Русначенко Михайло Олександрович, магістр

Арсенюк Ігор Ростиславович, к. т. н., доцент

ВНТУ, Вінниця

**ПЕРСПЕКТИВНІСТЬ ВИКОРИСТАННЯ МІКРОФРОНТЕНДІВ ДЛЯ РЕАЛІЗАЦІЇ ЗАСТОСУНКУ КОНТРОЛЮ ПРИЙНЯТТЯ ЛІКІВ**

Під час реалізації застосунків із великою кількістю функціональних можливостей, таких, наприклад, як додаток для контролю прийняття ліків [1] із часом постає проблема масштабування проекту. Зі збільшенням кількості модулів та функцій стрімко зростає кількість витрачених ресурсів для розробки та підтримки проекту. Проаналізуємо доцільність застосування монолітної або мікросервісної архітектури на прикладі застосунку для контролю прийняття ліків, який має досить великий набір функціоналу, зокрема, персональний кабінет, історія прийняття ліків, модуль нагадування про прийом ліків, інтелектуальний модуль, календар та інше.

Монолітна архітектура — це традиційна модель програмного забезпечення, що побудована як уніфікована одиниця, самодостатня та незалежна від інших програм. Ця архітектура представляє собою окрему велику обчислювальну мережу з єдиною кодовою базою, що об’єднує всі бізнес-завдання. Слово «моноліт» часто приписують чомусь великому та льодовиковому, що не далеко від істини монолітної архітектури для розробки програмного забезпечення [2]. Для того, щоб внести зміни у програмне забезпечення такого типу, потрібно оновити весь стек, отримавши доступ до бази коду та створити і розгорнувши оновлену версію інтерфейсу служби. Це робить оновлення обмежувальними та забирає досить багато часу розробника [2].

Мікросервісна архітектура — це архітектурний стиль для побудови програмних систем шляхом розбиття складних програм на набір невеликих, слабко пов’язаних і незалежно розгорнутих сервісів [3]. Кожен сервіс в архітектурі мікросервісів відповідає за певний, чітко визначений набір функцій і може свзаємодіяти з іншими сервісами через мережу [4].

На сьогоднішній день мікросервісна архітектура масово використовується для розробки серверного програмного забезпечення, проте майже не використовується для клієнтського, через свою новизну саме для клієнтських застосунків та складність у налаштуванні процесів. Проте даний підхід має значно більше переваг для великих проектів порівняно з монолітною архітектурою.

Переваги мікросервісної архітектури (мікрофронтенди) відносно монолітної:

1. *Масштабування:* ми можемо масштабувати окремі мікроіфронтенди залежно від їхніх конкретних вимог.
2. *Незалежна розробка*: різні команди та розробники можуть працювати над окремими мікрофронтендами, не впливаючи один на одного, що сприяє паралельній розробці та пришвидшить кінцевий результат.
3. *Модульність*: мікрофронтенди дозволяють розділяти програмне забезпечення на менші самостійні модулі, що полегшує керування та незалежну розробку окремих функцій або компонентів.
4. *Гнучкість*: ми можемо використовувати різні технології, фреймворки або бібліотеки в окремих мікрофронтендах без необхідності робити зміни в інших сервісах.
5. *Ізоляція*: ізоляція між мікрофронтендами зменшує ризик помилок в одному модулі, що також приводить до того, що помилка в одному модулі не виведе з ладу всю програму.

Проте є й певні недоліки мікрофронтендів, до яких, у першу чергу слід віднести такі:

1. *Складність*: мікрофронтенди створюють додаткову складність із точки зору маршрутизації, зв’язку та координації між різними компонентами.
2. *Проблеми з комунікацією*: зв’язок між мікрофронтендами може бути складним і потребуватиме спеціальних рішень або бібліотек для забезпечення плавної інтеграції та обміну даними.
3. *Контроль версій і сумісностей*: керування версіями та забезпечення сумісності всіх мікрофронтендів між собою може бути складним завданням, що може призвести до потенційних проблем з інтеграцією.

Для того, щоб реалізувати мікросервісну архітектуру для клієнтського програмного забезпечення зазвичай використовують ModuleFederationPlugin [5]. ModuleFederationPlugin дозволяє збірці надавати або споживати модулі з іншими незалежними збірками під час виконання, проте можна самостійно налаштувати процес збірки.

Поради щодо реалізації мікросервісної архітектури для React застосунків:

* Потрібно створити Core Application який буде відповідати за підвантаження збірок наших сервісів, вмонтовування та розмонтовування скриптів у DOM — Document Object Model [6].
* Кожен дочірній сервіс, що буде використовуватись у Core повинен мати однакову структуру збірки та функціонал, що дозволить глобально викликати (з об’єкта window) методи для монтування та розмонтування даного сервісу.
* Важливо, щоб у Core сервіса був доступ до збірок дочірніх сервісів, тому потрібно розміщувати сервіси у межах Docker контейнера, або ж ігнорувати CORS — Cross-Origin Resource Sharing [7] (не рекомендується через можливі вразливості сервісів).
* Використання окремої бібліотеки з набором повторюваних компонентів та функцій допоможе уникнути дублювання коду в сервісах.
* Для комунікації між сервісами можна використовувати всі доступні методи браузера такі як: localStorage, sessionStorage, CustomEvent, window, serviceWorkers та інші.

Аналізуючи всі вищенаведені переваги та недоліки можна дійти висновку, що за наявності ресурсів мікросервісна архітектура краще підходить для розробки великих клієнтських застосунків, зокрема і таких, як застосунок для контролю прийняття ліків, через свою гнучкість та масштабування. Адже над кожним модулем можуть незалежно працювати різні команди та розробники і при зміні у певному модулі не потрібно збирати увесь проект, а лише даний модуль. Також через можливість розділення великих модулів, таких як календар та модуль історії прийняття ліків, функціонал стає ізольованим і у випадку помилки в одному з модулів застосунок і далі працюватиме. Через розділення на мікрофронтенди ми маємо можливість використовувати різний набір компонентів для персонального кабінету та календаря.

**Література**

1. Русначенко М. О., Арсенюк І. Р. Обґрунтування доцільності використання react native для додатку прийому ліків. / Тези доповідей LI Науково-технічної конференції факультету інтелектуальних інформаційних технологій та автоматизації. – Вінниця: ВНТУ, 2022. URL: https://conferences.vntu.edu.ua/index.php/all-fksa/all-fksa-2022/paper/view/15904/13353)
2. Monolithic Architecture. URL: <https://www.atlassian.com/microservices/microservices-architecture/microservices-vs-monolith>.
3. Microservice Architecture. URL: <https://cloud.google.com/learn/what-is-microservices-architecture>.
4. Арсенюк І. Р. Комп'ютерні мережі: навчальний посібник / І. Р. Арсенюк, А. А. Яровий, І. Д. Івасюк. – Вінниця : ВНТУ, 2013. – 272 с.
5. ModuleFederationPlugin. URL: [https://webpack.js.org/plugins/module-federation-plugin](https://webpack.js.org/plugins/module-federation-plugin/).
6. DOM. URL: https://developer.mozilla.org/en-US/docs/Web/API/Document\_Object\_Model/Introduction.
7. CORS. URL: https://developer.mozilla.org/en-US/docs/Web/HTTP/CORS.