УДК 004.4
DOI https://doi.org/10.32782/IT/2022-1-7

Андрій МОРОЗОВ
кандидат технічних наук, доцент, проректор з науково-педагогічної роботи, Державний університет «Житомирська політехніка», вул. Чуднівська, 103, Житомир, Україна, 10005, morozov@ztu.edu.ua
***ORCID:** 0000-0003-3167-0683*
***Scopus Author ID:** 55912634100*

Тетяна ВАКАЛЮК
доктор педагогічних наук, професор, професор кафедри інженерії програмного забезпечення, Державний університет «Житомирська політехніка», вул. Чуднівська, 103, Житомир, Україна, 10005, tetianavakaliuk@gmail.com
***ORCID:** 0000-0001-6825-4697*
***Scopus Author ID:** 57211133927*

Юрій КУБРАК
кандидат технічних наук, доцент, доцент кафедри інженерії програмного забезпечення, Державний університет «Житомирська політехніка», вул. Чуднівська, 103, Житомир, Україна, 10005, kubrak79@ukr.net
***ORCID:** 0000-0002-1122-7580*
***Scopus Author ID:** 57215318599*

Денис ЗОСІМОВИЧ
здобувач, Державний університет «Житомирська політехніка», вул. Чуднівська, 103, Житомир, Україна, 10005, unsinedz@gmail.com
***ORCID:** 0000-0002-1533-3882*

Бібліографічний опис статті: Морозов, А., Вакалюк, Т., Кубрак, Ю., Зосімович, Д. (2022). Аналіз факторів впливу на архітектури програмних систем. *Information Technology: Computer Science, Software Engineering and Cyber Security,* 1, 44–52, doi: https://doi.org/10.32782/IT/2022-1-7

АНАЛІЗ ФАКТОРІВ ВПЛИВУ НА АРХІТЕКТУРИ ПРОГРАМНИХ СИСТЕМ

У статті авторами досліджено існуючі підходи та рекомендації для побудови архітектур програмного забезпечення. Створення нового програмного продукту це завжди ризик. Обрання правильної архітектури є суттєвим кроком до успіху. Проаналізовано процес розробки, побудови та змін в архітектурі існуючого програмного комплексу. Виявлено зовнішні чинники, які впливали на процес розробки. Ними виявилися наступні: постійні зміни функціональних і нефункціональних вимог до розроблюваної системи; архітектурні тренди, які існували на момент проектування первинної архітектури; зміни фінансових умов розгортання та подальшої підтримки; зміна суб'єкта відповідальності за подальшу розробку системи. Було проведено аналіз деяких існуючих на сьогоднішній день досліджень у сфері програмних архітектур. З цього можемо зробити висновок про те, що дана сфера є досить широкою, а також є актуальним предметом для проведення наукових досліджень. Деякі дослідження вивчають економічні аспекти та корпоративні архітектури, причини архітектурних змін на пізніх фазах розробки або способи оцінки їх впливу на системну архітектуру. Кожен раз протягом усього процесу розробки програмного забезпечення існує низка факторів, які впливають на неї і, зокрема, призводять до прийняття певних архітектурних рішень. Розділимо умовно дані чинники на дві категорії: внутрішні та зовнішні. Провівши аналіз специфіки реалізації архітектур на основі конкретного прикладу довготривалої розробки, визначимо, що конкретна реалізація кожного з виду архітектур є досить масштабним та кропітким процесом, який вимагає ретельного планування та попереднього аналізу. Архітектурні помилки та фактори, що не були взяті до уваги або були недооцінені, можуть призвести до нівелювання попереднього прогресу або значної переробки існуючої системи. Це призводить до значного збільшення загальної вартості розроблюваної системи.

***Ключові слова:** фактори впливу, програмні системи, архітектура.*

Andrii MOROZOV
PhD in Engineering, Associate Professor, Vice Rector in Scientific and Pedagogical Work, Zhytomyr Polytechnic State University, 103, Chudnivska str., Zhytomyr, 10005, Ukraine, morozov@ztu.edu.ua
ORCID: *0000-0003-3167-0683*
Scopus Author ID: *55912634100*

Tetiana VAKALIUK
Doctor of Pedagogical Sciences, Professor, Professor at the Department of Software Engineering, Zhytomyr Polytechnic State University, 103, Chudnivska str., Zhytomyr, 10005, Ukraine, tetianavakaliuk@gmail.com
ORCID: *0000-0001-6825-4697*
Scopus Author ID: *57211133927*

Yurii KUBRAK
PhD in Engineering, Associate Professor, Associate Professor at the Department of Software Engineering, Zhytomyr Polytechnic State University, 103, Chudnivska str., Zhytomyr, 10005, Ukraine, kubrak79@ukr.net
ORCID: *0000-0002-1122-7580*
Scopus Author ID: *57215318599*

Denys ZOSIMOVYCH
Zhytomyr Polytechnic State University, 103, Chudnivska str., Zhytomyr, 10005, Ukraine, unsinedz@gmail.com
ORCID: *0000-0002-1533-3882*

To cite this article: Morozov, A., Vakaliuk, T., Kubrak, Yu., Zosimovych, D. (2022). Analysis of factors influencing the architecture of software systems. *Information Technology: Computer Science, Software Engineering and Cyber Security,* 1, 44–52, doi: https://doi.org/10.32782/IT/2022-1-7

ANALYSIS OF FACTORS INFLUENCING THE ARCHITECTURE OF SOFTWARE SYSTEMS

In the article, the authors explored existing approaches and recommendations for building software architectures. Creating a new software product is always a risk. Choosing the right architecture is an essential step to success. The process of development, construction, and changes in the architecture of the existing software package is analyzed. Identified external factors influencing the development process. They turned out to be the following: constant changes in the functional and non-functional requirements for the system being developed; architectural trends that existed at the time of designing the primary architecture; changes in financial conditions for deployment and further support; changing the subject of responsibility for the further development of the system. An analysis was made of some of the current research in the field of software architectures. From this, we can conclude that this area is quite wide, and is also a topical subject for scientific research. Some studies examine economics and enterprise architectures, the reasons for architectural changes in late phases of development, or how to evaluate their impact on system architecture. Each time throughout the entire software development process, some factors influence it and, in particular, lead to the adoption of certain architectural decisions. We conditionally divide these reasons into two categories: internal and external. After analyzing the specifics of the implementation of architectures based on a specific example of long-term development, we determine that the specific implementation of each type of architecture is a fairly large-scale and painstaking process that requires careful planning and preliminary analysis. Architectural errors and factors that were not taken into account or underestimated can lead to the leveling of previous progress or significant reworking of the existing system. This leads to a significant increase in the total cost of the system being developed.

Key words: *action factors, software systems, architecture.*

Актуальність проблеми. Створення нового програмного продукту це завжди ризик. Обрання правильної архітектури є суттєвим кроком до успіху (System Architecture). Найбільш розповсюдженими на сьогоднішній день є наступні архітектури: мікро-сервісна архітектура (МСА); сервісно-орієнтована архітектура (СОА); безсерверна архітектура (БА); монолітна архітектура (МА).

Кожен раз протягом усього процесу розробки програмного забезпечення існує низка факторів, які впливають на неї і, зокрема, призводять до прийняття певних архітектурних рішень. Розділимо умовно дані чинники на дві категорії: внутрішні та зовнішні.

Під внутрішніми факторами впливу ми вважаємо ті, що виникли у команди під час процесу розробки. Прикладом таких можуть слугувати наступні:

– рівень професіоналізму окремих її членів, від якого залежить широта аналізу контексту проблем протягом їх вирішення;

– домовленості всередині команди, тобто набір існуючих або власних практик щодо розробки ПЗ;

– емоційний стан або інші персональні аспекти, що так чи інакше впливають на якість реалізованого розробником функціоналу.

Зовнішніми факторами будемо вважати ті, чия природа виникнення знаходиться поза межами команди розробки. Прикладом можуть слугувати наступні: процеси, впроваджені менеджментом для вирішення типових задач розробки програмного забезпечення; фінансова ситуація компанії; соціально-економічні фактори, що можуть впливати на компанію або напряму на команду розробки.

Аналіз останніх досліджень і публікацій. Сучасне ІТ – це перш за все про бізнес. Саме він зацікавлений, як ніхто інший, в мінімізації витрат на розробку програмних продуктів. Це спричинює попит на персонал, кваліфікований для проектування архітектур корпоративних систем. Дуже часто ці люди діляться власним досвідом на конференціях, у статтях, а також проводять дослідження архітектур.

У своїй праці «Архітектура програмних систем» Оуен Вудс (англ. Eoin Woods) разом з Ніком Розанскі досліджували шляхи моделювання архітектур, які відображають та тримають в балансі потреби стейкхолдерів програмних продуктів. Були розглянуті наступні проблеми (Nick Rozanski, Eóin Woods, 2011):

– істотні моменти моделювання систем з точки зору архітектури;

– наслідки ігнорування аспектів швидкодії, гнучкості та розміщення;

– сценарії та шляхи створення та перевірки архітектур;

– підходи до поєднання гнучких методологій розробки та корпоративних архітектур.

Спираючись на зазначене дослідження, Оуен Вудс описав поширені архітектурні помилки та рекомендації щодо їх уникнення, а саме (Eoin Woods, 2008):

– нефункціональні помилки є більш складними;

– частий фокус на більш локальні проблеми, приділяючи менше часу аналізу ширшого контексту;

– правильний системний дизайн дозволяє легко нашаровувати новий функціонал;

– оцінка важливості функціоналу шляхом виміру кількості прибутки з точки зору окупності інвестицій (ROI, return of investment);

– системи та їх архітектури будуються для стейкхолдерів;

– необхідно розділяти «позитивних» та «негативних» стейкхолдерів.

Виходячи з цього, можна зробити висновок про те, що дане дослідження проводилося на предметі корпоративних архітектур, тобто акцент було зроблено саме на аналізі економічних аспектів та бізнесу.

Розглянемо роботу Байрона Дж. Вільямса та Джефрі К. Карвера «Характеристика змін програмних архітектур». Дослідження стосувалося причин архітектурних змін, їх класифікації та обсягу їх впливу на компоненти системи. Також, було частково проаналізовано проблематику підтримки програмних продуктів, які мають застарілу або невідповідну вимогам архітектуру (Byron J. Williams, Jeffrey C. Carver, 2010).

Упор у дослідженні було зроблено саме на пізніх архітектурних змінах, тобто на тих, що відбуваються в останніх фазах розробки програмних систем. У результаті виявлено та охарактеризовано типи даних змін, а також було розроблено фреймворк для аналізу та кращого розуміння їх причин.

У науковій роботі Дзіяндзюна Дзяо та Хондзі Яня «Аналіз впливу змін для підтримки архітектурної еволюції» було запропоновано підхід до оцінки впливу архітектурних змін на систему. Для цього було представлено спосіб відокремлення та розділення архітектурних моделей. Однією з головних переваг даного підходу є оцінка характеру змін в архітектурі системи на основі аналізу її формальних архітектурних специфікацій, що дозволяє повністю автоматизувати даний процес (Jianjun Zhao, Hongji Yang, Baowen Xu, 2002).

Беручи до уваги предмет на результати зазначеного дослідження, можна визначити, що воно більше стосувалося технічних аспектів системних архітектур.

Було проведено аналіз деяких існуючих на сьогоднішній день досліджень у сфері програмних архітектур. З цього можемо зробити висновок про те, що дана сфера є досить широкою, а також є актуальним предметом для проведення наукових досліджень. Деякі дослідження вивчають економічні аспекти та корпоративні архітектури (Nick Rozanski, Eóin Woods, Eoin Woods), причини архітектурних змін на пізніх фазах розробки (Byron J. Williams, Jeffrey C. Carver, 2010) або способи оцінки їх впливу на системну архітектуру (Jianjun Zhao, Hongji Yang, Baowen Xu, 2002).

Мета дослідження. Здійснити аналіз факторів впливу на архітектури програмних систем.

Виклад основного матеріалу дослідження. Розпочнемо аналіз реалізації деяких з них на прикладі системи розкладу для Державного університету «Житомирська політехніка», яка розроблювалася протягом 4 років (активна розробка припадає сумарно на 16-18 місяців).

Під час першої фази розробки вимоги до системи були досить простими, а саме: відображення актуального розкладу бази університету в мобільному додатку з мінімальними або відсутніми затратами на розміщення і підтримку інфраструктури. Для простоти розробки система мала монолітну архітектуру (MA). Схематично архітектуру зображено на рис. 1.

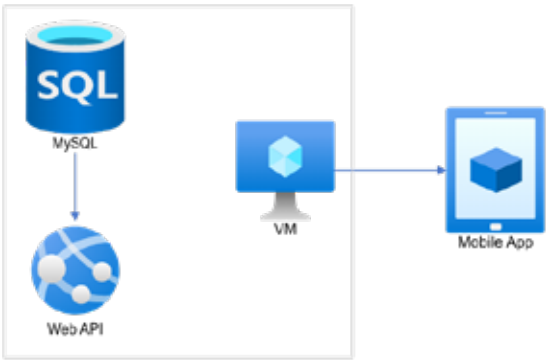

Рис. 1. Архітектурна схема моноліту

Монолітна архітектура – традиційна уніфікована модель побудови програмного забезпечення. В даному контексті монолітність означає скомпонованість функціоналу в єдиному місці, масивність, неможливість бути зміненим (Monolithic architecture definition).

На віртуальній машині (VM) було розміщено копію бази університету (MySQL) та серверний додаток (Web API). Мобільний додаток (Mobile App) виконує HTTP запити для отримання розкладу у форматі JSON та подальшого його відображення користувачеві. Мобільний додаток було опубліковано в режимі обмеженого тестування на платформі Google Play. Вихідний код системи наприкінці фази 1 наявний за посиланням (GitHub: ZSTU-App).

Для розробки системи було використано наступні технології: платформа Node.js та фреймворк Express; система управління базами даних MySQL; інфраструктура віртуальної машини – NginX, certbot; мобільний додаток – Flutter, SQLite.

З часом виникла необхідність розробки функціоналу для управління розкладом в базі. Вважатимемо її фазою 2. Вимоги до веб-додатка були наступні:

– наявність користувацького інтерфейсу панелі управління;

– зміни в інтерфейсі повинні виконувати зміни в базі даних;

– результатом змін має бути відображення оновленого розкладу на мобільному додатку.

На той момент необхідно було дати відповідь на фундаментальні запитання:

– чи продовжувати розробку в рамках моноліту?

– які технології використати для реалізації фази 2?

– як реалізувати COA в контексті моноліту з мінімальними затратами часу?

У результаті було розроблено окремий додаток з використанням технологій ASP.NET Core, HTML, CSS та JavaScript. Через брак досвіду з прийняття архітектурних рішень на той час останнє питання було зовсім не тривіальним. Виконувати міграцію системи з віртуальної машини у хмарне середовище Azure Cloud або Amazon Web Services було очевидно дорого, тим паче машина вже знаходилася в інфраструктурі хмарного стартапу Scaleway. Було прийнято рішення розмістити додаток панелі адміністрування на машині поряд з монолітом.

Отже, на момент виконання фази 2 архітектура додатку виглядала наступним чином (рис. 2).

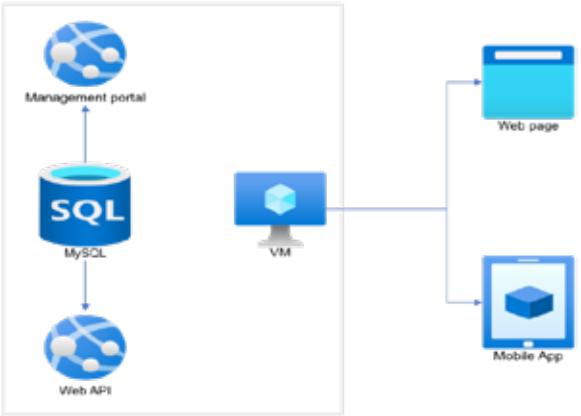

Рис. 2. Архітектурна схема системи (фаза 2)

COA визначає підходи для того, щоб зробити програмні компоненти багаторазовими у використанні та інтероперабельними шляхом впровадження сервісних інтерфейсів. Сервіси використовують спільні стандарти інтерфейсів та архітектурні шаблони так, що їх можна легко та швидко інтегрувати в нові додатки (What is SOA).

Додаток для управління розкладом та вебсервіс для його отримання можна вважати від-

носно незалежними, адже вони не комунікують між собою через відсутність такої потреби. Також, кожен з додатків був частиною однієї системи, адже мав доступ до одного джерела інформації – бази даних MySQL. Обидва функціональних компоненти системи мали власні програмні інтерфейси. Отже, можна сказати, що система почала набувати ознак СОА. Вихідний код додатку, що було створено у фазі 2, можна переглянути за посиланням (GitHub: ScheduleManager).

Однією з переваг СОА є можливість масштабування окремих сервісів. Даний вид масштабування можна назвати горизонтальним в контексті системного рішення, адже масштабується лише деякий окремий її компонент, а не вся система в цілому. Умови використання додатку було проаналізовано на момент проектування фази 1. У результаті було виявлено, що система не потребує спеціальної реалізації можливостей горизонтального масштабування в майбутньому. Проте, у випадку крайньої необхідності архітектура могла виглядати наступним чином (рис. 3).

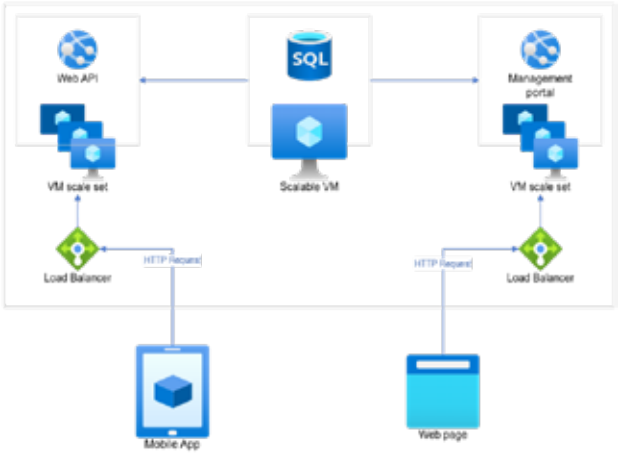

Рис. 3. Архітектура масштабованої системи

Для того, щоб оцінити обсяг потенційних робіт, розберемо архітектуру системи, яка надає можливість масштабування своїх компонентів.

На діаграмі (рис. 3) можна побачити появу нових сервісів, таких як розподілювач навантаження (load balancer), масштабований набір віртуальних машин (VM scale set) та окремо масштабована віртуальна машина (Scalable VM).

Розподілювач навантаження – сервіс, який відповідає за маршрутизацію вхідних запитів відповідно до навантаження кінцевих серверів. Використання даного компонента є необхідним для архітектур, в яких окремі компоненти існують у вигляді множини однотипних одиниць.

Масштабований набір віртуальних машин є прикладом такої множини. За допомогою збільшення кількості ідентичних сервісів вирішується проблема обмеженості апаратних потужностей однієї машини. Тепер окремий компонент можна розміщувати декілька разів, доки існують ресурси інфраструктури. Таким чином створюється кластер. Даний підхід називають горизонтальним масштабуванням (Horizontal vs Vertical scaling).

Під масштабуванням окремої віртуальної машини мається на увазі збільшення її обчислювальних потужностей, таких як центральний процесор, оперативна пам'ять, пропускна властивість мережі, оптимізація операційної системи тощо. Даний підхід називають вертикальним масштабуванням (Horizontal vs Vertical scaling).

Для надання системі масштабованості необхідно було б використати низку нових сервісів, налагодити їх взаємодію. Кошти та зусилля на розгортання та підтримку даної інфраструктури також були б значними. Найчастіше такі заходи проводяться для високонавантажених та розподілених систем, замовниками яких є великі корпорації, що можуть собі дозволити витратити десятки тисяч доларів на утримання подібних рішень.

Через те, що замовником виступав університет, а кінцевою аудиторією виступали студенти, викладачі, адміністрація та абітурієнти, кількість яких не можна назвати великою в контексті програмних рішень, розроблювана система не мала ознак високонавантаженої. А отже, дії щодо розробки засад для подальшого її масштабування були б зайвими.

Через деякий час після завершення фази 2 в системі було виявлено декілька суттєвих проблем, основною з яких була проблема її подальшої підтримки та розробки. Через те, що система складається з декількох компонентів, необхідно слідкувати та дороблювати кожний. Ресурси для виконання таких дій є досить обмеженими, а отже постало питання доцільності і можливості використання подібного програмного рішення. За час розробки фази 1 та 2 тренд безсерверної архітектури набував не лише більшої популярності в IT світі, але й з'явилася велика кількість відгуків щодо досвіду її впровадження.

Безсерверна архітектура – це підхід до побудови та розгортання хмарних додатків та сервісів без необхідності підтримки інфраструктури. В таких додатках виконання коду, його

підтримка та постачання є обов'язком обраної хмарної системи. Безсерверна архітектура виключає необхідність управління ресурсами, масштабуванням додатку, підтримки серверу, бази даних та сховища з боку розробника (System Architecture).

Існує дві основних концепції безсерверних архітектур: функція як сервіс (FaaS) та серверне забезпечення як сервіс (BaaS) (System Architecture).

FaaS (Function as a Service) – модель хмарного обчислення, в якій розробник може завантажувати окремі одиниці функціоналу у хмару та дозволити їм виконуватися незалежно (System Architecture).

BaaS (Backend as a Service) – модель хмарного обчислення, яка дозволяє розробнику «віддати на аутсорс» серверні аспекти (управління базою даних, хмарним сховищем, хостинг, автентифікація користувачів тощо) та писати і підтримувати лише клієнтську частину системи (GitHub: ZSTU-App).

Проаналізувавши існуючий ринок безсерверних сервісів, а також переваги, які вони надають, було прийнято рішення мігрувати існуючу систему управління розкладом на безсерверну архітектуру. Розробку в рамках даної міграції вважатимемо фазою 3. Постачальником послуг було обрано Google Firebase. Серед переваг було враховано наступні:

– простота розгортання та налаштування програмного рішення;

– вартість розміщення безсерверної інфраструктури, безкоштовні плани;

– простота підтримки та моніторингу за системою;

– простота інтеграції мобільних сервісів;

– простота інтеграції з існуючим мобільним рішенням.

Однією з основних особливостей безсерверних сервісів є умови тарифікації. В контексті BaaS вони найчастіше працюють за принципом Pay-as-you-go (плати у міру використання). Google Firebase є NoSQL базою даних та надає квоти на читання, запис та видалення рядків, а також на загальне місце збереження даних у віртуальному сховищі.

Використання Google Firebase змушувало вдатися до міграції існуючої MySQL бази даних (рис. 4). Під час першої спроби базу було мігровано зі збереженням існуючої третьої форми нормалізації (Database normalization basics). Загальний об'єм даних в колекціях Google Firebase складав близько 300 МБ, що задовольняло на той час безкоштовну квоту у 1 Гб.

Згодом під час первинної розробки та тестування виникла проблема – для перегляду розкладу одним користувачем додатку виконувалось одразу низка запитів на читання наступних сутностей: заняття, предмет, часовий проміжок заняття, викладач, аудиторія, підгрупа, група, факультет.

Маючи реальну базу розкладу, було виконано аналіз існуючих даних. Середня кількість занять для групи – 21; предметів для групи – 8 (наближено); унікальних аудиторій та викладачів майже стільки ж, скільки і пар, вважатимемо, що теж по 10; середня кількість підгруп для групи – 1,8; факультети та часові проміжки можна зберігати в кеші мобільного додатку, тобто не запитувати окремо. Маючи ці наближені дані, можна легко порахувати, що для 1 перегляду розкладу 1 групи виконуватиметься 21 + 8 + 10 + 10 + 1,8 = 50,8 запитів на читання рядків.

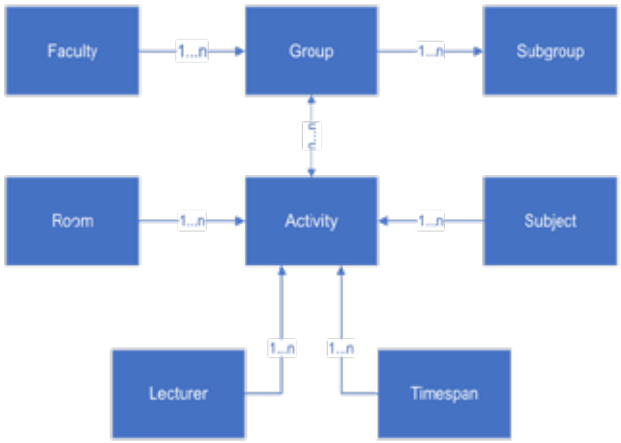

Рис. 4. Схема кількісних зв'язків сутностей існуючої бази

Беручи до уваги сценарій, коли студенти переглядають розклад перед кожною парою та щовечора, з'являється ще один множник переглядів на день, вважатимемо його 5. Якщо кожен 5 студент використовуватиме додаток, то маємо ще один наближений множник 800. В результаті – 50,8 * 5 * 800 ≈ 200 тис. запитів на читання рядка щодня, при безкоштовній квоті у 50 тис., що є недопустимим згідно з вимогами.

Повернемося до факту, що безкоштовна квота на віртуальне сховище була не використана майже на 70%. Наявність вільного місця дозволяє нам частково пожертвувати ним задля вирішення проблеми надмірної кількості запитів шляхом денормалізації бази даних способом шардованого розбиття даних.

Шардинг даних (англ. *data sharding*) – практика оптимізації систем управління базами

даних (СУБД) шляхом розділення рядків або колонок великих таблиць бази на декілька менших таблиці. В результаті отримані таблиці називаються шардами (або партиціями). Кожна з них або має однакому схему та унікальні набори рядків (горизонтальне шардування), або схему, атрибути якої є підмножиною оригінальної схеми (вертикальне шардування) (An overview of sharding and partitioning).

Упровадження зазначеного підходу к контексті нетабличної NoSQL бази Google Firebase мало власну інтерпретацію. Так, проаналізувавши варіанти використання (use-cases) додатку, єдиним типом, пов'язаним із зазначеною проблемою, було визначено пошук та перегляд розкладу для групи. Це дозволило дефініювати формат ключів партицій для множини занять – «сутність/ідентифікатор», наприклад «group/24».

Користуючись перевагами, які надають NoSQL рішення для збереження даних, а саме можливістю збереження множини даних в рамках одного запису, кожна партиція мала всі необхідні дані для задоволення варіанту використання «пошук та перегляд розкладу за групою», а отже 1 перегляд розкладу для 1 групи виконував лише 1 запит на читання рядка, зменшуючи оцінювану очікувану кількість щоденних запитів в реальному середовищі до 4 тис. При цьому, розмір сховища виріс до 625 МБ, що також попадало під безкоштовну квоту. Часткову умовну схему вихідної сутності, шарду, наведено у таблиці 1.

Для автоматизації процесу засобами Node.js та мовою JavaScript було розроблено скрипт, що виконував міграцію даних з вхідної бази MySQL у локальний образ бази Google Firestore. Вихідний образ завантажувався цілісно у хмарне середовище. Це дозволило легально «обійти» квоту на запис рядків до колекцій зазначеної вихідної бази в реальному середовищі. Вихідний код фази 3 можна знайти за посиланням [9].

Важливо зазначити, що проаналізована схема даних Google Firestore вважалася незмінною і використовувалася лише для читання, тобто при оновленні даних в існуючій MySQL базі необхідно було повторити процес міграції. Оновлення нової схеми вважаємо досить дорогою операцією через роздробленість даних та квоти на запис, що показує нам втрати в консистентності системи, а саме:

– складність операції запису через велику кількість партицій;

– відсутність єдиного «джерела правди»;

– зміни в одній партиції не впливають на інші;

– відсутність механізму транзакцій для виконання атомарних змін у декількох партиціях одночасно (обмеження використаної СУБД).

Таблиця 1

Часткова умовна схема шарду

timetable_items		
Поле	**Тип**	**Опис**
key	string	Ключ партиції
timetableId	string	Ідентифікатор пов'язаного запису налаштувань розкладу
items	timetable_group_item[]	Набір сутностей опису занять
timetable_group_item		
dayNumber	integer	День тижня
weekNumber	integer	Номер тижня
activity	Activity	Сутність заняття
Activity		
name	string	Назва предмету
room	string	Номер аудиторії
time	{start: string, end: string}	Проміжок часу
tutor	Tutor	Викладач
type	string	Лабораторна, лекція, практика
groups	Group[]	Присутні групи
Group		
id	integer	Ідентифікатор групи
facultyId	integer	Ідентифікатор факультету
year	string	Курс, освітній рівень
name	string	Назва групи
subgroups	Subgroup[]	Набір підгруп

Так емпіричним шляхом була перевірена CAP (англ.) теорема або теорема Брюера. Згідно з даною теоремою, розподілена система може мати лише 2 з 3 зазначених властивостей (What is the CAP theorem): консистентність (Consistency); доступність (Availability); толерантність до партицій (Partition tolerance).

У результаті фази 3 було проведено міграцію даних та написано автоматизований скрипт, який повністю забрав на себе функції веб-додатку, розробленого у фазі 1. Це призвело до того, що значну частину роботи фази 1 було втрачено через відсутність необхідності її подальшого використання. Також, наслідком зміни сховища даних маємо необхідність проведення змін у компонентах системи, що від нього залежать, а саме в мобільному додатку.

Отже, провівши аналіз специфіки реалізації архітектур на основі конкретного прикладу довготривалої розробки, визначимо, що конкретна реалізація кожного з виду архітектур є досить масштабним та кропітким процесом, який вимагає ретельного планування та попереднього аналізу. Архітектурні помилки та фактори, що не були взяті до уваги або були недооцінені, можуть призвести до нівелювання попереднього прогресу або значної переробки існуючої системи. Це призводить до значного збільшення загальної вартості розроблюваної системи.

Висновки і перспективи подальших досліджень. В результаті авторами досліджено існуючі підходи та рекомендації для побудови архітектур програмного забезпечення. Проаналізовано процес розробки, побудови та змін в архітектурі існуючого програмного комплексу. Виявлено зовнішні чинники, які впливали на процес розробки. Ними виявилися наступні: постійні зміни функціональних і нефункціональних вимог до розроблюваної системи; архітектурні тренди, які існували на момент проектування первинної архітектури; зміни фінансових умов розгортання та подальшої підтримки; зміна суб'єкта відповідальності за подальшу розробку системи. До перспектив подальших досліджень відносимо вдосконалення архітектури додатку на прикладі розкладу Державного університету «Житомирська політехніка».

ЛІТЕРАТУРА:

1. System Architecture. URL: https://rubygarage.org/blog/monolith-soa-microservices-serverless (дата звернення: 01.11.2021).

2. Monolithic architecture definition. URL: https://whatis.techtarget.com/definition/monolithic-architecture (дата звернення 16.11.2021).

3. GitHub: ZSTU-App. URL: https://github.com/unsinedz/ZSTU-App (дата звернення: 05.11.2021).

4. What is SOA. URL: https://www.ibm.com/cloud/learn/soa (дата звернення: 05.11.2021).

5. GitHub: ScheduleManager. URL: https://github.com/unsinedz/ScheduleManager (дата звернення: 05.11.2021).

6. Horizontal vs Vertical scaling. URL: https://touchstonesecurity.com/horizontal-vs-vertical-scaling-what-you-need-to-know (дата звернення: 05.11.2021).

7. Database normalization basics. URL: https://docs.microsoft.com/en-us/office/troubleshoot/access/database-normalization-description (дата звернення: 14.11.2021).

8. An overview of sharding and partitioning. URL: https://hazelcast.com/glossary/sharding (дата звернення 14.11.2021).

9. GitHub: Zhytomyr-polytechnic. URL: https://github.com/unsinedz/zhytomyr-polytechnic/tree/develop (дата звернення: 14.11.2021).

10. What is the CAP theorem. URL: https://www.ibm.com/cloud/learn/cap-theorem (дата звернення: 05.11.2021).

11. Nick Rozanski, Eóin Woods. Software Systems Architecture: Working With Stakeholders Using Viewpoints and Perspectives. Addison-Wesley, 2011, 704 p. URL: https://books.google.com/books?id=ka4QO9kXQFUC (дата звернення: 19.11.2021).

12. Eoin Woods. Top 10 Architecture Mistakes, 2008. 42 p. URL: http://jaoo.dk/dl/jaoo-aarhus-2008/slides/EoinWoodsTop10Mistakes.pdf (дата звернення: 19.11.2021).

13. Byron J. Williams, Jeffrey C. Carver. Characterizing software architecture changes: a systematic review. *Information and Software Technology*.Volume 52, Issue 1, January 2010, Pages 31-51. https://doi.org/10.1016/j.infsof.2009.07.002

14. Jianjun Zhao, Hongji Yang, Baowen Xu. Change impact analysis to support architectural evolution. *Journal of Software Maintenance and Evolution Research and Practice,* Vol. 14(5), 2002. Pp. 317-333. DOI: 10.1002/smr.258

REFERENCES:

1. System Architecture. Retrieved from: https://rubygarage.org/blog/monolith-soa-microservices-serverless.
2. Monolithic architecture definition. Retrieved from: https://whatis.techtarget.com/definition/monolithic-architecture.
3. GitHub: ZSTU-App. Retrieved from: https://github.com/unsinedz/ZSTU-App.
4. What is SOA. Retrieved from: https://www.ibm.com/cloud/learn/soa.
5. GitHub: ScheduleManager. Retrieved from: https://github.com/unsinedz/ScheduleManager.
6. Horizontal vs Vertical scaling. Retrieved from: https://touchstonesecurity.com/horizontal-vs-vertical-scaling-what-you-need-to-know.
7. Database normalization basics. Retrieved from: https://docs.microsoft.com/en-us/office/troubleshoot/access/database-normalization-description.
8. An overview of sharding and partitioning. Retrieved from: https://hazelcast.com/glossary/sharding.
9. GitHub: Zhytomyr-polytechnic. Retrieved from: https://github.com/unsinedz/zhytomyr-polytechnic/tree/develop.
10. What is the CAP theorem. Retrieved from: https://www.ibm.com/cloud/learn/cap-theorem.
11. Nick Rozanski, Eóin Woods. Software Systems Architecture: Working With Stakeholders Using Viewpoints and Perspectives. Addison-Wesley, 2011, 704 p. Retrieved from: https://books.google.com/books?id=ka4QO9kXQFUC.
12. Eoin Woods. Top 10 Architecture Mistakes, 2008. 42 p. Retrieved from: http://jaoo.dk/dl/jaoo-aarhus-2008/slides/EoinWoodsTop10Mistakes.pdf.
13. Byron J. Williams, Jeffrey C. Carver. Characterizing software architecture changes: a systematic review. *Information and Software Technology*. Volume 52, Issue 1, January 2010, Pages 31-51. https://doi.org/10.1016/j.infsof.2009.07.002
14. Jianjun Zhao, Hongji Yang, Liming Xiang, Baowen Xu. Change impact analysis to support architectural evolution. *Journal of Software Maintenance and Evolution Research and Practice,,* Vol. 14(5), 2002. Pp. 317-333. DOI: 10.1002/smr.258