Національний технічний університет України
«Київський політехнічний інститут імені Ігоря Сікорського»

Інформаційні системи

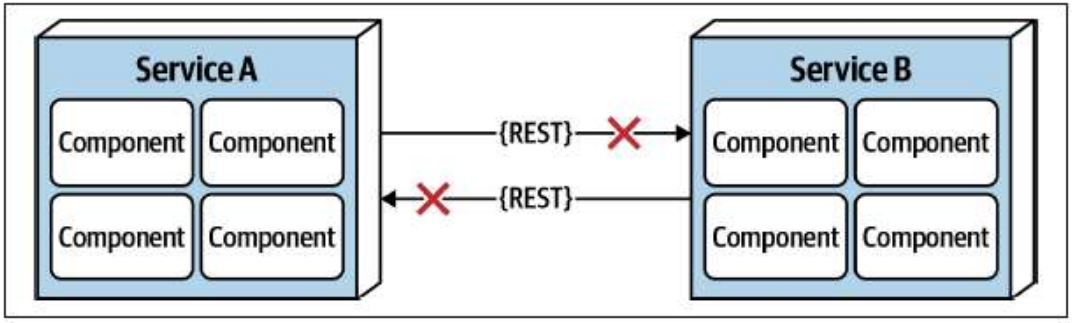
Викладач: к.т.н., доц. Саяпіна Інна Олександрівна

План заняття:

- Помилкові уявлення про організацію розподілених архітектур
- Багатошарова архітектура (Layered Architecture)

Помилкове уявлення #1
Мережа є надійною

- Розробники і архітектори припускають, що мережа надійна, але це не так. Це важливо для всіх розподілених архітектур, тому що всі стилі розподіленої архітектури покладаються на мережу для зв'язку до та від служб, а також між службами.
- Наприклад, служба B може бути справною, але служба A не може отримати доступ до неї через проблему з мережею; або ще гірше, служба A надіслала запит службі B для обробки деяких даних і не отримала відповіді через проблему з мережею.
- Ось чому між службами існують такі речі, як тайм-аути та Circuit breakers. Чим більше система покладається на мережу (наприклад, архітектура мікросервісів), тим потенційно менш надійною вона стає.

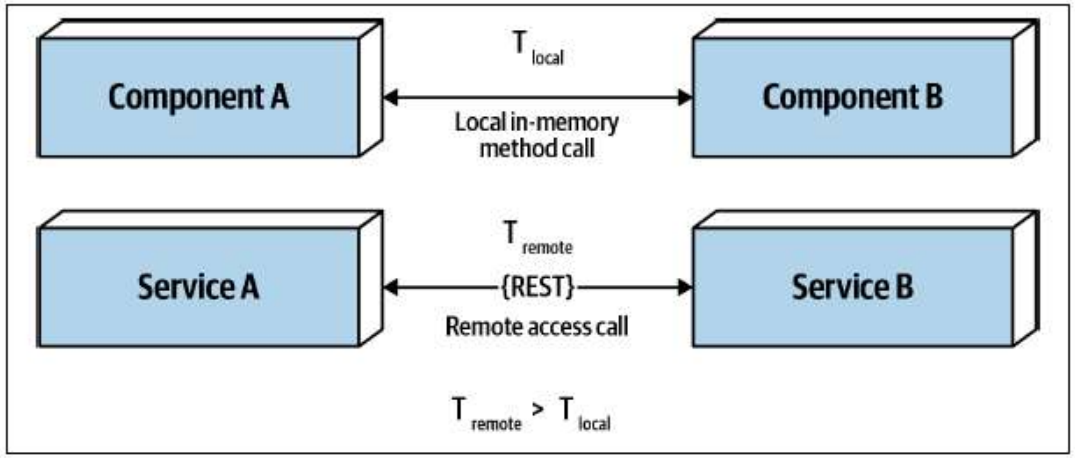


Representational State Transfer (REST)

- Це архітектурний стиль, який визначає набір принципів і обмежень для створення веб-сервісів. Основні принципи архітектури RESTful включають:
  - **Архітектура клієнт-сервер**. Клієнт і сервер є окремими та взаємодіють за допомогою чітко визначеного інтерфейсу, зазвичай за допомогою HTTP.
  - **Зв'язок без збереження стану**: кожен запит від клієнта до сервера повинен містити всю необхідну інформацію для обробки запиту. Сервер не зберігає клієнтський контекст між запитами.
  - **Кешування**: відповіді від сервера можуть кешуватися клієнтом або будь-якими проміжними кешами для підвищення продуктивності.
  - **Багаторівнева архітектура**: клієнт взаємодіє з сервером через уніфікований інтерфейс, не знаючи про будь-які основні деталі реалізації або проміжні компоненти.
  - Веб-служби RESTful зазвичай використовують такі методи HTTP, як GET, POST, PUT і DELETE для взаємодії з ресурсами на сервері. Ресурси ідентифікуються унікальним URL, і сервер повертає представлення цих ресурсів у відповідь на запити клієнта.
- Популярність веб-сервісів RESTful значно зросла за останні роки завдяки їх простоті, масштабованості та зручності використання. Вони широко використовуються для створення API, мікросервісів та інших розподілених систем.

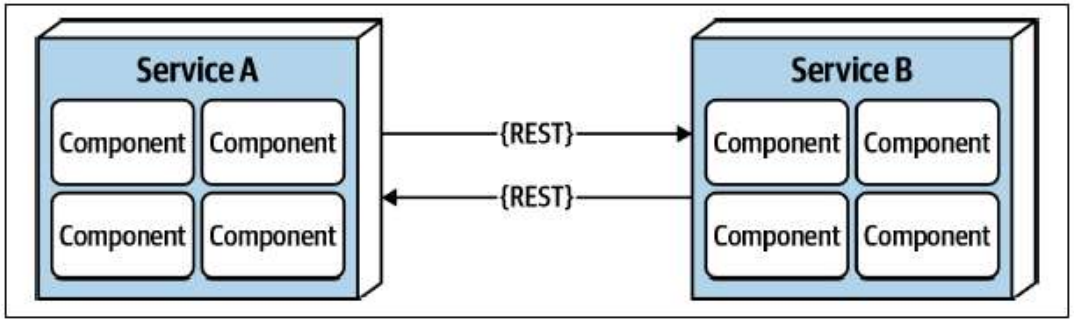
Помилкове уявлення #2
Затримка дорівнює нулю

- Час здійснення локального виклику до іншого компонента через виклик методу або функції (*t_local*) вимірюється в наносекундах або мікросекундах. Однак, той самий виклик через протокол віддаленого доступу (наприклад, REST або RPC), час доступу (*t_remote*) вимірюється в мілісекундах. Тому *t_remote* завжди буде більшим за *t_local*. Затримка в будь-якій розподіленій архітектурі не дорівнює нулю.
- Використовуючи будь-яку розподілену архітектуру, архітектори повинні знати середнє значення затримки, щоб визначити, чи можлива розподілена архітектура.



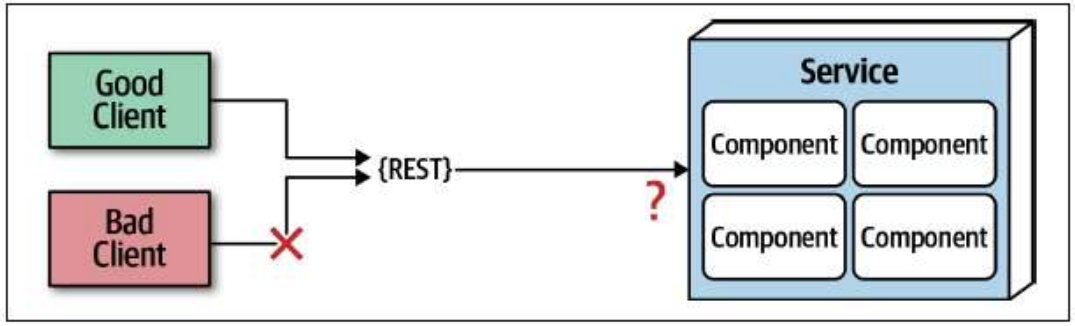
Помилкове уявлення #3
Пропускна здатність нескінченна

- Коли системи розбиваються на менші одиниці розгортання (сервіси) у розподіленій архітектурі, такій як мікросервіси, зв'язок із цими сервісами та між ними підвищує використання пропускної здатності, спричиняючи сповільнення мережі, що впливає на затримку і надійність.
- Зчеплення штампів (stamp coupling/data structure coupling) - це тип зчеплення в розробці програмного забезпечення, де модулі залежать від формату або порядку даних, що передаються між ними. Це може зробити код більш крихким і складним для підтримки з часом.



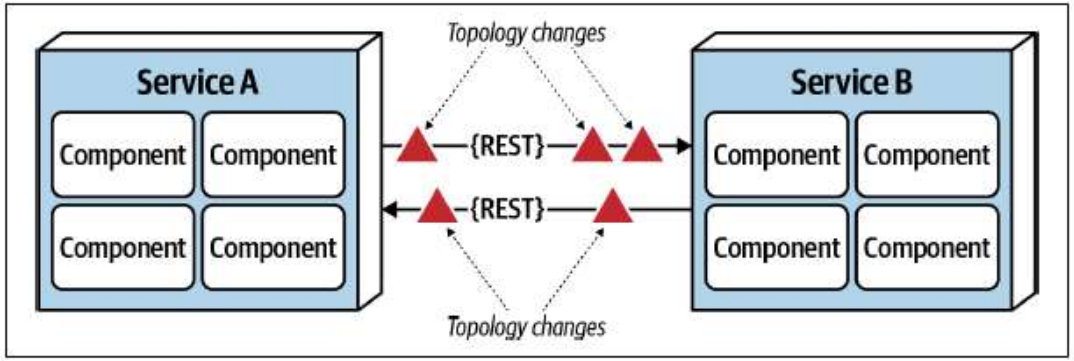
Помилкове уявлення #4
Мережа є безпечною

- Безпека стає набагато складнішою в розподіленій архітектурі. Кожна кінцева точка кожного розподіленого блоку розгортання має бути захищена, щоб невідомі або неправильні запити не надходили до цієї служби.
- При переході від монолітної до розподіленої архітектури площа поверхні для загроз і атак збільшується у рази. Необхідність захищати кожну кінцеву точку, навіть якщо здійснюється міжсервісний зв'язок, є ще однією причиною, чому продуктивність, як правило, нижча в синхронних, розподілених архітектурах, таких як мікросервіси.



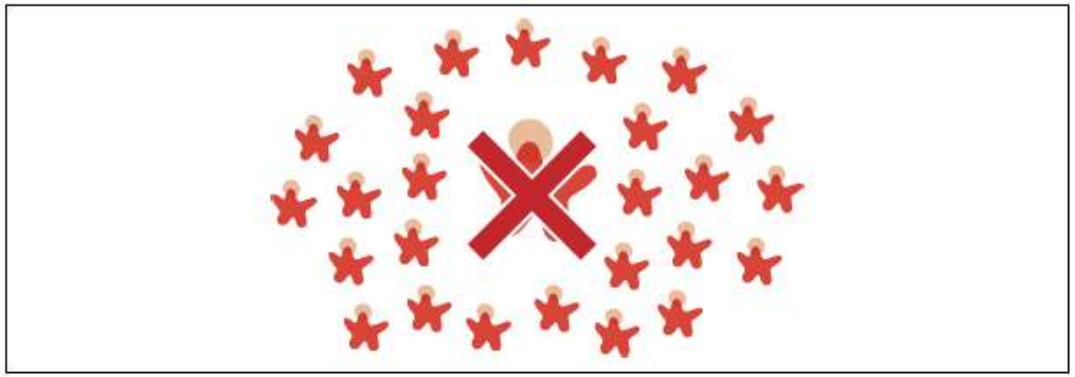
Помилкове уявлення #5
Топологія ніколи не змінюється

- Архітектори припускають, що топологія, включаючи всі маршрутизатори, вузли, комутатори, брандмауери, мережі та прилади, що використовуються в загальній мережі, статична і ніколи не змінюється. Але вона весь час змінюється.
- Яке значення цієї помилки?
- Навіть "незначне" мережеве оновлення може нівелювати усі припущення архітектора щодо затримки, викликаючи тайм-аути та circuit breakers. Архітектори повинні постійно спілкуватися з адміністраторами мережі, щоб знати майбутні зміни і оперативно вносити корективи.



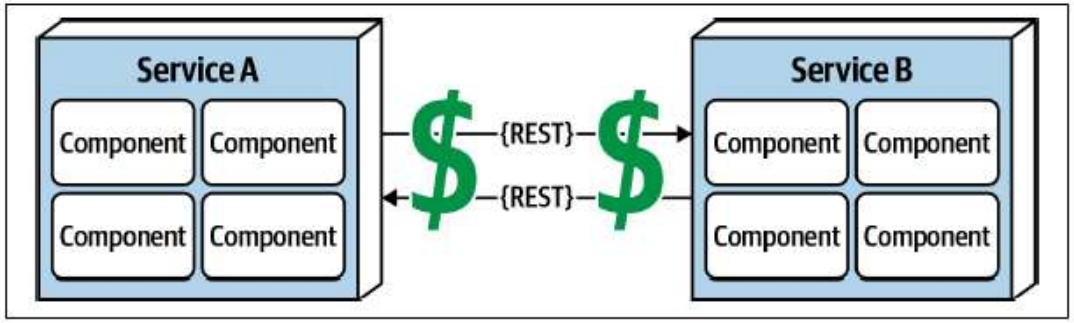
Помилкове уявлення #6
Є лише один адміністратор

- Архітектори весь час припускаються цієї помилки, вважаючи, що їм потрібно співпрацювати та спілкуватися лише з одним адміністратором. Але у типовій великій компанії є десятки мережевих адміністраторів.
- Внаслідок складності розподіленої архітектури потрібна координація з усіма мережевими адміністраторами, щоб все працювало правильно.



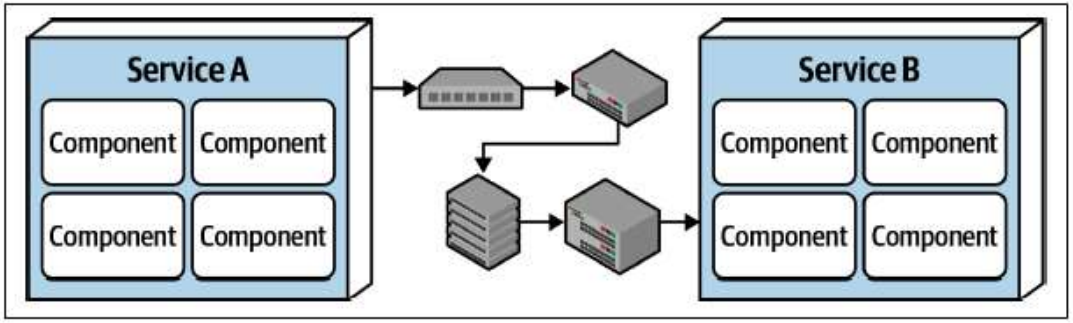
Помилкове уявлення #7
Транспортні витрати дорівнюють нулю

- Архітектори помилково припускають, що необхідна інфраструктура є на місці та є достатньою для здійснення простого RESTful звернення або розбиття монолітної програми. Зазвичай розподілені архітектури коштують значно більше, ніж монолітні архітектури, через збільшення потреб у додатковому обладнанні, серверах, шлюзах, брандмауерах, нових підмережах, проксі-серверах тощо.
- Щоразу розглядаючи можливість розподіленої архітектури, треба аналізувати поточну топологію сервера та мережі з огляду на пропускну здатність, полосу пропусканя, затримку та забезпечення безпеки, щоб не потрапити у пастку неочікуваної помилки.



Помилкове уявлення #8
Мережа є однорідною

- Більшість архітекторів і розробників припускають, що мережа є однорідною — складається лише з одного постачальника мережевого обладнання. Але більшість компаній мають у своїй інфраструктурі кілька постачальників мережевого обладнання і не всі ці різнорідні постачальники апаратного забезпечення добре взаємодіють. Можуть бути ситуації, які не були повністю перевірені, і тому мережеві пакети час від часу втрачаються



Інші помилкові уявлення
Розподілене ведення журналу (Distributed logging)

- Через розподіл програмних і системних журналів у розподіленій архітектурі пошук першопричини, чому конкретна операція була відкинута, є дуже складним і трудомістким.
- Розподілені архітектури містять від десятків до сотень різних журналів, усі розташовані в різних місцях і мають різний формат, що ускладнює виявлення проблеми.
- Інструменти консолідації ведення журналів, такі як Splunk, допомагають консолідувати інформацію з різних джерел і систем в один консолідований журнал, але до кінця не вирішують проблеми, пов’язаніі з розподіленим веденням журналу.

Інші помилкові уявлення
Розподілені транзакції (Distributed transactions)

- Реалізація вимог ACID для розподілених архітектур є набагато складнішою, ніж для монолітних. Розподілені архітектури покладаються на кінцеву узгодженість (eventual consistency), щоб гарантувати, що дані, оброблені окремими блоками розгортання, у певний невизначений момент часу синхронізуються в узгоджений стан. Це компроміс розподіленої архітектури: висока масштабованість, продуктивність і доступність в жертву узгодженості та цілісності даних.
- Кінцева узгодженість — це властивість розподілених систем, де за наявності достатнього часу та відсутності нових оновлень усі копії сховища даних зрештою будуть узгоджені одна з одною. Це означає, що дані можуть бути тимчасово неузгодженими в репліках.
- Шаблон Saga — це послідовність транзакцій, яка оновлює кожну службу та публікує повідомлення чи подію, щоб ініціювати наступний крок транзакції. Якщо крок не вдається, сага виконує компенсаційні транзакції, які протидіють попереднім транзакціям.
- Також використовуються транзакції BASE

Інші помилкові уявлення
Розподілені транзакції (Distributed transactions)

- **BASE** — це набір принципів, які часто використовуються для розподілених систем. На відміну від властивостей ACID, які надають перевагу чіткій узгодженості та правильності над доступністю, BASE-транзакції надають перевагу над сильною узгодженістю. BASE-транзакції характеризуються такими принципами:
  - **Basically Available** : система повинна бути розроблена так, щоб бути завжди доступною та реагувати на запити користувачів, навіть у разі часткових збоїв або тимчасових збоїв.
  - **Soft state** : стан системи може змінюватися з часом, навіть за відсутності зовнішніх вхідних даних, і цю зміну слід терпіти.
  - **Eventually consistent** : система не обов'язково повинна бути негайною або сильно узгодженою, але з часом стане узгодженою з часом, маючи достатньо часу для поширення оновлень у системі.
- Транзакції BASE не забезпечують такий самий рівень узгодженості та гарантій коректності, як транзакції ACID, і можуть вимагати ретельного проектування та керування, щоб забезпечити підтримку узгодженості даних протягом тривалого часу.

Інші помилкові уявлення
Ведення контрактів та управління версіями (Contract maintenance and versioning)

- Ще одним особливо складним завданням у рамках розподіленої архітектури є створення контрактів, обслуговування та керування версіями.
- **Контракт** - це поведінка та дані, узгоджені як клієнтом, так і службою.
- Супровід контрактів ускладнюється у розподілених архітектурах через незв'язані сервіси та системи, що належать різним командам та відділам. Ще складнішими є комунікаційні моделі, необхідні для застарілих версій.

План заняття:

- Багатошарова архітектура (Layered Architecture):
  - особливості застосування
  - топологія та призначення шарів
  - приклади
  - принципи проєктування
  - закриті/відкриті шари, антипатерн «архітектурна воронка»
  - основні характеристики використання

Монолітна архітектура (Monolithic Architecture)

- Представляє вироджений випадок архітектури : до складу ПЗ входить тільки одна програма.
- Таку архітектуру вибирають зазвичай у тому випадку, коли ПЗ повинно виконувати одну яскраво виражену функцію і її реалізація не представляється дуже складною.

```
public class MonolithicApplication {
    public static void main(String[] args) {
        // Code for the entire application goes here
        // No separation of concerns
        // No modularity or scalability
    }
}
```

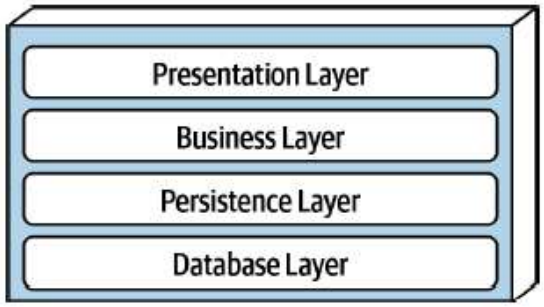
Багатошарова архітектура (Layered Architecture)

- Терміни шар (layer) і рівень(tier) часто змішують. Однак -
  - • layer (**шар)** позначає логічне розділення функціональності,
  - • tier (**рівень, ярус)** позначає фізичне розгортання на різних системах.

- є одним із найпоширеніших архітектурних стилів
- в шари традиційної багаторівневої архітектури добре вписуються організаційні рівні більшості компаній-розробників ПЗ: розробники інтерфейсу користувача, розробники серверної частини, експерти з баз даних тощо.
- Якщо розробник або архітектор не впевнений, який стиль архітектури вони використовують, або якщо команда розробників Agile «тільки починає кодувати», велика ймовірність, що вони впроваджують стиль багатошарової архітектури.

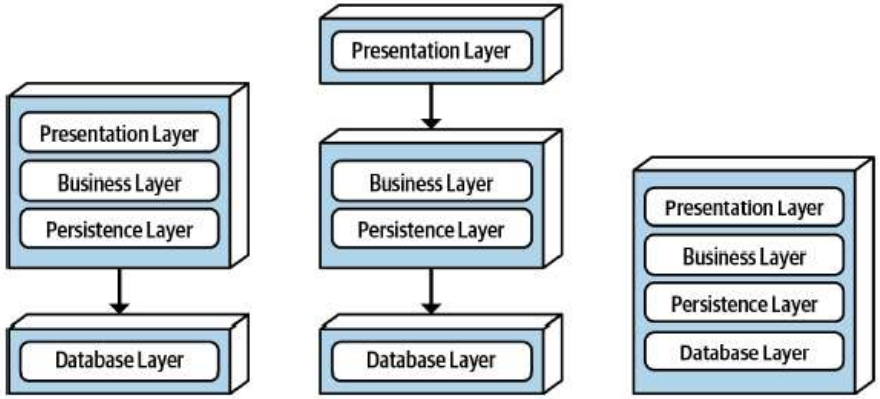
Топологія багатошарової архітектури

- Компоненти в багатошаровій архітектурі організовані в логічні горизонтальні шари, кожен шар виконує певну роль у програмі
- Немає конкретних обмежень щодо кількості та типів шарів, але більшість багатошарових архітектур складаються з чотирьох стандартних шарів: презентації (Presentation), бізнесу (Business), збереження (Persistence/Data Access) та бази даних (Database).
- Менші програми можуть мати лише три шари, тоді як більші та складніші бізнес- програми можуть містити п'ять або більше шарів.



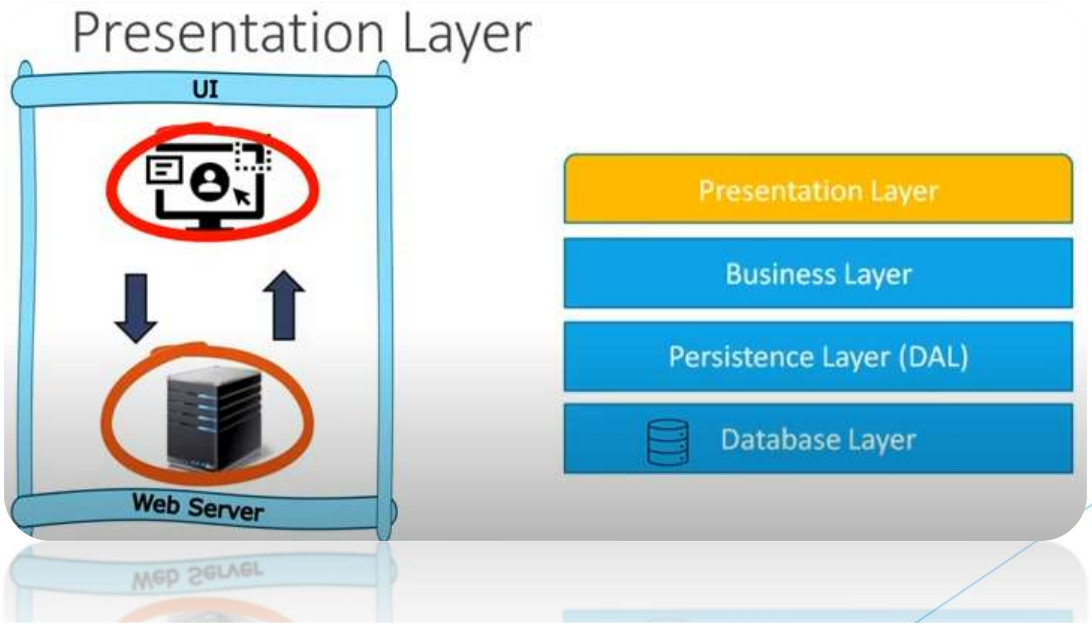
Топологія багатошарової архітектури. Варіанти топології з точки зору розгортання

- Перший варіант: Presentation, Business Layer і Persistence Layer - це єдиний блок розгортання, причому шар бази даних представлений як окрема зовнішня фізична база даних (або файлова система).
- Другий варіант: Presentation Layer фізично відокремлений у власний блок розгортання, а Business Layer і Persistence Layer об’єднані в другий блок розгортання.
- Третій варіант поєднує всі чотири стандартні шари в одне розгортання, включаючи шар бази даних. Цей варіант може бути корисним для невеликих додатків із внутрішньою вбудованою базою даних або базою даних у пам’яті.



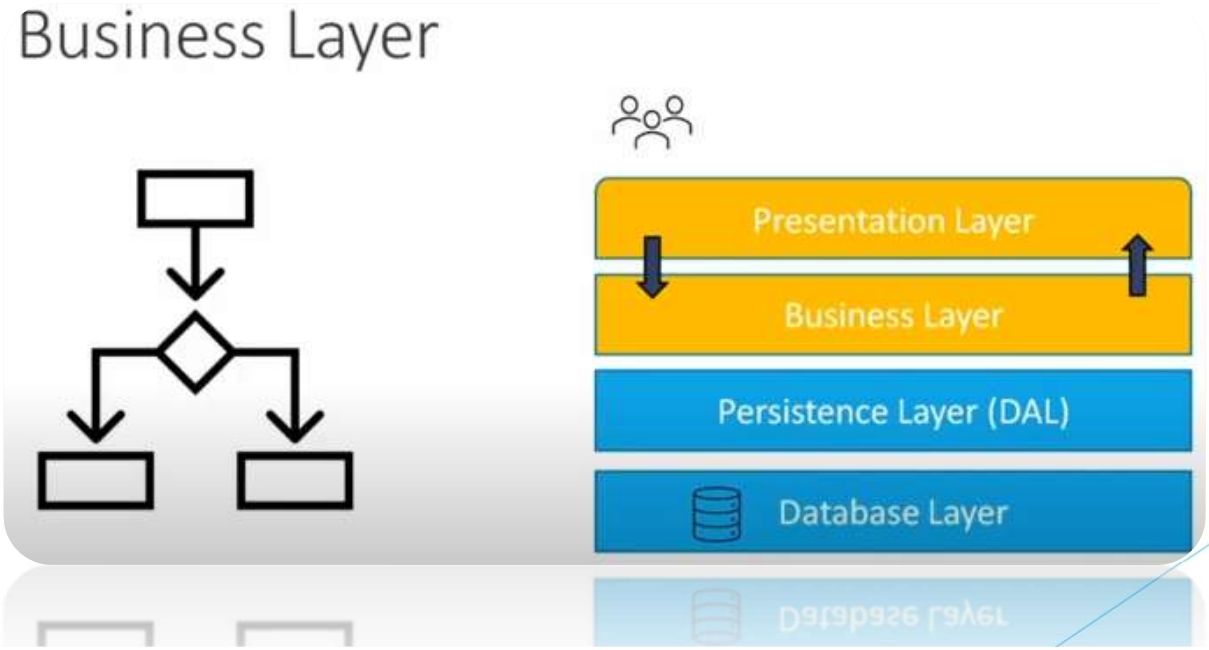
Топологія багатошарової архітектури

- **Шар представлення** містить код інтерфейсу користувача. Сюди входять розмітка сторінки, обробка дій користувача, логіка взаємодії з веб-сервером, перевірка даних, введених користувачем, логіка спілкування браузера, відправка та отримання HTTP запитів і відображення результату.
- За необхідності шар представлення надсилає дані на подальшу обробку в шар бізнес-логіки і, коли обробка завершена, повертає відповідь користувача



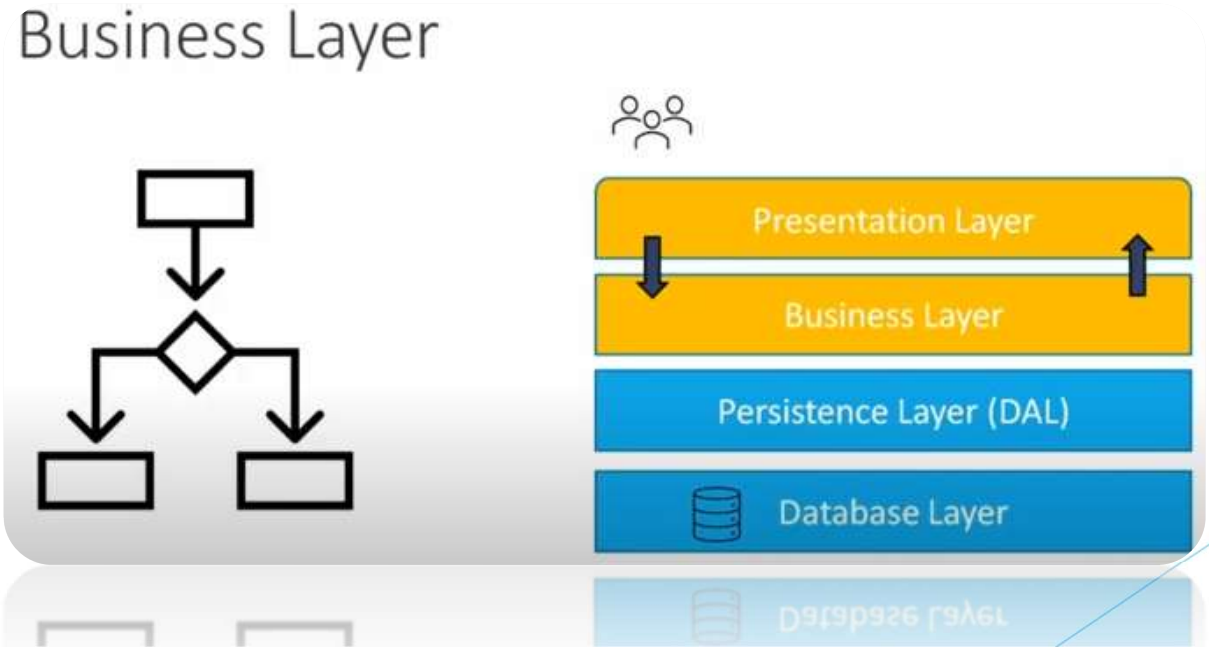
Топологія багатошарової архітектури

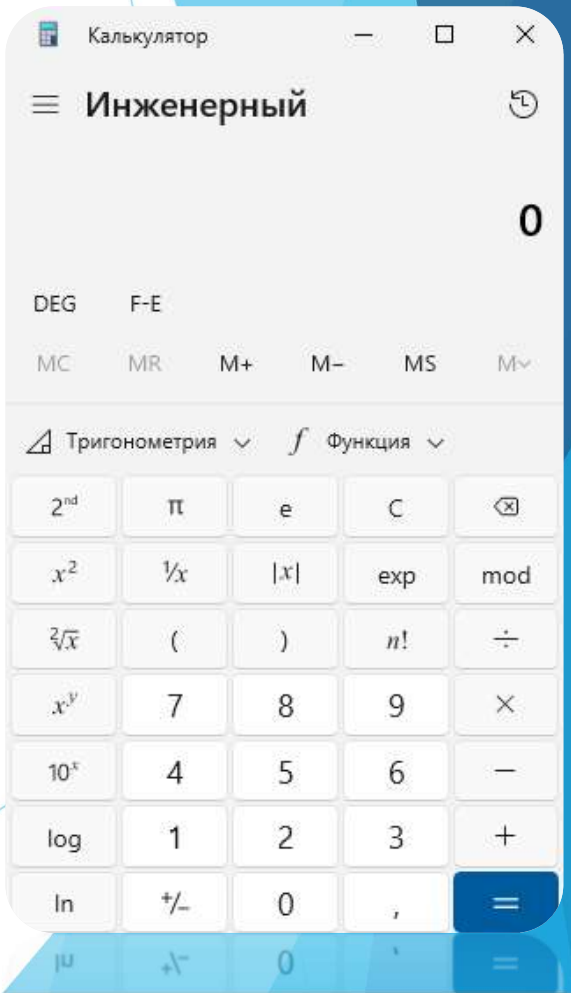
- Шар **Business Layer** виконує бізнес-логіку. Під **бізнес-логікою** слід розуміти обчислення, пов'язані з обробкою даних відповідно до бізнес правил, за якими працює система. При необхідності business layer звертається до persistence layer для доступу до даних.



Топологія багатошарової архітектури. Приклад.

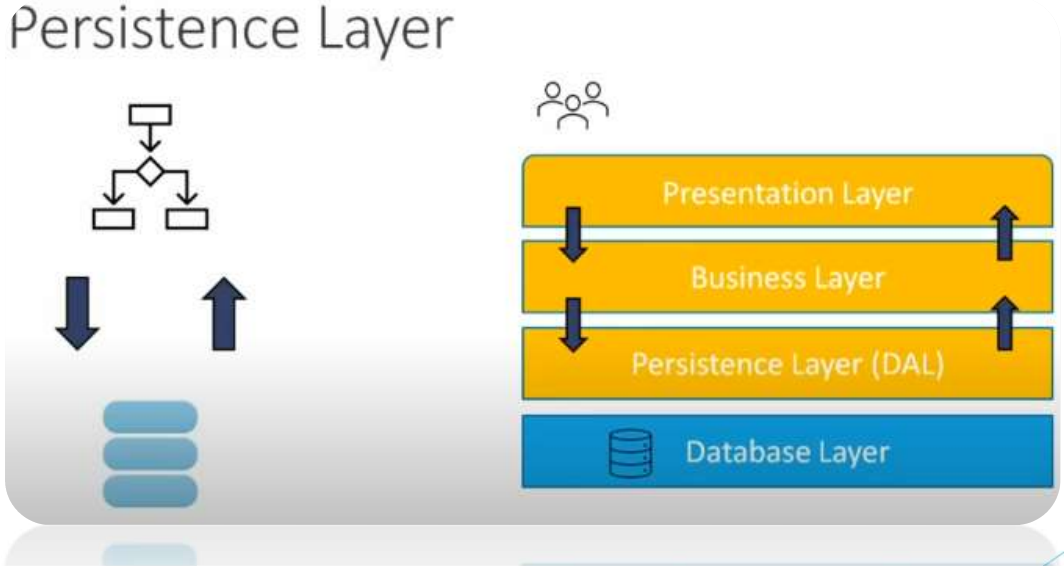
- Калькулятор: шар представлення - це кнопки і поля введення виводу і вікно калькуляції в цілому. Коли ми натискаємо кнопки - виконуються обчислення, ці обчислення є бізнес-логіка калькулятора.





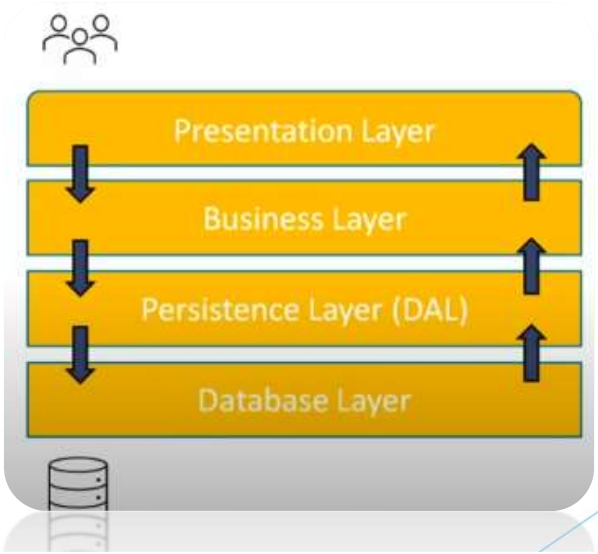
Топологія багатошарової архітектури.

- **Шар збереження/доступу до даних (Persistence/Data Access layer)**- цей шар приймає запити від шару бізнес-логіки та на їх основі формує запити до бази даних і, отримавши від бази даних результат, відправляє відповідь назад бізнес шару.
- Якщо ми працюємо з реляційними базами даних, то основне завдання persistence layer - це формування та відправлення прямих SQL запитів, або використання різних ORM, патернів Repository, Unit of Work



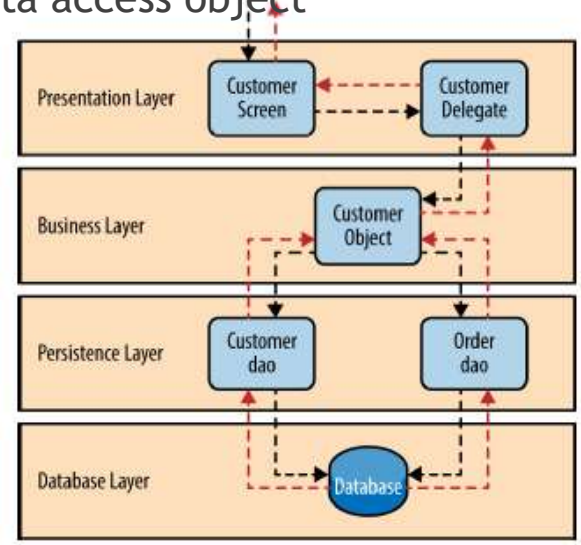
Топологія багатошарової архітектури.

- Під **Database Layer** зазвичай мається на увазі сама база даних.
- Якщо говорити в термінах веб-сервера - це драйвери бази даних, наприклад oracle data провайдер, ODBC, JDBC, OLE DB і так далі. Усередині додатка саме драйвери баз даних є database layer, оскільки код веб-сервера не знає, що таке база даних і він завжди має справу тільки з драйверами до цих баз даних



Приклад структури багатошарової архітектури.

- Приклад: розглянемо запит від бізнес-користувача на отримання інформації про клієнта для конкретної особи. Чорні стрілки показують запит, що надходить до бази даних для отримання даних про клієнта, а червоні стрілки показують відповідь, що повертається на екран для відображення даних. У цьому прикладі інформація про клієнта складається як з даних клієнта, так і з даних про замовлення розміщені клієнтом. Dao – це data access object



Приклад зв’язків між шарами у коді

```
public class PresentationLayer {
    private BusinessLayer business;

    public void setBusinessLayer(BusinessLayer business) {
        this.business = business;
    }

    public void start() {
        // Code for starting the presentation layer goes here
        // Interact with the business layer as necessary
    }
}

public class BusinessLayer {
    private DataAccessLayer dataAccess;

    public void setDataAccessLayer(DataAccessLayer dataAccess) {
        this.dataAccess = dataAccess;
    }

    // Methods for implementing business logic go here
}

public class DataAccessLayer {
    // Methods for accessing data go here
}

public class LayeredApplication {
    public static void main(String[] args) {
        // Instantiate objects of each layer and connect them
        PresentationLayer presentation = new PresentationLayer();
        BusinessLayer business = new BusinessLayer();
        DataAccessLayer dataAccess = new DataAccessLayer();
        presentation.setBusinessLayer(business);
        business.setDataAccessLayer(dataAccess);

        // Start the application
        presentation.start();
    }
}
```

Багатошарова архітектура. Приклад 2

Presentation Layer

```
public class ShopUI {
    private ShopController shopController;

    public void setShopController(ShopController shopController) {
        this.shopController = shopController;
    }

    public void displayProducts() {
        // display products using shopController
    }

    public void addToCart(Product product) {
        // add product to cart using shopController
    }

    // other UI methods
}
```

Controller Layer

```
public class ShopController {
    private ShopService shopService;

    public void setShopService(ShopService shopService) {
        this.shopService = shopService;
    }

    public List<Product> getProducts() {
        // call shopService to get products
    }

    public void addProductToCart(Product product) {
        // call shopService to add product to cart
    }

    // other controller methods
}
```

Service Layer

```
public class ShopService {
    private ProductRepository productRepository;
    private ShoppingCart shoppingCart;

    public void setProductRepository(ProductRepository productRepository
        this.productRepository = productRepository;
    }

    public void setShoppingCart(ShoppingCart shoppingCart) {
        this.shoppingCart = shoppingCart;
    }

    public List<Product> getProducts() {
        // call productRepository to get products
    }

    public void addProductToCart(Product product) {
        // add product to shopping cart
        shoppingCart.addProduct(product);
    }

    // other service methods
}
```

Domain Layer

```
public class Product {
    private String name;
    private double price;
    private int quantity;

    // getters and setters
    // other product methods
}

public class ShoppingCart {
    private List<Product> products;

    public void addProduct(Product product) {
        // add product to cart
    }

    // other shopping cart methods
}
```

Persistence Layer

```
public class ProductRepository {
    public List<Product> findAll() {
        // retrieve products from database
    }

    // other repository methods
}
```

Загальні принципи проєктування з використанням багатошарової архітектури:

- **Абстракція.**
- **Інкапсуляція.** Під час проектування немає необхідності робити які-небудь припущення про типи даних, методи і властивості або реалізацію, оскільки всі ці деталі приховані в рамках шару.
- **Чітке розділення функціональності між шарами.** Верхні шари, такі як Presentation Layer, посилають команди нижнім шарам, таким як Business Layer і Persistance Layer, і можуть реагувати на події, що виникають в цих шарах, забезпечуючи можливість передачі даних між шарами вгору і вниз.
- **Висока зв'язність.** Чітко означені межі відповідальності для кожного шару і гарантоване включення в шар тільки функціональності, безпосередньо пов'язаній з його завданнями, допоможе забезпечити максимальну зв'язність в рамках шару.
- **Можливість повторного використання.** Відсутність залежностей між нижніми і верхніми шарами забезпечує потенційну можливість їх повторного використання в інших сценаріях.
- **Слабке зчеплення.** Для забезпечення слабкого зчеплення між шарами зв'язок між ними грунтується на абстракції і подіях.

Separation of Concerns (розділення відповідальності) у багатошаровій архітектурі

- **Концепція розділення відповідальності** у багатошаровій архітектурі. Компоненти в межах певного шару обмежені, вони стосуються лише логіки, яка відноситься до цього шару.
- Наприклад, компоненти Presentation Layer обробляють лише логіку представлення, тоді як компоненти Business Layer обробляють лише бізнес-логіку. Це дозволяє розробникам використовувати свій конкретний технічний досвід, щоб зосередитися на технічних аспектах домену (таких як логіка представлення або логіка збереження). Компромісом цієї переваги, однак, є відсутність загальної гнучкості (здатності швидко реагувати на зміни).

Поділ за технічною роллю у багатошаровій архітектурі

- Багатошарова архітектура є технічно розділеною архітектурою (на відміну від архітектури з доменним поділом - domain-driven design).
- Групи компонентів групуються не за доменом, а за технічною роллю в архітектурі (наприклад, представлення чи бізнес). У результаті будь-яка конкретна бізнес-сфера поширюється на всі шари архітектури. Наприклад, домен «клієнт» міститься на шарі представлення, шарі бізнес-правил, шарі послуг і шарі бази даних, що ускладнює застосування змін до цього домену. Як наслідок, підхід до проектування, орієнтований на домен (domain-driven design), не так добре працює зі стилем багаторівневої архітектури.