УДК 004.4
DOI: https://doi.org/10.26642/ten-2020-1(85)-128-134

К.Р. Колос, д.пед.н., доц., проф.
А.І. Баранов, магістрант
Р.В. Петросян, ст. викладач
Державний університет «Житомирська політехніка»

Аналіз побудови клієнтських частин вебдодатків на основі Microfrontend підходу

У сучасному світі вебтехнології проникають майже у всі аспекти життя людини. З їх допомогою можна створювати, налагоджувати та синхронізувати облікові записи на різних пристроях, керувати процесами бізнесу. Водночас постає проблема підвищення рівня ефективності проєктування інформаційних систем, які будуть надавати доступ до різноманітної інформації з будь-якого куточка світу, де є доступ до мережі Інтернет. Все більше вебдодатків з'являється з кожним днем, потужність комп'ютерної техніки зростає, а разом з цим, підвищується рівень їх складності, що в свою чергу обумовлює збільшення зусиль та часу на їх підтримку. Це зумовлює потребу в обґрунтуванні використання Microfrontend підходу. Саме тому в статті вирізнено переваги та недоліки розробки мікросервісів. Обґрунтовано основні підходи до побудови мікрофронтенд додатка: композиція шаблонів на стороні сервера; інтеграція при збірці; інтеграція під час виконання з використанням HTML-елемента iframe; інтеграція під час виконання за допомогою засобів мови програмування JavaScript; інтеграція під час виконання з використанням технології web components. На основі аналізу зазначених підходів виокремлено і охарактеризовано переваги: поступові (інкрементальні) оновлення; прості, не пов'язані між собою кодові бази; незалежне розгортання; автономні команди, та недоліки: значний об'єм коду, що завантажується; відмінності середовища; складність управління розробки мікрофтондендів. Тому перед переходом до такої архітектури необхідно враховувати: наявність ресурсів для достатнього рівня автоматизації та забезпечення управління додатковою необхідною інфраструктурою; зміни в процесі розробки, тестування та випуску у великій кількості компонентів; зростання складності, пов'язаної з використанням більшої кількості інструментів та підходів до розробки; забезпечення достатнього рівня якості, узгодженості та управління великою кількістю кодових баз. Отже, обираючи архітектуру мікрофронтендів слід зважити і проаналізувати наявність технічної та організаційної доцільності для прийняття такого підходу.

Ключові слова: *мікросервіси; вебдодатки; Microfrontend підхід; архітектура*.

Актуальність теми. У сучасному світі вебтехнології проникають майже у всі аспекти життя людини. З їх допомогою можна створювати, налагоджувати та синхронізувати облікові записи на різних пристроях, керувати процесами бізнесу.

Водночас постає проблема підвищення рівня ефективності проєктування інформаційних систем, які будуть надавати доступ до різноманітної інформації з будь-якого куточка світу, де є доступ до мережі Інтернет. З кожним днем з'являється все більше вебдодатків, потужність комп'ютерної техніки зростає, а разом з цим, підвищується рівень їх складності, що, в свою чергу, обумовлює збільшення зусиль та часу на їх підтримку, зокрема: оновлювати змістове наповнення вебдодатків і технологій, які використовуються під час розробки для збільшення швидкодії додатків, зменшувати час на обрахунки, покращувати зручність роботи із застосунками тощо.

Зі зростанням розміру додатка збільшується кількість часу необхідного для того, щоб новий співробітник розібрався в архітектурі додатка і почав розробляти новий функціонал системи. Водночас зростає ймовірність внесення дефектів у різних частинах системи під час розробки нового функціонала.

Тому важливим є уважний та ретельний підхід до вибору архітектури системи, що дозволить визначити подальший розвиток всього проєкту. Побудова великих і складних систем є непростою задачею, для розв'язання якої застосовується як монолітний, так і розподілений підходи залежно від контексту та бізнес вимог.

Аналіз останніх досліджень та публікації, на які спирається автор. Зокрема, М.Фаулер [1], С.Ньюмен [2], І.Надерешвілі [3], І.Івентс [4], М.Фаулер [5], М.Бауман [6], Л.Цзін [7], С.Невман [8], А.Кумар [9], Р.Купер [10] та ін. у своїх працях висвітлюють різні варіанти побудови додатків, що базуються на архітектурі мікрофронтендів.

Мета статті – проаналізувати переваги та недоліки кожного варіанта побудови додатків, що базуються на архітектурі мікрофронтендів, що дозволить обрати варіант, який найкраще відповідатиме поставленим завданням, виокремити ключові компоненти системи і сформувати вимоги до неї.

Викладення основного матеріалу. Проєктування та розробка якісного програмного забезпечення є досить важливими в сучасному інформаційному світі. Під час розвитку галузі інформаційних технологій обґрунтовано низку підходів і концепцій побудови складних програмних систем. Архітектура системи є показником якісно побудованої системи, що правильно описує та формально є її моделлю. Основне завдання архітектури як набору структурних компонентів, які зв'язані між собою та формують функціонал системи, – це управління складністю, якісне та доцільне відображення предметної області.

Тривалий час «монолітна архітектура» займала ключове місце під час побудови як серверних, так і клієнтських вебдодатків. Система, яка використовує зазначений підхід, являє собою моноліт, що розміщений на одному сервері, запущений в одному процесі і виконує при цьому всю бізнес-логіку системи.

Моноліт на стороні сервера піддається лише горизонтальному масштабуванню, яке досягається за рахунок запуску великої кількості окремих серверів із кожним окремим монолітом. З часом з'являлися інші підходи до розробки серверних додатків, серед яких є сервіс-орієнтована архітектура (SOA), яка, на відміну від моноліту, є розподіленою системою, що обмінюється повідомленнями, використовуючи певні протоколи.

З розвитком індустрії інформаційних технологій розроблено новий підхід до організації SOA – мікросервісна архітектура (MSA). Її можна вважати частиною SOA, яка має певні відмінності від класичної SOA. Основна відмінність – це незначна кількість коду, тоді як в SOA об'єм кодової бази не має значення. Також важливою особливістю MSA є те, що кожен мікросервіс обмежено власним контекстом – бізнес-задачею. Обмежень за кількістю сервісів не існує; важливо, щоб кожен з них працював лише над своєю бізнес-задачею. Для міжсервісної комунікації використовують стандартні протоколи передачі даних (наприклад HTTP) і в той же час сервіс може мати свій протокол для спілкування з іншими. Сервіси можуть бути написані з використанням різних технологій, фреймворків і мов програмування, використовуючи необмежену кількість бібліотек. Зазвичай, відбувається децентралізація збереження даних, тобто кожен з сервісів має власну базу даних [2].

Серед переваг мікросервісної архітектури можна виокремити:

– низьку зв'язність між компонентами системи;

– відносно просте розгортання: кожен сервіс розгортається незалежно від інших, в той час, як під час використання монолітної архітектури вся система розгортається разом в одному процесі і неможливі внесення динамічних змін;

– просте масштабування системи як у горизонтальному, так і у вертикальному напрямку;

– відносна відмовостійкість: під час недієздатності окремих сервісів система залишається працездатною;

– застосування різних технологій, фреймворків і мов програмування, а також різних типів сховищ даних;

– під час створення сервісів можлива організація невеликих груп програмістів, відповідальних за розробку сервісу.

Серед недоліків мікросервісного підходу можна вирізнити такі:

– складність розробки та заміни членів команди, яка прямо пропорційно залежить від кількості мов програмування, фреймворків та типів баз даних, використаних під час розробки;

– втрата ресурсів під час міжсервісної комунікації, що зумовлює збільшення часу на опрацювання, серіалізацію та десеріалізацію повідомлень;

– проблеми з версіонуванням та підтримкою зворотної сумісності між сервісами;

– значно комплексніше та складніше, порівняно з монолітом, інтеграційне тестування.

Найчастіше розробку додатків, в яких було успішно використано мікросервісний підхід, розпочинали з моноліту, який з часом збільшувався в обсязі. Водночас буває й таке, що проєкт розпочинається з мікросервісного підходу і не досягає мети. Тому найкраще розпочати з розробки моноліту, навіть якщо є впевненість, що проект буде мати значний обсяг. Мікросервіси є досить ефективним архітектурним прийомом, але їх переваги розкриваються лише під час використання у великих і складних системах.

До розробки додатка необхідно насамперед переконатися у його доцільності, корисності та попиті серед користувачів. На початку розробки максимальну цінність має швидкість розробки, а отже, і швидкість отримання відгуків від користувачів. Наступна проблема розробки мікросервісних додатків у тому, що потрібно отримати набір обмежених контекстів, а будь-які переробки та рефакторинг значно складніший при використанні мікросервісів порівняно з монолітом. Побудова моноліту створює можливість легко розділити його на логічні частини, виявити слабкі місця та недоліки системи, виокремити часто вживані фрагменти коду у власні бібліотеки тощо.

Зараз існує значна кількість шляхів реалізації стратегії «спочатку моноліт». Найлогічнішим є проєктування моноліту з чіткою модульною структурою. За правильної реалізації перехід до мікросервісів буде досить простим.

Більш загальним вважається написання моноліту з поступовим відокремленням від нього сервісів. У такому випадку значна частина моноліту залишиться як початкове «ядро» системи, а більша частина нової розробки відбуватиметься в сервісах. Також здебільшого використовують повну заміну моноліту на мікросервіси. Серед недоліків цього підходу є необхідність зупинки розробки нового функціоналу, що найчастіше є неприйнятним для бізнесу.

Існує багато підходів, які можна назвати мікрофронтендовими. Як правило, для кожної сторінки в додатку є мікрофронтенд і єдиний додаток для контейнерів, який:

– надає загальні елементи сторінки, такі як: header та footer вебсторінки, меню тощо;

– вирішує загальні задачі, такі як: аутентифікація, навігація, локалізація, система доступів.

Об'єднує різні мікрофронтенди на сторінку та повідомляє кожному мікрофронтенду, коли, де і як себе відобразити.

Виокремлюють п'ять основних підходів до побудови мікрофронтенд додатка:

– композиція шаблонів на стороні сервера (server side composition);

– інтеграція при збірці;

– інтеграція під час виконання (runtime integration) з використанням HTML-елемента iframe;

– інтеграція під час виконання (runtime integration) за допомогою засобів мови програмування JavaScript;

– інтеграція під час виконання (runtime integration) з використанням технології web components.

Композиція шаблонів на стороні сервера – широко відомий підхід до розробки вебінтерфейсів, який ґрунтується на композиції HTML-сторінок на сервері з різних шаблонів або фрагментів.

При цьому використовується index.html, який містить будь-які загальні елементи сторінки, а потім сервер складає вміст сторінки з фрагментів HTML-файлів:

```
<html lang="en" dir="ltr">
  <head>
    <meta charset="utf-8">
    <title>Feed me</title>
  </head>
  <body>
    <h1> Feed me</h1>
    <!--# include file="$PAGE.html" -->
  </body>
</html>
```

Цей файл обслуговується за допомогою Nginx налаштуванням змінної $PAGE, порівнюючи з URL-адресою, яка запитується:

```
server {
    listen 8080;
    server_name localhost;
    root /usr/share/nginx/html;
    index index.html;
    ssi on;
    # Redirect / to /browse
    rewrite ^/$ http://localhost:8080/browse redirect;
    # Decide which HTML fragment to insert based on the URL
    location /browse {
      set $PAGE 'browse';
    }
    location /order {
      set $PAGE 'order';
    }
    location /profile {
      set $PAGE 'profile'
    }
    # All locations should render through index.html
    error_page 404 /index.html;
}
```

Це досить стандартна композиція на стороні сервера. Такий підхід належить до мікрофронтендів, тому, що розділивши код таким чином, кожен фрагмент є самостійною частиною бізнес-логіки, яку може поставити незалежна команда. Тут не показано, як ці різні HTML-файли потрапляють на вебсервер, але припущення полягає в тому, що кожен з них має свій конвеєр розгортання, який дозволяє розміщувати зміни на одній сторінці, без використання будь-якої іншої сторінки. Для ще більшої незалежності можна використати окремий сервер, що відповідає за надання та обслуговування кожного мікрофронтенда.

Під час **інтеграції при збірці** кожен мікрофронтенд публікується як окремий пакет, а контейнер враховує пакети як окремі залежності. При такому підході файл package.json матиме такий вигляд:

```
{
  "name": "@feed-me/container",
  "version": "1.0.0",
  "description": "A food delivery web app",
  "dependencies": {
    "@feed-me/browse-restaurants": "^1.2.3",
    "@feed-me/order-food": "^4.5.6",
    "@feed-me/user-profile": "^7.8.9"
  }
}
```

На перший погляд такий підхід має сенс. Створюється єдиний пакет JavaScript, що дозволяє уникнути дублювання кодової бази та сторонніх залежностей. Однак такий підхід означає, що доведеться перекомпілювати та випустити кожен мікрофронтенд, щоб внести зміни до будь-якої окремої частини продукту, що впливає на час, що витрачається на збірку.

Інтеграція з використанням iFrame – один з найпростіших підходів до складання програм разом у браузері. За своєю природою iFrame полегшують створення сторінки з незалежних підсторінок. Вони також пропонують хороший рівень ізоляції з точки зору стилізації та глобальних змінних, за рахунок того, що не заважають один одному. Цей підхід є досить непопулярним. Згадана вище проста ізоляція, як правило, робить їх менш гнучкими, ніж інші варіанти. Побудувати інтеграцію між різними частинами програми може бути важко, вони роблять значно складнішою маршрутизацію, взаємодію з історією браузера та глибоку маршрутизацію. Також суттєвим недоліком є неможливість додавання HTML, який виходить за рамки iFrame, що є критичним для відображення спливаючих вікон, підказок тощо.

Найбільш гнучким та найпопулярнішим є **runtime integration за допомогою JavaScript**. Кожен мікрофронтенд включається на сторінку за допомогою тега <script>, і при завантаженні створює глобальну функцію в об'єкті window як точку входу. Потім додаток для контейнерів визначає, який мікрофронтенд має бути завантаженим, і викликає відповідну функцію, щоб повідомити мікрофронтенд, коли і де себе виводити.

На відміну від інтеграції при збиранні, кодова база кожного мікрофронтенда може бути розгорнута окремо. На відміну від iFrames, можна отримати повну гнучкість для побудови інтеграції між мікрофронтендами. Гнучкість цього підходу в поєднанні з незалежним розгортанням робить його найпопулярнішим.

Runtime integration з використанням web components полягає в тому, що кожен мікрофронтенд є кастомним HTML-елементом відповідно до стандарту web components, а не глобальною функцією в об'єкті window.

Кінцевий результат тут схожий на попередній приклад, головна відмінність полягає в тому, що вирішили робити «вебкомпонент». Варто зазначити, що технологія web components має досить низьку підтримку серед популярних браузерів, що є суттєвим недоліком.

Переваги та недоліки розробки мікрофронтендів

За останні роки популярність мікросервісів стрімко зростає, і багато організацій використало цей архітектурний стиль, щоб уникнути обмежень великих монолітів.

Під час розробки вебдодатків найчастіше виникають проблеми з:

– безпечною інтеграцією нового коду в існуючий додаток;

– використанням нових функцій мови JavaScript або іншої з безлічі мов, які можна компілювати в JavaScript і супутніми проблемами зі збіркою додатка;

– масштабуванням розробки з метою розподілу частин додатка між командами.

Ці проблеми можуть негативно вплинути на здатність ефективно доставляти високоякісний продукт своїм клієнтам. Останнім часом приділяється все більше уваги загальній архітектурі та організації кодової бази, необхідних для складної, сучасної веброзробки. Зокрема з'являються схеми розкладання монолітів фронтенда на більш дрібні, прості шматки – **мікрофронтенди**, які можна розробити, протестувати і розгорнути самостійно і які при цьому відображаються для клієнтів як єдиний продукт.

Деякі реалізації мікрофронтендів можуть призвести до дублювання залежностей, збільшуючи кількість трафіка для користувача. Крім того, різке підвищення автономності команди може спричинити роздробленість у роботі ваших команд. Однак вважаємо, що цими ризиками можна керувати і що переваги мікрофронтендів часто перевищують витрати.

Виокремимо переваги архітектури мікрофронтендів.

1. Поступові (інкрементальні) оновлення. Основною перевагою є те, що з'являється більше простору для прийняття конкретних рішень щодо окремих частин продукту, вдосконалення архітектури та залежностей, експериментів з новими технологіями та способами взаємодії. У випадках, якщо заплановано внесення змін до дизайну, залежностей чи елементів, то немає необхідності зупиняти весь

процес розробки, оскільки заплановані зміни можуть бути розроблені і впроваджені поступово й ізольовано.

2. Прості, не пов'язані між собою кодові бази. Вихідний код для кожного окремого мікрофронтенда за визначенням буде значно меншим, ніж вихідний код суцільного моноліту. Ці менші кодові бази, як правило, простіші для роботи розробників. Зокрема, можна уникнути проблем, які виникають внаслідок ненавмисного та неналежного зв'язку між компонентами, що не мають впливати один на одного. Мікрофронтенди спонукають розробників бути чіткими й обдуманими щодо того, як різні частини програми взаємодіють з даними та реагують на події.

3. Незалежне розгортання. Як і у випадку з мікросервісами незалежне розгортання є ключовою перевагою. Воно прискорює час розгортання, що в свою чергу зменшує пов'язані з цим ризики. Кожен мікрофронтенд має розгортатися незалежно від стану кодових баз інших компонентів. Якщо окремий мікрофронтенд готовий бути розгорнутим на live серверах, він має це зробити, і це рішення має залежати лише від команди, яка його проєктує та підтримує.

4. Автономні команди. Розподіл кодових баз та циклів випуску окремих частин додатка дає можливість створити окремі, повністю незалежні команди, які розробляють секцію продукту від ідеї до розгортання на live сервери. Для цього команди мають бути сформовані навколо вертикальної структури бізнесу, а не навколо технічних можливостей. Найпростіший спосіб зробити це – розробити продукт на основі того, що бачать кінцеві користувачі, тому кожен мікрофронтенд інкапсулює одну сторінку програми та належить одній команді. Це приносить більшу згуртованість роботи команд, ніж якби команди були сформовані навколо технічних чи «горизонтальних» питань, таких як стилізація, форми чи валідація.

Мікрофронтенди – це архітектурний стиль, коли незалежні додатки для інтерфейсу складаються у єдине ціле, розрізання великих монолітів на дрібніші, більш керовані шматки, а потім – явне розуміння залежностей між ними. Вибір технологій, кодові бази, команди та процеси випуску повинні мати можливість працювати і розвиватися незалежно один від одного, без зайвої координації.

Зазначимо недоліки архітектури мікрофронтендів.

1. Значний об'єм коду, що завантажується. Незалежно створені пакети JavaScript можуть спричинити дублювання загальних залежностей, збільшивши об'єм трафіка, які необхідно надсилати по мережі кінцевим користувачам. Наприклад, якщо кожен мікрофронтенд містить власну копію React, то це змушує користувачів завантажувати React n разів. Існує прямий взаємозв'язок між продуктивністю сторінки та залученням/конверсією користувачів, і значна частина світу працює з інтернет-підключенням набагато повільнішим, ніж звикли у високорозвинених містах, тому є низка причин дбати про розміри завантажень. Це питання вирішити непросто. Досить складно знайти баланс між бажанням дозволити командам самостійно розробляти свої частини додатка, щоб вони могли працювати автономно, і бажанням будувати програми таким чином, щоб вони могли ділитися загальними залежностями

Один із підходів полягає у зовнішньому застосуванні загальних залежностей від складених пакетів. Недоліком такого підходу є те, що всі частини додатка мають використовувати точні версії цих залежностей. За необхідності змінити версію залежності потрібні будуть великі скоординовані зусилля щодо оновлення та блокування розробки нового функціонала. Це все, чого потрібно намагатися уникати, в першу чергу, завдяки мікрофронтендам. Проте навіть якщо буде вирішено не робити нічого щодо дублювання залежностей, можливо, кожна окрема сторінка все одно буде завантажуватися швидше, ніж під час використання монолітного додатка.

У класичних монолітах, коли завантажується будь-яка сторінка програми, часто завантажується вихідний код та залежності кожної сторінки відразу, а використовуючи архітектуру мікрофронтендів, буде завантажуватися лише код, який потрібний для роботи цієї сторінки. Це може призвести до швидшого початкового завантаження сторінок, але повільнішої наступної навігації, оскільки користувачі змушені повторно завантажувати однакові залежності на кожній сторінці. Якщо не перекривати мікрофронтенди непотрібними залежностями, або якщо користувачі дотримуватимуться лише однієї чи двох сторінок у програмі, можна досягти високого рівня продуктивності, навіть при дублюванні залежностей.

2. Відмінності середовища. Кожний мікрофронтенд має розроблятися без огляду на інші. Можливе навіть виникнення необхідності запустити його в «автономному» режимі на окремій сторінці, а не всередині програми-контейнера. Це може значно спростити розробку, особливо, коли програма-контейнер є складною і застарілою, що є досить поширеним при міграції з моноліту. Однак існують ризики, пов'язані з розробкою в середовищі, яке містить значні відмінності, порівняно з live-середовищем. Якщо контейнер для розробки поводиться інакше, ніж частина програми в яку буде інтегруватися код, що розробляється, то можливе виявлення порушень чи зміни у функціонуванні під час розгортання на live-серверах. Особливо небезпечними можуть бути глобальні стилі, які спричиняються контейнером або іншими мікрофронтендами. Рішенням цієї проблеми схоже на ситуацію, коли особливості середовища мають значення. Необхідно забезпечити регулярну інтеграцію та розгортання

мікрофронтендів на тестові середовища і впроваджувати на цих середовищах процес ручного й автоматизованого тестування з метою найшвидшого виявлення інтеграційних проблем.

3. Складність управління. Використання більш розподіленої архітектури мікрофронтендів неминуче призводить до такого збільшення кількості елементів, якими необхідно керувати – більше репозитаріїв і, як наслідок, – більше інструментів, доменів та серверів, збірки та розгортання конвеєрів. Тому перед переходом до такої архітектури необхідно враховувати:

– наявність ресурсів для достатнього рівня автоматизації та забезпечення управління додатковою необхідною інфраструктурою;

– зміни в процесі розробки, тестування та випуску у великій кількості компонентів;

– зростання складності, пов'язаної з використанням більшої кількості інструментів та підходів до розробки;

– забезпечення достатнього рівня якості, узгодженості та управління великою кількістю кодових баз.

Отже, обираючи архітектуру мікрофронтендів, варто зважити і проаналізувати наявність технічної та організаційної доцільності для прийняття такого підходу.

Висновки та перспективи подальших досліджень. При підході до розв'язання архітектурних задач існує велика кількість ефективних прийомів, тому важливим є уважний та ретельний підхід до вибору архітектури системи, оскільки це дозволяє визначити подальший розвиток всього проєкту. Побудова великих і складних систем є непростою задачею, для розв'язання якої застосовується як монолітний, так і розподілений підходи залежно від контексту та бізнес-вимог.

Існує декілька варіантів побудови додатків, що базуються на архітектурі мікрофронтендів. Аналіз переваг та недоліків кожного з них дозволяє обрати варіант, який максимально відповідає поставленим завданням. Перспективи подальших досліджень вбачаємо у моделюванні архітектури модулів монолітного додатка в мікросервісний.

Список використаної літератури:

1. *Фаулер М.* Архитектура корпоративных программных приложений / *М.Фаулер*. – М. : Издательский дом «Вильяме», 2006 – 544 с.
2. *Ньюмен С.* Создание микросервисов / *С.Ньюмен*. – СПб. : Питер, 2016 – 304 с.
3. Microservice Architecture: Aligning Principles, Practices, and Culture / *I.Nadareishvili, R.Mitra, M.McLarty, M.Amundsen*. – O'Reilly Media, 2016 – 146 p.
4. *Evans E.* Domain-Driven Design: Tackling Complexity in the Heart of Software / *E.Evans.* – Addison-Wesley, 2003 – 560 p.
5. *Fowler M.* Microservices / *M.Fowler* [Електронний ресурс]. – Режим доступу : http://martinfowler.com/articles/microservices.html.
6. *Baumann M.* Micro-Frontends: Studienarbeit / *M.Baumann*. – HSR Hochschule fur Technik Rapperswil, 2019. – 59 p.
7. *Цзін Л.* Мікро-фронтенд – розширення концепції мікропослуг до розвитку «теорії» / *Л.Цзін* [Електронний ресурс]. – Режим доступу : https://insights.thoughtworks.cn/micro-frontends-1/.
8. *Newman S.* Building Microservices: Designing Fine-Grained Systems / *S.Newman*. – USA : O'Reilly Media, 2015. – 280 p.
9. *Kumar A.* Micro Frontends Architecture: Introduction, Design, Techniques & Technology / *A.Kumar*. – USA : Kdp Print Us, 2019. – 124 p.
10. *Kuepper R.* Hands-On Swift 5 Microservices Development: Build microservices for mobile and web applications using Swift 5 and Vapor 4 / *R.Kuepper*. – UK : Pack Publishing Ltd, 2020. – 362 p.

References:

1. Fauler, M. (2006), *Arhitektura korporativnyh programmnyh prilozhenij*, Izdatel'skij dom «Vil'jame», Moskva, 544 p.
2. N'jumen, S. (2016), *Sozdanie mikroservisov*, Piter, SPb., 304 p.
3. Nadareishvili, I., Mitra, R., McLarty, M. and Amundsen, M. (2016), *Microservice Architecture: Aligning Principles, Practices, and Culture*, O'Reilly Media, 146 p.
4. Evans, E. (2003), *Domain-Driven Design: Tackling Complexity in the Heart of Software*, Addison-Wesley, 560 p.
5. Fowler, M., *Microservices*, [Online], available at: http://martinfowler.com/articles/microservices.html
6. Baumann, M. (2019), *Micro-Frontends: Studienarbeit*, HSR Hochschule fur Technik Rapperswil, 59 p.
7. Czin, L., *Mikro-frontend – rozshyrennja koncepcii' mikroposlug do rozvytku «teorii'»*, [Online], available at: https://insights.thoughtworks.cn/micro-frontends-1/
8. Newman, S. (2015), *Building Microservices: Designing Fine-Grained Systems*, O'Reilly Media, USA, 280 p.
9. Kumar, A. (2019), Micro Frontends Architecture: Introduction, Design, Techniques & Technology, Kdp Print Us, USA, 124 p.
10. Kuepper, R. (2020), *Hands-On Swift 5 Microservices Development: Build microservices for mobile and web applications using Swift 5 and Vapor 4*, Pack Publishing Ltd, UK, 362 p.

Колос Катерина Ростиславівна – доктор педагогічних наук, доцент, професор кафедри комп'ютерних наук Державного університету «Житомирська політехніка».

Наукові інтереси:

– комп'ютерні науки;
– комп'ютерно орієнтоване навчальне середовище.

http://orcid.org/0000-0002-1038-8569.

Баранов Андрій Ігорович – магістрант Державного університету «Житомирська політехніка».

Наукові інтереси:

– комп'ютерні науки;
– створення вебдодатків.

Петросян Руслан Валерійович – старший викладач кафедри комп'ютерних наук Державного університету «Житомирська політехніка».

Наукові інтереси:

– методи цифрової обробки;
– штучний інтелект;
– математичне моделювання;
– комп'ютерні системи спеціального призначення;
– вебтехнології.

Стаття надійшла до редакції 20.03.2020.