УДК 004.41:004.05:004.65

# ACCESS REFACTORING CATEGORY

**V. Struzik, S. Hrybkov, V. Chobanu**
*National University of Food Technologies*

**Key words:**
*Microservices*
*Service-oriented architecture*
*Refactoring*
*Access refactoring category*
*Monolith*
*Databases*

**Article history:**
Received 09.03.2020
Received in revised form 23.03.2020
Accepted 06.04.2020

**Corresponding author:**
V. Struzik
**E-mail:**
struzik.vladislav@gmail.com

**ABSTRACT**

The advantages and disadvantages of monolithic and microservice architecture patterns are investigated in the article, as well as the conditions of the feasibility of their usage in the development of corporate information systems. These architectural patterns are the most common. Different options for interacting with databases are also considered separately.

The software has frequent changes due to the high dynamics of development of the modern world on operate phase. The software must constantly meet business requirements. This leads to increase in complexity of the code and, as a consequence, support for the software as a whole. Software developers apply for one of the techniques of extreme programming, refactoring, to reduce "technical debt" and improve the operating process. The authors of the article focus on the research of database refactoring.

Six existing categories of database refactoring are described in the article. The development of a new category of database refactoring, namely the category of access refactoring, is an important part of this article. Operations of the category are described and recommendations for their use are provided. The newly created category accumulates changes in the database management system associated with access to the data, such as operations associated with changes to the location of the database object, operations associated with changes to the attributes of user authentication, operations associated with changes of user authorization rights. Authors described the feasibility of using access refactoring category for migration between architectural patterns and when security policy events occur.

**DOI:** 10.24263/2225-2924-2020-26-2-4

# КАТЕГОРІЯ РЕФАКТОРИНГ ДОСТУПУ

**В. А. Струзік, С. В. Грибков, В. В. Чобану**
*Національний університет харчових технологій*

*У статті досліджено переваги та недоліки монолітного та мікросервісного шаблонів архітектури, а також умови доцільності їх використання в розробці корпоративних інформаційних систем. Ці шаблони архітектури є найбільш розповсюдженими. Також окремо розглянуто різні варіанти взаємодії сервісів з базами даних.*

*Під час експлуатації програмний продукт піддається частим змінам через високу динаміку розвитку сучасного світу, а сам продукт має постійно відповідати бізнес-вимогам. Це призводить до зростання складності програмного коду і, як наслідок, підтримки програмного забезпечення загалом. Задля зменшення «технічного боргу» та поліпшення процесу експлуатації розробники програмного забезпечення звертаються до одного з прийомів методології екстремального програмування — рефакторингу. Автори статті наголошують саме на дослідженні рефакторингу баз даних.*

*Наведено шість існуючих категорій рефакторингу баз даних, а також подано їхній загальний опис. Важливою частиною є розробка нової категорії рефакторингу баз даних, зокрема категорії рефакторинг доступу. Описано операції цієї категорії та надано рекомендації щодо їх використання. Новостворена категорія акумулює в собі зміни в системі управління базою даних, що пов'язані з доступом до даних, тобто операції, пов'язані зі змінами розташування об'єкта бази даних, атрибутів аутентифікації користувача, авторизаційних прав користувача. Підкреслено доцільність використання певних операцій рефакторингу доступу при переході між шаблонами архітектури та при виникненні подій, що пов'язані з політикою безпеки.*

***Ключові слова:*** *мікросервіси, сервіс-орієнтована архітектура, рефакторинг, рефакторинг доступу, моноліт, бази даних.*

**Постановка проблеми.** На сьогодні створено шість категорій рефакторингу баз даних, проте жодна з них не описує зміни в системі управління базою даних, що пов'язані з доступом до цих даних. Доцільно доповнити класифікацію категорій рефакторингу баз даних додавши нову категорію — рефакторинг доступу, та описати операції, які вона включає.

**Аналіз останніх досліджень і публікацій.** Автори фундаментальної праці [10] описали шість категорій рефакторингу баз даних, серед яких: категорія рефакторинг структури; категорія рефакторинг якості даних; категорія рефакторинг посилальної цілісності; категорія рефакторинг архітектури; категорія рефакторинг методів; зміни, що не входять до операцій рефакторингу.

Проте жодна з описаних категорій не передбачає операції, пов'язані з доступом до даних.

**Мета статті:** створення категорії рефакторинг доступу, що акумулює в собі зміни в системі управління базою даних, пов'язані з доступом до цих даних.

**Викладення основних результатів дослідження.** *Корпоративні системи та їх особливості.* Усі сучасні компанії інтенсивно розвиваються, що вимагає постійної систематизації інформації й автоматизації всіх бізнес-процесів. На ранніх етапах автоматизацію бізнес-процесів компанії можливо забезпечити за рахунок використання стандартних офісних додатків, проте з часом обсяги інформації зростають і досягають таких об'ємів, що виникає потреба у створенні та використанні корпоративних інформаційних систем.

Такі системи об'єднують технічні й програмні засоби для роботи зі стратегічною й оперативною інформацією компанії, а також реалізують ідеї та методи автоматизації всіх бізнес-функцій управління. Корпоративні системи мають складну ієрархічну структуру, функціонують у розподіленій обчислювальній мережі та розраховані на використання багатьма користувачами.

Корпоративні системи передбачають підвищені вимоги до надійності функціонування та збереження цілісності даних, що забезпечуються за рахунок механізмів підтримки посилальної цілісності й транзакційної властивості сучасних систем управління базами даних.

Основною характеристикою корпоративної інформаційної системи є можливість збільшення її функціональності — гнучко та оперативно змінювати алгоритми функціонування. При цьому обов'язковою вимогою до таких систем є здатність до інтеграції з іншими програмними продуктами.

Перевагами використання корпоративної інформаційної системи як невід'ємного інструменту керування підприємством є підвищення якості планування за рахунок оперативного та швидкого доступу й обробки даних; зменшення часу на виявлення проблем і можливостей; використання інструментів стратегічного планування, що дає змогу якісно покращити прийняття оперативних і стратегічних рішень; інтеграція з різноманітним програмним забезпеченням для формування єдиного програмного комплексу; покращення координації діяльності співробітників і підрозділів, а також забезпечення їх необхідною й актуальною інформацією; контроль за виконанням поставлених завдань.

*Загальні характеристики розподілених систем.* На сьогодні майже будь-яка велика інформаційна система є розподіленою, адже обробка інформації виконується на декількох комп'ютерах, а не зосереджена на одній обчислювальній машині. Донедавна більшість систем були централізованими. Такі системи функціонували в рамках однієї обчислювальної машини (мейнфрейму) з підключеними до неї терміналами. Термінали не підтримували інших функцій, крім введення та виведення інформації, а всі основні операції виконувались централізовано на сервері. Виділяють три основні характеристики розподілених систем: прозорість, відкритість і масштабованість. Авторами [1] виділено ще три характеристики: конкурентність, паралельність, відмовостійкість. Але вони є різними типами прозорості розподіленої системи, адже конкурентність і паралельність передбачає прозорість паралельного доступу та виконання, а відмовостійкість — прозорість відмов.

Насамперед варто зазначити, що кожен клієнтський додаток встановлений автономно на певному комп'ютері, проте з точки зору користувача це єдина система. Важливе завдання розподілених систем полягає в тому, щоб приховати той факт, що процеси і ресурси фізично розподілені між комп'ютерами.

Розподілені системи, які представляються користувачам і програмам у вигляді єдиної інформаційної системи, називають прозорими. Існують різні типи прозорості [2]:

- прозорість доступу надає можливість приховати різницю в представленні даних і в способах доступу користувачів до ресурсів;
- прозорість розташування надає можливість приховати від користувача, де саме фізично розташований у системі необхідний йому ресурс;
- прозорість реплікації надає можливість приховати той факт, що існує кілька копій ресурсу;
- прозорість паралельного доступу надає можливість використовувати одночасно один ресурс декільком користувачам без надання їм інформації про його використання іншим користувачем;
- прозорість відмов надає можливість ніколи не повідомляти користувача про відновлення роботи тієї частини системи, яку він не використовує чи не має доступу;
- прозорість збереження маскує реальне (жорсткий диск) або віртуальне (оперативна пам'ять) збереження ресурсів.

Інша важлива характеристика розподілених систем — це відкритість. Відкрита розподілена система — це система з інтерфейсом підключення, звернення до якого має стандартний синтаксис і семантику. Наявність інтерфейсу надає можливість спільної роботи довільного процесу, що має потребу в ньому, з іншим довільним процесом, що його імплементує. Інтерфейс підключення також дає змогу двом незалежним групам розробників створити абсолютно різні реалізації цього інтерфейсу для окремих розподілених систем, тобто надавати послуги за моделлю B2B (англ. business to business — бізнес для бізнесу).

Наступна важлива характеристика відкритих розподілених систем — це гнучкість. Під гнучкістю розуміється легкість конфігурування системи, що складається з різних компонентів. У розподілених системах є можливість додавання до системи нових компонентів або заміна існуючих, можливо від різних виробників, без додаткових труднощів. При цьому компоненти, з якими не проводилося жодних дій, залишаються незмінними.

Значною перевагою розподілених систем є їхня здатність до масштабування, яка вимірюється за трьома різними показниками. По-перше, система може бути масштабована стосовно її розміру, що забезпечує легкість підключення до неї додаткових ресурсів. По-друге, система може бути масштабована географічно, тобто користувачі і ресурси можуть бути розташовані географічно в різних приміщеннях, містах, країнах. По-третє, система може бути масштабована в адміністративному сенсі, тобто бути простою в управлінні при роботі в безлічі адміністративно незалежних організацій. Завдяки цій характерній рисі у випадку, коли виникає необхідність у збільшенні обчислювальних ресурсів, є можливість їх нарощувати, не змінюючи програмну складову системи. Проте на практиці нарощування обмежене мережею, що об'єднує окремі комп'ютери.

Незважаючи на всі переваги розподілених систем порівняно з централізованими системами, вони мають і ряд істотних недоліків. Через те, що розподілені системи складніші за централізовані, набагато важче оцінити їхній

ресурс у цілому, а також тестувати такі системи. Це пов'язано з тим, що продуктивність системи залежить не від швидкості роботи одного процесора, а від пропускної здатності мережі та швидкості роботи різних процесорів.

Серед багатьох проблем основними є безпека, складність адміністрування, обмеження масштабування.

Складність організації захисту полягає в тому, що доступ до системи можна отримати з багатьох різних вузлів мережі, а передані повідомлення можуть бути перехоплені. Тобто дані, що передаються між компонентами, слід захистити як від спотворення, так і від перегляду сторонніми особами. Різновид типів атак, які можуть бути використані до розподілених систем, кількісно більше порівняно з централізованими системами. Додаткова складність при створенні полягає в тому, що алгоритми аутентифікації й авторизації необхідно реалізувати в усіх компонентах розподіленої системи.

Проблеми адміністрування системи включають проблеми балансування навантаження на вузли системи, відновлення даних у разі виникнення відмов. Через розподіленість розміщення ресурсів виникає необхідність створення гнучких засобів адміністрування.

У розподілених системах адміністрування повинно відбуватися постійно, а у зв'язку з їхніми особливостями виникають такі основні проблеми:

- балансування навантаження на вузли системи;
- відновлення даних у разі виникнення помилки;
- моніторинг вузлів системи;
- оновлення програмного забезпечення на вузлах системи в автоматичному режимі.

Виділимо основні проблеми маштабованості розподілених систем. Проблема кількісного збільшення вузлів системи, що не завжди можливе, у зв'язку з обмеженістю служб та алгоритмів, тобто виникнення стану гонитви (англ. race condition, race hazard) [3]. Проблема нестачі обчислювальних ресурсів сервера, який займається агрегуванням даних, що зібрані з вузлів системи в загальне глобальне представлення. Проблема пропускної спроможності комп'ютерної мережі при географічному масштабуванні. Якщо вузли розподіленої системи можуть знаходитись у географічно віддалених точках, що призводить до можливого зменшення загальної надійності та продуктивності розподіленої системи при низькій швидкості передачі даних.

*Мікросервісні та монолітні системи.* Монолітні системи являють собою цілісний продукт, в якому інтерфейс користувача та виконання прикладних задач обробки даних поєднується в єдиний програмний модуль («код»), що забезпечує виконання певної задачі з чітким виконанням кожного кроку для досягнення потрібного результату. Така система є автономною і незалежною від інших програмних додатків, працює в рамках однієї обчислювальної системи, незалежно від внутрішньої модульності або архітектурних шарів.

Основними перевагами використання монолітного шаблону архітектури є:

- простота розробки — метою сучасних засобів розробки, в тому числі інтегрованих середовищ розробки, є підтримка розробки монолітних систем;

- простота розгортання — для цього необхідно завантажити систему у відповідне середовище виконання, без потреби розгортання додаткових підсистем, які необхідні для її функціонування;

- просте масштабування — дає змогу розгорнути декілька копій програми та розмістити їх за балансувальником навантаження.

Незважаючи на ряд переваг, такі системи мають певні недоліки, що виникають, коли система розширюється та змінюється команда розробників:

- значна за розміром монолітна система є складною для підтримки та розширення, особливо для новачків у команді розробників, що уповільнює її подальше розширення та з часом призводить до погіршення програмного коду;

- перевантаження засобів розробки, що призводить до зменшення продуктивності розробників, адже чим більша кодова база системи, тим повільніше працює інтегроване середовище розробника;

- перевантаження середовища виконання, адже чим більша за розміром система, тим більше часу витрачається на її тестування, розгортання та завантаження;

- ускладненість безперервного розгортання — особливо часто ця проблема виникає саме у розробників інтерфейсу користувача, адже, щоб оновити один компонент, необхідно заново розгорнути всю систему;

- проблема масштабування — масштабування можливе тільки горизонтально, крім того, різні модулі системи мають різні вимоги до ресурсів, а з монолітним шаблоном архітектури ми не можемо масштабувати кожен компонент окремо;

- довгострокова відданість технологічному стеку — монолітна архітектура змушує прив'язатися до технологічного стеку, як наслідок, виникає обмеженість використання компонентів, що базуються на інших технологіях. Якщо ж платформа, що використовується, застаріє, то поступова міграція на нову платформу неможлива, а створення нової системи з нуля призводить до великих ризиків.

Мікросервісний шаблон архітектури є сучасним уявленням сервіс-орієнтованої архітектури, що здійснює публікацію сервісами своїх інтерфейсів, даючи змогу іншим сервісам звертатися до них за допомогою стандартних мережевих протоколів, які не залежать від мов програмування і платформ.

Мікросервіс — це незалежний, автономний ресурс, спроектований як окремий сервіс у рамках інформаційної системи, що взаємодіє з іншими мікросервісами через стандартні способи зв'язку, такі як гіпертекстовий транспортний протокол, черги повідомлень тощо. Унікальність мікросервісів обумовлена тим, що кожен розробляється, тестується, розгортається і масштабується незалежно від інших. Ідея використання мікросервісів заснована на кращих принципах розробки програмного забезпечення, в тому числі таких, як слабка зв'язність (coupling), висока згуртованість (cohesion), висока масштабованість.

Розробка інформаційної системи з використанням мікросервісного шаблону архітектури має ряд переваг:

- забезпечує безперервне розгортання системи — через постійне оновлення інформаційної системи, за рахунок CI/CD (англ. continuous integration and continuous delivery), адже кожна команда відповідальна за певний елемент, та

може, незалежно від інших команд, розробляти, тестувати, розгортати і масштабувати;

- кожен мікросервіс відносно малий — розробнику легше зрозуміти та вносити зміни, швидше проводиться тестування, робота в інтегрованому середовищі розробки більш продуктивна за рахунок швидкодії, зменшуються втрати часу на запуск системи;

- ізоляція несправностей — порівняно з монолітною системою, у якій один некоректно працюючий компонент впливає на всю систему, якщо в одному сервісі мікросервісної системи стався, наприклад, витік пам'яті, це вплине тільки на конкретний сервіс, інші сервіси будуть продовжувати обробляти запити;

- усуває довгострокову відданість технологічному стеку — при розробці нового сервісу можливо обирати технології, які доцільно використати, а не обмежуватися раніше обраними, також при внесенні серйозних змін в існуючий сервіс є можливість переписати його окремо, використовуючи новий технологічний стек, та не впливати на будь-які інші компоненти системи.

Проте мікросервісний шаблон архітектури має певні недоліки:

- складність розгортання — у виробничому процесі збільшується час на розгортання й управління системою за рахунок багатьох різних служб;

- збільшене споживання ресурсів, адже заміна N монолітних екземплярів системи на NxM мікросервісів призводить до збільшення кількості необхідних обчислювальних ресурсів, якщо кожна служба працює у своїй власній віртуальній машині;

- команди розробників повинні чітко розуміти, як реалізувати механізми міжсервісного зв'язку і впоратися з частковою відмовою, тому виникає потреба в інтеграційному тестуванні в рамках взаємодії між сервісами; як реалізувати запити, які охоплюють кілька служб; усі нюанси створення монолітних додатків.

*Процес переходу між мікросервісним і монолітним шаблоном архітектури.* Мікросервісний та монолітний шаблони архітектури достатньо тісно пов'язані. Найчастіше на початкових етапах розробки нового сервісу розробники програмного забезпечення надають перевагу саме монолітному шаблону архітектури. Такий вибір обумовлено перевагами цього шаблону архітектури, що описані вище: простота розробки, простота розгортання та просте масштабування. Крім того, при використанні монолітного шаблону архітектури є можливість дешевої розробки мінімально життєздатного продукту (англ. Minimum viable product — MVP). Такий шаблон архітектури зручний у використанні у більшості випадків лише на початку розробки та підтримки системи, оскільки під час експлуатації та розширення монолітного програмного продукту розробники яскраво бачать вузькі місця, що потребують масштабування. Важливо звернути увагу на те, що при проектуванні програмного продукту відповідно до монолітного шаблону архітектури використовують одну базу даних, в якій і зберігаються всі бізнес-важливі дані. Тож вузьким місцем найчастіше стають запити до бази даних. Для усунення проблем вузьких місць необхідно провести детальний аналіз, що надасть можливість усвідомлено приймати рішення про виконання оптимізації чи масштабування.

Масштабування баз даних виконується у двох вимірах: вертикально та горизонтально. Вертикальне масштабування полягає в збільшенні обчислювальних ресурсів, а горизонтальне — у створенні додаткового екземпляра серверу баз даних.

У загальному випадку горизонтальне масштабування баз даних передбачає три варіанти [4]:

- реплікація — синхронне або асинхронне копіювання елементів бази даних між декількома серверами;
- шардінг — розподілення даних між різними фізичними серверами за обраним фактором [5];
- партиціонування — розбиття таблиць, що містять велику кількість записів, за обраним критерієм, на множину фізичних таблиць, доступ до яких надається через логічну таблицю.

При виникненні проблем з обробкою запитів на читання застосовують реплікацію, а в разі проблем із записом або обробки великого об'єму — шардінг. Партиціонування застосовують тоді, коли при фіксації транзакції багато часу займає перебудова індексів. Загалом, перед проведенням горизонтального масштабування доцільно пересвідчитися, чи можливо провести реплікацію або шардування саме вузького місця, тобто виділити частину логічної схеми бази даних в окрему фізичну базу даних, що надасть можливість ізолювати проблему та зменшити вплив на швидкодію інформаційної системи загалом. За наявності такої можливості варто застосувати операцію виділення схеми бази даних, що відноситься до категорії рефакторинг доступу.

Якщо вузьким місцем є бізнес-логіка або взаємодія з окремою частиною бази даних інформаційної системи, то виникає необхідність виділення вузького місця в окремий сервіс, тобто відбувається перехід від монолітного до мікросервісного шаблону архітектури. Крім того, потреба у виділення окремого сервісу виникає за бізнес-потреби масштабування команди розробників.

Однією зі стратегій переходу до мікросервісного шаблону архітектури є декомпозиція за бізнес-складовою (decompose by business capability). Така стратегія передбачає, що модель бізнесу впливає на модульність та архітектуру інформаційної системи. Іншою стратегією реалізації мікросервісного шаблону архітектури є декомпозиція за субдоменом (decompose by subdomain), яка передбачає створення структури інформаційної системи відповідно до моделі предметної області. Для коректного виділення окремого субдомену доречно застосувати предметно-орієнтоване проектування, що полягає у створенні програмних абстракцій, які описують модель предметної області. Ця модель включає бізнес-логіку, що встановлює зв'язок між реальними умовами області застосування продукту і кодом. У стратегії декомпозиції за субдоменом домен розмежовується на субдомени, які відповідають різним частинам предметної області.

Загалом, сервіси доцільно створювати відповідними за складністю до складу команди розробників, щоб над ними було легко працювати та легко проводити тестування. Декомпозиція виконується таким чином, щоб більшість змін у

вимогах впливали лише на конкретний сервіс. Це обумовлено тим, що для змін, які впливають одразу на декілька сервісів, необхідно залучати відразу більше однієї команди, що, у свою чергу, збільшує час виконання завдання. Додатковими вимогами до реалізації сервісів відповідно до сервіс-орієнтованої архітектури є висока згуртованість (cohesion) та низька зв'язаність (coupling) [6; 7].

При переході до мікросервісного шаблону архітектури не варто обмежуватись однією стратегією переходу, для отримання оптимального результату допускається комбінування вищевказаних стратегій.

При використанні мікросервісного шаблону архітектури мають місце два варіанти взаємодії з базою даних: «база даних кожному сервісу» (database per service) та «спільна база даних» (shared database).

Головною особливістю підходу «база даних кожному сервісу» є те, що кожний сервіс виконує маніпуляції в рамках власної бази даних (рис. 1), а головним обмеженням є те, що жодний інший сервіс не може мати привілеїв доступу до бази даних іншого сервісу. Зв'язок або обмін даними може відбуватися лише за допомогою набору чітко визначених інтерфейсів програмування додатків (англ. API — application programming interface). Неправильне розділення моноліту на мікросервіси може призвести до високої зв'язаності між сервісами, оскільки виникає необхідність проводити обмін даними для реалізації їхньої логіки. Інший недолік цього патерну — це ймовірність виникнення проблем з реалізацією бізнес-транзакцій, що охоплюють декілька мікросервісів. Для кращої реалізації цього використовують шаблон сага (saga pattern) [8]. Зазвичай, підхід «база даних кожному сервісу» застосовують для ізоляції вузького (навантаженого) місця.

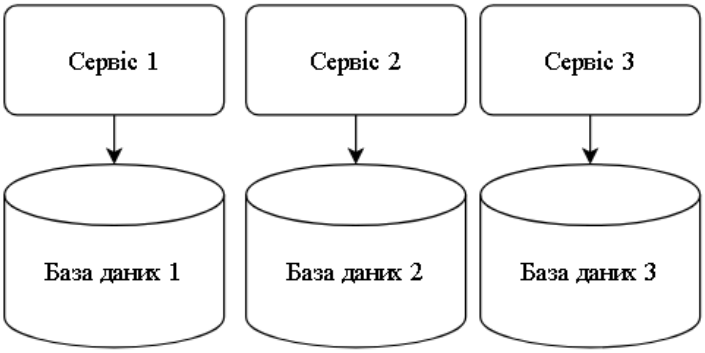


**Рис. 1. Підхід database per service**

Спільна база даних (shared database) — підхід, при якому різні мікросервіси використовують одну базу даних (рис. 2). Перевагою такого підходу є проста реалізація бізнес-транзакцій — розробники використовують транзакції ACID для забезпечення консистентності даних. Крім того, спільна база даних полегшує роботу. Важливою перевагою цього підходу є просте відновлення бази даних з бекапу порівняно з варіантом «база даних кожному сервісу», де виникає складність з відновленням консистентного стану після збою. У свою

чергу, недоліком є те, що робота за варіантом «спільна база даних» може бути повільнішою. Оскільки всі сервіси використовують одну базу даних, важливо координувати внесення змін до схеми бази даних з розробниками інших сервісів, що звертаються до тих самих таблиць бази даних. Таким чином виникає зв'язаність часу розробки (development time coupling) [9]. Крім того, уповільнити роботу може зв'язаність під час виконання (runtime coupling) [9]. Це ситуація, коли декільком мікросервісам необхідно отримати доступ до однієї таблиці бази даних. Тоді у випадку, якщо з таблицею бази даних вже працює певний мікросервіс, інший мікросервіс, що звертається до цієї таблиці пізніше, повинен чекати на завершення попереднього звернення. Також варто зазначити, що спільна база даних може не відповідати вимогам зберігання даних і доступу до всіх мікросервісів.

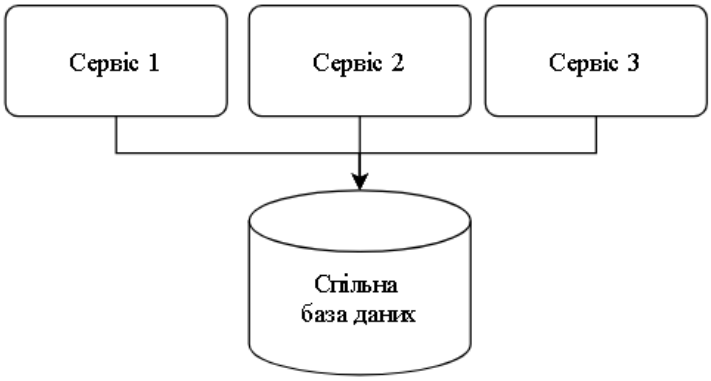


**Рис. 2. Підхід shared database**

При виділенні вузького місця в окремий сервіс з використанням підходу «база даних кожному сервісу» виникає потреба в таких операціях категорії рефакторингу доступу:

- звуження привілеїв доступу;
- виділення схеми бази даних.

Об'єднання сервісів, тобто перехід від мікросервісного шаблону архітектури до монолітного, найчастіше виникає при зміні бізнес-процесів, зокрема тоді, коли два і більше сервісів набувають великої зв'язаності, що призводить до надмірної складності їх оновлення. Злиттям мікросервісів намагаються досягти скорочення накладних витрат на запуск, експлуатацію та обслуговування екземплярів серверів (web-серверів, СУБД тощо), а також у разі зміни трудової політики компанії (скорочення штату або заміна розробників на менш кваліфікованих). При виконанні об'єднання сервісів через їхню високу зв'язаність доцільно виконати злиття баз даних. Об'єднання надасть можливість прямого доступу до спільних даних, тобто забезпечить усі переваги підходу «спільна база даних». Під час виконання об'єднання сервісів матимуть місце такі операції рефакторингу доступу:

- розширення привілеїв доступу;
- злиття схем баз даних.

Додатковою перевагою злиття сервісів є підвищення швидкодії виконання запитів за рахунок виключення процесів аутентифікації, авторизації, серіалізації та десеріалізації при комунікації між сервісами.

*Операції категорії рефакторинг доступу.* Скотт В. Емблер, Прамодкумар Дж. Садаладж у своїй фундаментальній книзі «Рефакторинг баз даних: еволюційне проектування» запропонували шість категорій рефакторингу баз даних, а саме [10]:

- категорія рефакторинг структури направлена на зміну структур однієї або декількох таблиць чи представлень бази даних, включає в себе заміну зв'язків один-до-багатьох на асоціативні таблиці, поділ таблиць, перейменування таблиць тощо;
- категорія рефакторинг якості даних забезпечує ефективність збереження та роботи з такими даними, що досягається за рахунок додавання правил валідації, форматів введення даних, забезпечення неможливості залишати порожніми поля та використання значень за замовчуванням тощо;
- категорія рефакторинг посилальної цілісності акумулює операції, які забезпечують, що будь-які дані, маючи зовнішні ключі, не посилаються на видалені записи, а також при видаленні первинного ключа зовнішні ключі будуть відповідно реагувати;
- категорія рефакторинг архітектури направлена на зміни з метою поліпшення правил взаємодії, за якими зовнішні програми працюють з базою даних;
- категорія рефакторинг методів направлена на внесення змін у код збережуваних процедур, функцій або тригерів (приклад додавання та видалення параметрів процедури з метою підвищення загальної якості роботи бази даних);
- зміни, що не входять до операцій рефакторингу, впливають на семантику схеми бази даних, додаючи до неї нові елементи.

У [10] зазначено, що обрана стратегія розподілу за категоріями була введена з метою поліпшення подачі матеріалу та покращення розробки інструментальних засобів рефакторингу баз даних.

До цього переліку варто додати ще одну категорію рефакторингу баз даних — рефакторинг доступу. Категорія рефакторинг доступу акумулює в собі зміни в системі управління базою даних, що пов'язані з доступом до даних, тобто операції, пов'язані зі змінами розташування об'єкта бази даних, атрибутів аутентифікації користувача, авторизаційних прав користувача. Цього мінімально достатньо для успішного доступу до даних.

До цієї категорії належать такі операції [11]:

- зміна атрибутів аутентифікації;
- звуження привілеїв доступу;
- розширення привілеїв доступу;
- виділення схеми бази даних;
- злиття схем баз даних.

Варто детально розглянути кожну з перелічених операцій для їх розуміння та необхідності застосування.

Операція зміна атрибутів аутентифікації передбачає створення нового користувача для заміщення старого. Основними причинами застосування операції зміни атрибутів аутентифікації є компроментація пароля користувача або планова ротація паролей. Також її застосовують при підвищенні вимог до складності пароля, зміні способу аутентифікації (використання Unix Socket, PAM тощо). Зміну атрибутів аутентифікації варто проводити через створення нового користувача задля збереження безперебійної роботи інформаційної системи.

Процес застосування операції зміни атрибутів аутентифікації:

1. Створення нового користувача. Головною вимогою до нового користувача є те, що він повинен володіти повністю ідентичними привілеями доступу (атрибутами авторизації), що і в старого користувача. Тобто множини привілеїв доступу старого ($P_{U1}$) та нового ($P_{U2}$) користувачів мають бути ідентичними — $P_{U1} \equiv P_{U2}$.

2. Заміна користувача бази даних у тестовому середовищі. За наявності тестування варто змоделювати виконання операції рефакторингу у тестовому середовищі, зокрема виконати заміну тестового користувача, що асоційований з користувачем у виробничому середовищі, та провести зміну налаштувань підключення до бази даних у тестових додатках. Для правильності проведення тестування необхідно в базі даних тестового середовища видалити старого користувача. За успішного переходу на використання нової конфігурації підключення та виконання всіх тестів можливий перехід до наступного пункту процесу застосування операції рефакторингу.

3. Заміна користувача бази даних у виробничому середовищі. В усіх програмних додатках, що здійснюють підключення до бази даних за допомогою старого користувача, виконується редагування конфігурації підключення вказанням атрибутів аутентифікації нового користувача. Заміну користувача варто виконувати у додатках послідовно.

4. Аудит бази даних на предмет наявності аутентифікації за атрибутами старого користувача. Пункт є не обов'язковим до виконання за наявності тестування, а в іншому випадку необхідно перевірити, чи не використовується старий користувач бази даних перед його видаленням. Усі сучасні СУБД мають у своєму арсеналі вбудовані інструменти чи плагіни для проведення аудиту безпеки. Авторами статті рекомендовано їх до використання під виконання поточного пункту. Проведення аудиту безпеки цікавить тільки для виявлення використання старого користувача після міграції всіх програмних додатків на використання нового користувача.

5. Видалення старого користувача. Після того, як перевірено, шо старий користувач не використовується, його необхідно видалити. Видалення рого користувача не є обов'язковим, але наполегливо рекомендується для уникнення несанкціонованого доступу до бази даних за його використанням.

Схематично процес застосування операції зміни атрибутів аутентифікації відображено на рис. 3.

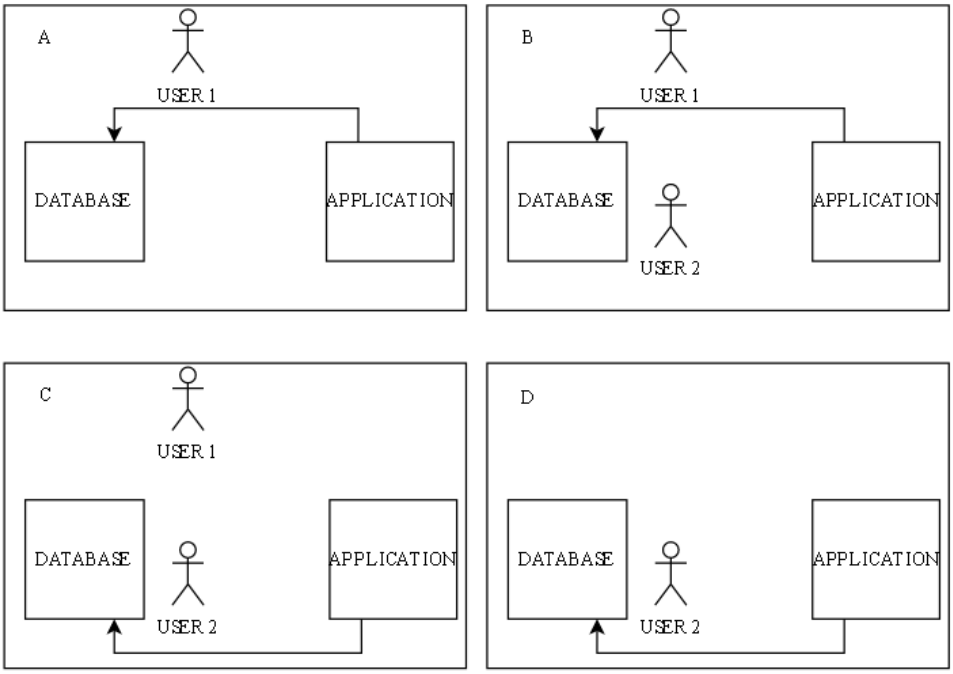


**Рис. 3. Схематичне зображення процесу застосування операції зміни атрибутів аутентифікації**

Розглянемо наступну операцію рефакторингу доступу — операцію звуження привілеїв доступу. Метою впровадження цієї операції є скасування привілеїв доступу до об'єктів бази даних, що на момент впровадження операції рефакторингу були надані. Потреба використання цієї операції виникає після виділення вузького місця в окрему базу даних, при перетворенні монолітної системи у сервіс-орієнтовану, а також при зміні політики доступу до даних співробітниками компанії.

Процес застосування операції звуження привілеїв доступу:

1. Створення нового користувача. Щоб гарантувати безперебійну роботу інформаційної системи, впровадження нової множини привілеїв доступу (атрибутів авторизації) необхідно здійснювати через створення нового користувача. Новий користувач має бути створений з цільовою множиною привілеїв доступу ($P_{U2}$) та бути повністю налаштованим до використання.

2. Використання нового користувача в місцях звуження привілеїв. У місцях доступу до даних, що відбуваються за привілеями, які підпали під звуження, виконуємо заміну старого користувача на нового. Новий користувач використовується через окреме підключення до бази даних.

3. Використання нового користувача бази даних у тестовому середовищі. Доцільно змоделювати проведення рефакторингу у тестовому середовищі, якщо це можливо. Необхідно використати нового користувача, що відповідає користувачу у виробничому середовищі, та замінити підключення до бази даних у місцях доступу до даних, що відбуваються за привілеями, які підпали під

звуження. Для коректного проведення тестування у старого користувача видаляються привілеї з бази даних тестового середовища. Коли всі тести будуть успішно пройдені, можливий перехід до наступного етапу операції рефакторингу.

4. Використання нового користувача бази даних у виробничому середовищі. Необхідно провести конфігурацію окремого підключення в усіх програмних додатках, із зазначенням атрибутів аутентифікації нового користувача. Рекомендовано виконувати заміну користувача у додатках послідовно.

5. Аудит бази даних на предмет наявності авторизації старого користувача за привілеями, які підпали під звуження. Цей пункт необхідний для виявлення використання старого користувача, проте не обов'язковий до виконання, якщо було проведено тестування. В іншому випадку, перед видаленням привілеїв доступу зі старого користувача, необхідно пересвідчитись у тому, що вони не використовуються. Авторами статті рекомендовано використовувати інструменти та плагіни для проведення аудиту безпеки, що вбудовані в усі сучасні СУБД.

6. Видалення зі старого користувача привілеїв, які підпали під звуження. Після перевірки того, що привілеї не використовуються, їх можна видалити. Кінцеві привілеї старого користувача ($P'_{U1}$) повинні набути вигляду: $P'_{U1} = P_{U1} - (P_{U1} \cap P_{U2})$, де $P_{U1}$ — початкові привілеї доступу старого користувача. Це не є обов'язковим, проте рекомендується з метою безпеки даних для уникнення несанкціонованого доступу.

Схематично процес застосування операції звуження привілеїв доступу відображено на рис. 4.

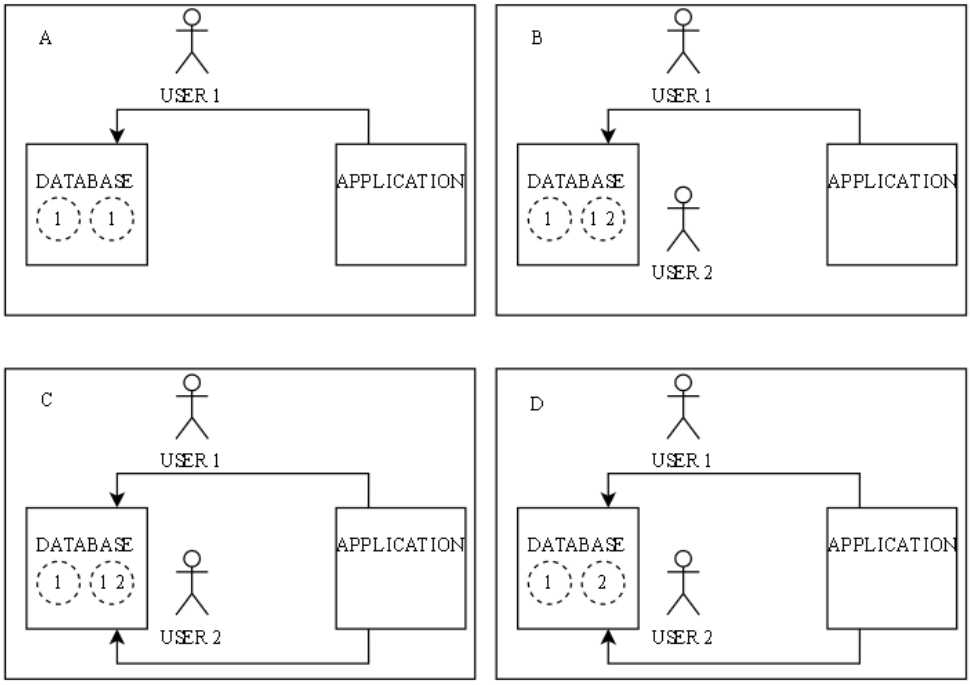


**Рис. 4. Схематичне зображення процесу застосування операції звуження привілеїв доступу**

Операція розширення привілеїв доступу впроваджується для надання існуючому користувачу прав доступу до об'єктів бази даних, що йому недоступні. Найчастіше використовується після зміни схеми бази даних, що впроваджувалися задля доповнення або додавання до відображення в реляційній базі даних сутностей реального світу (створення таблиць, процедур тощо).

Процес впровадження операції розширення привілеїв доступу проходить в один етап — наданні існуючому користувачеві привілеїв, що йому необхідні. На відміну від операцій рефакторингу, описаних вище, новий користувач не створюється, адже не виникає посилення передумов відповідно до принципу підстановки Барбари Лісков і, як наслідок, зберігається зворотна сумісність між наявною та новою схемою доступу до даних. Хоча принцип підстановки Барбари Лісков сформульований у рамках об'єктно-орієнтованого програмування, він може бути застосований до інших сфер за допомогою методу екстраполяції, тобто перенесення знань з однієї предметної області на іншу. Обов'язковою умовою до нової множини привілеїв доступу ($P'_{U1}$) є те, що вона повинна являти собою об'єднану множину з наявних ($P_{U1}$) і надаваних ($P_{ext}$) привілеїв доступу $P'_{U1} = P_{U1} \cup P_{ext}$.

Схематично процес застосування операції розширення привілеїв доступу відображено на рис. 5.

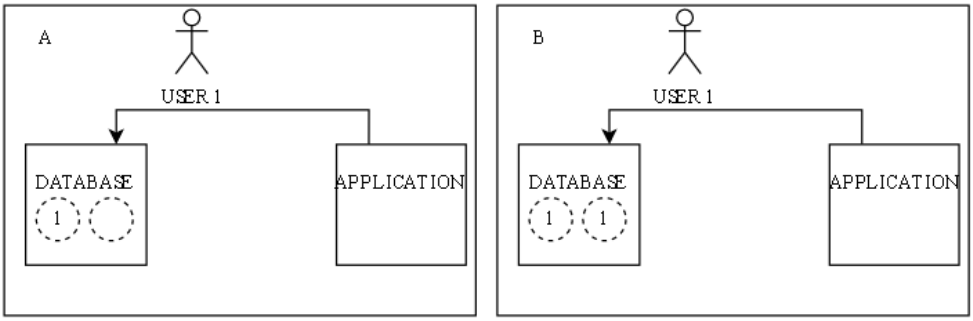


**Рис. 5. Схематичне зображення застосування операції розширення привілеїв доступу**

*Операція виділення схеми бази даних.* Метою впровадження операції є виокремлення групи таблиць, представлень і процедур зберігання, що їх обслуговують (сегменту даних), до окремої бази даних. Застосування цієї операції знаходить своє місце під час переходу між варіантами взаємодії з базою даних у сервіс-орієнтованій архітектурі, зокрема від варіанта «спільна база даних» до варіанта «бази даних кожному сервісу» за потреби подальшого масштабування операцій зчитування, запису або обчислення.

Процес застосування операції виділення схеми бази даних передбачає:

1. Визначення сегмента даних. Сегмент даних, які необхідно перемістити в окрему базу даних, відбувається відповідно до сервісу, що буде його обслуговувати. Важливо виділити сегмент даних таким чином, щоб сервіс відповідав базовому принципу розподілення відповідальності «інформаційний експерт», що входить до групи шаблонів GRASP, тобто, виділяючи з монолітної

системи сервіс за однією з двох стратегій, необхідно забезпечити достатній набір даних, щоб він набув статусу інформаційного експерта.

2. Видалення зв'язаності з переміщуваними даними. На цьому етапі застосовується комплекс заходів з використанням операцій рефакторингу з категорій рефакторинг посилальної цілісності, рефакторинг структури, рефакторинг методів з метою знищення будь-якої зв'язаності між переміщуваними даними та даними, що залишаються в глобальній базі даних. Зв'язаностями також вважаються випадки наявності зовнішніх ключів між таблицями різних сегментів і використання цих таблиць у представленнях і процедурах зберігання.

3. Створення нового користувача. Для роботи з виокремленим сегментом даних створюється новий користувач з метою обмеження доступу.

4. Взаємодія через нового користувача. В програмних додатках, що використовують базу даних, необхідно виконати зміни з використанням нового користувача в окремо створеному підключенні при взаємодіях з сегментом даних, що виділяється. Для коректного проведення цього етапу варто звернутися до рефакторингу коду. Кінцевою метою цього етапу є локалізація алгоритмів, що виконують обробку сегмента даних, що виділяється, в окремий модуль, та формалізація інтерфейсу взаємодії з програмним модулем.

5. Перевірка працездатності інформаційної системи у тестовому середовищі. Виконуючи цей пункт операції рефакторингу, важливо пересвідчитись у тому, що не виникло порушень у роботі алгоритмів інформаційної системи. Для правильності проведення тестування необхідно попередньо скасувати привілеї доступу старого користувача до об'єктів бази даних, що виокремлюються. За умови успішного виконання всіх тестів та відсутності звернень до виокремлених даних від імені старого користувача можливий перехід до наступного пункту процесу застосування операції рефакторингу.

6. Аудит бази даних на предмет звернень до виокремлених даних від імені старого користувача. За відсутності стовідсоткового покриття тестами взаємодії з базою даних варто застосувати інструменти чи плагіни аудиту безпеки. Їх використання допоможе виявити місця в алгоритмах інформаційної системи, що виконують доступ до даних від імені старого користувача.

7. Скасування привілеїв доступу старого користувача до виокремленого сегмента даних. Після того, як перевірено, шо старий користувач не здійснює звернень до виокремлених даних, виконується скасування привілеїв доступу.

8. Створення нової бази даних і виконання міграції даних. Спочатку виконується створення нової бази даних, що за фізичною структурою повністю ідентична до виокремленого сегмента, а також користувача до неї, що за привілеями доступу є відповідним до користувача, створеного в пункті 3. Далі виконується налаштування міграції даних зі старої бази даних до новоствореної. Авторами цієї статті рекомендується виконувати міграцію даних за допомогою інструментів реплікації. На момент написання статті майже всі популярні СУБД мають у своєму арсеналі можливість проводити реплікацію окремих таблиць, а не всієї бази даних. Якщо використовувана СУБД не підтримує реплікацію на рівні таблиці, то можливе налаштування репліки всієї старої бази даних з подальшим

видаленням з неї об'єктів, що не відносяться до виокремленого сегмента. Видаляти зайві об'єкти варто після переключення інформаційної системи на нову базу даних і знищення реплікації.

9. Заміна нового користувача старої бази даних на користувача нової бази даних. Після завершення міграції даних необхідно виконати заміну користувача, що здійснює взаємодію з виокремленим сегментом у старій базі даних на користувача, створеного для взаємодії з новою базою даних, що створена у пункті 8. Після виконання переключення користувачів необхідно зупинити роботу інструментів, що підтримують міграцію у наближеному до реального часу.

10. Видалення виокремленого сегмента зі старої бази даних. Після остаточного переходу на нову базу даних варто виконати очистку старої бази даних від об'єктів, що відносяться до виокремленого сегмента даних. Цей пункт не є обов'язковим, проте рекомендований до виконання з метою зменшення технічного боргу.

*Операція злиття схем баз даних.* На відміну від попередньої операції рефакторингу ця операція передбачає злиття різних баз даних в єдину з метою спрощення реалізації бізнес-транзакцій і забезпечення консенсетності даних. Відповідно до теореми CAP [12], ми відмовляємось від стійкості до розподілення, а як перевагу отримуємо узгодженість і доступність. Можливе застосування цієї операцій знаходить своє місце під час переходу від бази даних кожному сервісу до спільної бази даних.

Процес застосування операції злиття схем баз даних:

1. Уніфікація назв об'єктів баз даних, що зливаються. Перед початком безпосереднього злиття баз даних необхідно провести уніфікацію назв таблиць, представлень, користувачів та інших об'єктів баз даних, що зливаються. Виконання цього пункту необхідне для уникнення конфліктів між іменами при злитті. Авторами статті рекомендується використовувати операції рефакторингу структури.

2. Визначення головної бази даних з-поміж тих, що зливаються. Щоб оптимізувати проведення рефакторингу та не створювати нову базу даних, необхідно вибрати головну базу даних, до якої буде виконуватись злиття другорядної бази даних.

3. Міграції даних. Спочатку до головної бази необхідно скопіювати фізичну структуру, повністю ідентичну до другорядної бази даних. Потім виконується копіювання користувачів, що взаємодіють з другорядною базою даних зі збереженням їхніх привілеїв доступу. Далі виконується налаштування міграції даних із другорядної бази даних до головної. Деталі використання реплікації задля проведення міграції даних описані в операції виділення бази даних.

4. Заміна користувача другорядної бази даних на відповідного користувача головної бази даних. Після завершення міграції даних необхідно виконати заміну користувачів, що взаємодіють з другорядною базою даних, на нових користувачів головної бази даних, що були створені на попередньому етапі. Після цього, так само як і в операції виділення схеми бази даних, необхідно

зупинити роботу інструментів, що підтримують міграцію у наближеному до реального часу.

5. Перевірка працездатності інформаційної системи у тестовому середовищі. Виконання цього пункту дає змогу пересвідчитися в тому, що не виникло збоїв у роботі інформаційної системи. Для правильності проведення тестування необхідно попередньо з користувачів другорядної бази даних скасувати всі привілеї доступу. Після успішного виконання всіх тестів і відсутності звернень до даних від імені користувачів другорядної бази даних можливий перехід до наступного етапу операції рефакторингу.

6. Аудит бази даних на предмет звернень до даних від імені користувачів другорядної бази даних. У разі відсутності повного покриття тестами взаємодії з базою даних варто застосувати інструменти та плагіни аудиту безпеки. Використання цих засобів дасть змогу виявити місця в алгоритмах системи, що виконують доступ до даних від імені користувачів другорядної бази даних.

7. Видалення другорядної бази даних. Після остаточного перенесення даних до головної бази даних і переключення користувачів варто видалити другорядну базу даних. Цей пункт не є обов'язковим, проте рекомендований до виконання для уникнення витоку даних.

## Висновки

Підсумовуючи, варто зазначити, що для якісної та швидкої розробки програмного забезпечення, а також для економії коштів, що можуть піти на виправлення архітектурної помилки інформаційної системи, необхідно відповідально ставитися до вибору шаблону архітектури. При створенні сучасних інформаційних систем розробники намагаються забезпечити їхню високу адаптивність під вимоги ринку, що стрімко змінюються. У статті наведені переваги та недоліки розповсюджених шаблонів архітектури програмного забезпечення. При тривалій розробці та підтримці інформаційної системи її програмний код може стати доволі важким у сприйнятті розробниками, тому виникає необхідність проведення рефакторингу. Запропоновано нову категорію рефакторингу — рефакторинг доступу, що акумулює в собі зміни, які пов'язані з доступом до об'єктів бази даних, у системі управління базою даних. Передумова її створення — відсутність операцій, що застосовуються для реагування на події політики безпеки (ротація паролей, компрометація паролей, зміна вимог до складності паролей, зміна способу аутентифікації тощо), зміни у правах доступу та масштабування баз даних.

## Література

1. Tanenbaum A. S., Steen M.V. Distributed systems: principles and paradigms. Pearson Prentice Hall, Pearson Education, Inc. 2007. 685 с.

2. Crowcroft J. Transparencies. Open Distributed Systems. Department of Computer Science and Technology, University of Cambridge. 18 с.

3. Howard M., LeBlanc D., Viega J. 24 DEADLY SINS OF SOFTWARE SECURITY Programming Flaws and How to Fix Them. McGraw-Hill Education. 2010. 393 с.

4. Nath K. Understanding database scaling patterns. URL: https://medium.com/@kousiknath/understanding-database-scaling-patterns-ac24e5223522 (дата звернення: 12.12.2019).

5. Drake M. Understanding Database Sharding. URL: https://www.digitalocean.com/community/tutorials/understanding-database-sharding (дата звернення: 12.12.2019).

6. Richardson C. Pattern: Decompose by business capability Context. URL: https://microservices.io/patterns/decomposition/decompose-by-business-capability.html (дата звернення: 04.11.2019).

7. Business Capabilities determine Microservices. URL: https://www.capstera.com/business-capabilities-determine-microservices/ (дата звернення: 06.11.2019).

8. Richardson C. Pattern: Saga. URL: https://microservices.io/patterns/data/saga.html (дата звернення: 04.11.2019).

9. Richardson C. Pattern: Shared database Context. URL: https://microservices.io/patterns/data/shared-database.html (дата звернення: 06.11.2019).

10. Эмблер С., Садаладж П. Рефакторинг баз данных. Эволюционное проектирование. Вильямс, Москва, Россия. 2016. 368 с.

11. Струзік В. А. Категорія рефакторинг доступу. Міжнародна науково-технічна конференція студентів, аспірантів та молодих вчених «Комп'ютерні науки, інформаційні технології та системи управління». 27—29 листопада 2019 року. Івано-Франківськ. С. 20—21.

12. Brewer E. CAP Twelve Years Later: How the "Rules" Have Changed URL: https://www.infoq.com/articles/cap-twelve-years-later-how-the-rules-have-changed. (дата звернення: 24.11.2019).