, 729-740. https://doi.org/10.3844/jcssp.2025.729.740.
6. Meihong, Y., Xiaoli, W., Binlei, C., Yinglong, W., & Ying, G. (2025). Full stack optimization of microservice architecture: systematic review and research opportunity. *Cluster Computing*, 28. https://doi.org/10.1007/s10586-025-05690-6.
7. Slivka, S. (2024). Microservices architecture for ERP systems. *Bulletin of Cherkasy State Technological University*, 32-42. https://doi.org/10.62660/bcstu/4.2024.32.
8. Binildas, Ch. & Tarun, T. (2025). Microservices Architecture. *Practical Microservices Architectural Patterns*, 61-90. https://doi.org/10.1007/979-8-8688-1606-2_4.
9. Kažemaks, D. & Decouchant, J. (2025). SoK: Microservice Architectures from a Dependability Perspective. *arXiv*. https://doi.org/10.48550/arXiv.2503.03392.
10. Chaplia, O. & Klym, H. (2024). Microservice architecture for cyber-physical systems. *Visnyk of Kherson National Technical University. Information technologies*, 2(89), 242-250. https://doi.org/10.35546/kntu2078-4481.2024.2.34.
11. Borovskova, Ye. A. (2025). Investigating the performance impact of lazy loading in web applications. *Infocommunication and computer technologies*, 1(09), 95-101. https://doi.org/10.36994/2788-5518-2025-01-09-12
12. Single-spa. https://single-spa.js.org/docs
13. window.history API. https://developer.mozilla.org/en-US/docs/Web/API/History_API
14. Butsch, Th., Bell, R., & Warren, V. (2025). The incident command self-managed organization: A hybrid model for adaptive organizational resilience. *Development and Learning in Organizations.* https://doi.org/10.1108/DLO-07-2025-0254.
15. Graph QL best practices. URL: https://graphql.org/learn/best-practices/
16. Sutharsica, A. & Arambepola, N. (2025). Micro-Frontend Architecture: A Comparative Study of Startups and Large Established Companies-Suitability, Benefits, Challenges, and Practical Insights, *International Research Conference on Smart Computing and Systems Engineering (SCSE)*, 1-6. https://doi.org/10.1109/SCSE65633.2025.11030972.
17. Configuring single-spa. https://single-spa.js.org/docs/configuration/
18. Working with history API. https://developer.mozilla.org/en-US/docs/Web/API/History_API/Working_with_the_History_API
19. SSR in microfrontends. https://single-spa.js.org/docs/ssr-overview/
20. Vivek, J. (2022). Optimizing web performance with lazy loading and code splitting. *International Journal of Core Engineering & Management*, 7(3), 193-199. https://doi.org/10.5281/zenodo.14956631.
21. What is TTI. https://web.dev/articles/tti
22. FCP overview. https://web.dev/articles/fcp
23. Jackson, C. (2019). Micro Frontends. *ThoughtWorks Technology Radar*. https://martinfowler.com/articles/micro-frontends.html
24. Трофименко, О.Г., Манаков, С.Ю, Корнійчук, М.М., Лобода, Ю.Г., Чикунов, П.О. (2025). Модальні вікна в інтерфейсі користувача засобами React/Next.js. *Системи та технології*, 69(1), 92-102. https://doi.org/10.32782/2521-6643-2025-1-69.11
25. Hyseni, L., Dermaku, A., & Dika, Z. (2025). Evaluating Web Frameworks for Personal Learning Decision-Making: A Comparative Analysis. *International Journal of Computational and Experimental Science and Engineering*. https://doi.org/11. 10.22399/ijcesen.1845.
26. Манаков, С.Ю., Трофименко, О.Г., Чикунов, П.О., Гура, В.І. (2025). ШІ-керовані системи розробки кросплатформних застосунків. *Відкриті інформаційні та комп'ютерні інтегровані технології,* 105, 184-199. https://doi.org/10.32620/oikit.2025.105.15
27. Single-spa recommended setup. https://single-spa.js.org/docs/recommended-setup/

КІБЕРБЕЗПЕКА: освіта, наука, техніка
CYBERSECURITY: EDUCATION, SCIENCE, TECHNIQUE
№ 3 (31), 2025
ISSN 2663 - 4023

Trofymenko Olena
PhD, Associate Professor,
Associate Professor at the Department of Information Technologies
of the National University "Odessa Law Academy", Odesa, Ukraine,
ORCID: 0000-0001-7626-0886
trofymenko@onua.edu.ua

Manakov Serhii
PhD, Associate Professor,
Associate Professor at the Department of Information Technologies
of the National University "Odessa Law Academy", Odesa, Ukraine,
ORCID: 0000-0001-5930-4592
s.manakov@meta.ua

Chykunov Pavlo
PhD, Associate Professor,
Associate Professor at the Department of Information Technologies
of the National University "Odesa Law Academy", Odesa, Ukraine,
ORCID: 0000-0003-4959-774412:27
pavel@onua.edu.ua

Loboda Yuliia
PhD, Associate Professor,
Associate Professor at the Department of Information Technologies
of the National University "Odessa Law Academy", Odesa, Ukraine,
ORCID: 0000-0001-7083-552X
jul.loboda@gmail.com

Hurin Yevhen
student
of the National University "Odessa Law Academy", Odesa, Ukraine,
ORCID: 0009-0000-4640-6581
egurin443@gmail.com

ROUTING IN MICROFRONTEND APPLICATIONS: ARCHITECTURAL APPROACHES, CHALLENGES AND SOLUTIONS

Abstract. The article focuses on the analysis and comparison of routing approaches in microfrontend architecture, which represents one of the key areas of modern web development. The relevance of the topic is determined by the rapid dynamics of technological progress and the constant growth of requirements for scalability, modularity, and flexibility of client systems. Microfrontend architecture is considered the frontend equivalent of the microservice paradigm, ensuring the autonomy of individual modules, their independent testing and updating, while at the same time generating a number of specific challenges, among which routing occupies a central place. Routing defines the consistency of interactions between subsystems, affects performance and user experience quality, and forms the basis for further scalability of the system. The article analyzes modern approaches to organizing routing in microfrontend architecture, namely global, local, and hybrid models. It is shown that the global approach provides centralized control and navigation consistency but reduces module autonomy and creates the risk of centralized failures. The local approach, by contrast, guarantees team independence and development speed but leads to route conflicts, issues with synchronizing navigation history, and reduced consistency of the user experience. Hybrid models attempt to combine the advantages of both approaches but leave unresolved the issues of synchronization across different frameworks and SSR/SSG support. Particular attention is paid to key challenges arising in the routing process, including URL conflicts, navigation history coordination, SEO and indexing problems, difficulties with lazy loading of modules, dependence on specific frameworks, UI/UX consistency, performance overhead, limited team resources, and the mismatch between the scale of microfrontends and the needs of small projects. Additionally, the problem of state synchronization between modules, which is critical for ensuring consistency of business logic and data, is examined. The paper proposes practical solutions to address these challenges.

Keywords: microfrontend architecture; routing; routing challenges; performance; scalability; framework; web development; UI/UX.

REFERENCES

1. Pereiaslavska, S. & Smahina, O. (2023). Designing the routing level in microservice architectures on the Spring platform. *Innovative technologies and scientific solutions for industries*, 3(25), 64–78. https://doi.org/10.30837/ITSSI.2023.25.064
2. Shruthi, A. (2025). Building Microfrontend Architecture with Flutter for Modular Apps. *The American Journal of Engineering and Technology*, 07, 142-150. https://doi.org/10.37547/tajet/Volume07Issue05-12.
3. Kurian, G. & Sanjeev, K. (2024). Integrating Microservices and Microfrontends: A Comprehensive Literature Review on Architecture, Design Patterns, and Implementation Challenges. *Journal of Scientific Research and Technology*, 1-12. https://doi.org/10.61808/jsrt115.
4. Nikulina, O. & Khatsko, N. (2023). Method of converting the monolithic architecture of a front-end application to microfrontends. *Bulletin of National Technical University KhPI Series System Analysis Control and Information Technologies*, 79-84. https://doi.org/10.20998/2079-0023.2023.02.12.
5. Kungne, W., Kouamou, G., & Ayang, P. (2025). GenMicro: A Tool for Generating Microservice Architectures with In-Depth Microservice Design. *Journal of Computer Science*, 21, 729-740. https://doi.org/10.3844/jcssp.2025.729.740.
6. Meihong, Y., Xiaoli, W., Binlei, C., Yinglong, W., & Ying, G. (2025). Full stack optimization of microservice architecture: systematic review and research opportunity. *Cluster Computing*, 28. https://doi.org/10.1007/s10586-025-05690-6.
7. Slivka, S. (2024). Microservices architecture for ERP systems. *Bulletin of Cherkasy State Technological University*, 32-42. https://doi.org/10.62660/bcstu/4.2024.32.
8. Binildas, Ch. & Tarun, T. (2025). Microservices Architecture. *Practical Microservices Architectural Patterns*, 61-90. https://doi.org/10.1007/979-8-8688-1606-2_4.
9. Kažemaks, D. & Decouchant, J. (2025). SoK: Microservice Architectures from a Dependability Perspective. *arXiv*. https://doi.org/10.48550/arXiv.2503.03392.
10. Chaplia, O. & Klym, H. (2024). Microservice architecture for cyber-physical systems. *Visnyk of Kherson National Technical University. Information technologies*, 2(89), 242-250. https://doi.org/10.35546/kntu2078-4481.2024.2.34.
11. Borovskova, Ye. A. (2025). Investigating the performance impact of lazy loading in web applications. *Infocommunication and computer technologies*, 1(09), 95-101. https://doi.org/10.36994/2788-5518-2025-01-09-12
12. Single-spa. https://single-spa.js.org/docs
13. window.history API. https://developer.mozilla.org/en-US/docs/Web/API/History_API
14. Butsch, Th., Bell, R., & Warren, V. (2025). The incident command self-managed organization: A hybrid model for adaptive organizational resilience. *Development and Learning in Organizations.* https://doi.org/10.1108/DLO-07-2025-0254.
15. Graph QL best practices. URL: https://graphql.org/learn/best-practices/
16. Sutharsica, A. & Arambepola, N. (2025). Micro-Frontend Architecture: A Comparative Study of Startups and Large Established Companies-Suitability, Benefits, Challenges, and Practical Insights, *International Research Conference on Smart Computing and Systems Engineering (SCSE)*, 1-6. https://doi.org/10.1109/SCSE65633.2025.11030972.
17. Configuring single-spa. https://single-spa.js.org/docs/configuration/
18. Working with history API. https://developer.mozilla.org/en-US/docs/Web/API/History_API/Working_with_the_History_API
19. SSR in microfrontends. https://single-spa.js.org/docs/ssr-overview/
20. Vivek, J. (2022). Optimizing web performance with lazy loading and code splitting. *International Journal of Core Engineering & Management*, 7(3), 193-199. https://doi.org/10.5281/zenodo.14956631.
21. What is TTI. https://web.dev/articles/tti
22. FCP overview. https://web.dev/articles/fcp
23. Jackson, C. (2019). Micro Frontends. *ThoughtWorks Technology Radar*. https://martinfowler.com/articles/micro-frontends.html

24. Trofymenko, O.G., Manakov, S.Yu., Korniichuk, M.M., Loboda, Yu.V., Chykunov, P.O. (2025). Modal windows in the in UI with React/Next.js. *Systems and Technologies*, 69(1), 92-102. https://doi.org/10.32782/2521-6643-2025-1-69.11
25. Hyseni, L., Dermaku, A., & Dika, Z. (2025). Evaluating Web Frameworks for Personal Learning Decision-Making: A Comparative Analysis. *International Journal of Computational and Experimental Science and Engineering*. https://doi.org/11. 10.22399/ijcesen.1845.
26. Manakov, S.Yu., Trofymenko, O.G., Chykunov, P.O., Гура, B.I. (2025). AI-driven cross-platform application development systems. *Open Information and Computer Integrated Technologies*, 105, 184-199. https://doi.org/10.32620/oikit.2025.105.15
27. Single-spa recommended setup. https://single-spa.js.org/docs/recommended-setup/