# Лекція 2. Класифікація архітектурних парадигм (продовження)

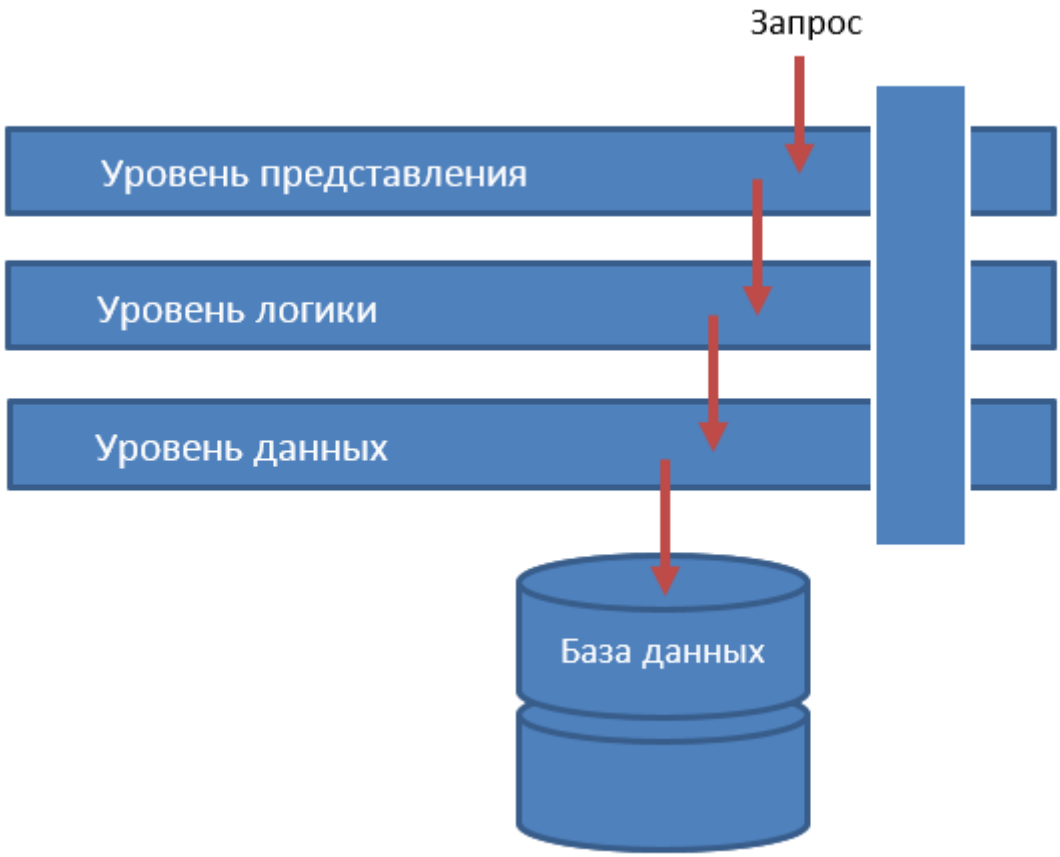
*Слайд 2.* **Зміст лекції 2**

2.1 Багаторівнева архітектура
2.2 Ключові ідеї.
2.3 Архітектура, керована подіями (EDA)
2.4 Мікроядерна архітектура

*Слайд 3.* ***2.1 Багаторівнева архітектура***

Є однією з найвідоміших архітектур, в якій кожен шар виконує певну функцію. Залежно від ваших потреб ви можете реалізувати будь-яку кількість рівнів, але занадто велика їх кількість призведе до надмірного ускладнення системи. Часто виділяють три основні рівні: рівень представлення, рівень логіки і рівень даних.

*Слайд 4.*




*Слайд 5.* Шару не обов'язково знати, що роблять його сусіди. Тут проявляється така властивість як поділ відповідальності. Якщо всі три шари є закритими, то запит користувача до верхнього рівня ініціює ланцюжок звернень з верхнього рівня до найнижчого. У цьому випадку рівень представлення відповідає за користувальницький інтерфейс і відображення даних для користувача і нічого не знає про існування фізичного сховища даних. Нічого про

існування бази даних не знає і рівень логіки - його «турбують» тільки правила бізнес-логіки. Доступ до бази даних має лише через рівень управління даними.

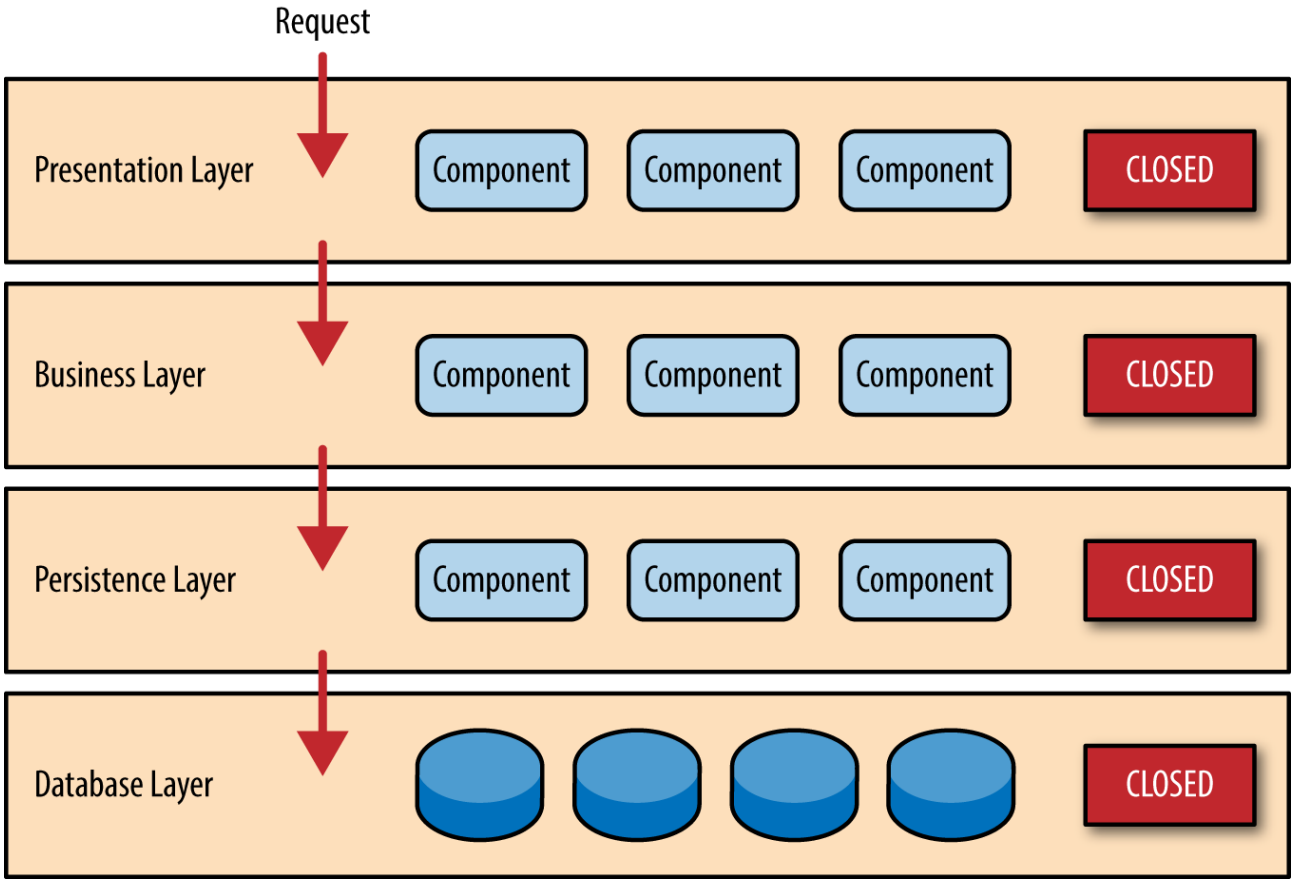
Перевагами застосування такої архітектури є простота розробки (в основному через те, що цей вид архітектури всім знаком) і простота тестування. Серед недоліків можна виділити можливі складнощі з продуктивністю і масштабуванням - всьому виною необхідність опрацювання запитів і даних по всіх рівнях (знову ж таки, в тому випадку, якщо всі верстви є закритими).

Компоненти в багаторівневій архітектурі організовані в горизонтальні шари, кожен шар яких виконує певну роль в додатку (наприклад, уявлення логіки або бізнес-логіки). Хоча багаторівнева архітектура моделі не визначає кількість і типи шарів, які повинні існувати в шаблоні, більшість багаторівневих архітектур складається з чотирьох стандартних шарів: уявлення, бізнес логіка, дані і бази даних. У деяких випадках, бізнес-шар і шар даних об'єднані в єдиний бізнес-шар, зокрема, коли логіка даних (наприклад, SQL або HSQL) вкладається в компоненти бізнес-рівня. Таким чином, невеликі додатки можуть мати тільки три шари, в той час як більш великі і складні бізнес-додатки можуть містити п'ять або більше шарів.

Кожен шар має певну роль і відповідальність у додатку. Наприклад, рівень уявлення відповідатиме за обробку всіх призначених для користувача інтерфейсів і браузер зв'язку з логікою, в той час як бізнес-шар буде нести відповідальність за виконання конкретних бізнес-правил, пов'язані із запитом. Кожен шар в архітектурі утворює абстракції навколо роботи, яку необхідно зробити, щоб задовольнити конкретний бізнес запит. Наприклад, рівнем подання не потрібно знати або турбуватися про те, як отримати дані клієнта; це потрібно тільки, щоб прибрати цю інформацію на екрані в певному форматі. Крім того, бізнес-шару не потрібно турбуватися про те, як форматувати дані клієнта для відображення на екрані або, це потрібно тільки, щоб отримати дані з шару збереження, виконання бізнес-логіки щодо даних (наприклад, обчислювати значення або агреговані дані), і передати цю інформацію до рівня представлення даних.

*Слайд 6.* **2.2 Ключові ідеї.**

Зверніть увагу, що кожен з шарів в архітектурі позначається як закритий. Це дуже важливе поняття в багаторівневій архітектурі. Замкнутий за чисельністю прошарок означає, що запит переміщається від шару до шару, він повинен пройти через шар прямо під ним, щоб дістатися до наступного шару нижче того. Наприклад, запит, що походять з шару подання повинен спочатку пройти через бізнес-шар, а потім на шар даних, перш ніж, нарешті, потрапити в шар бази даних.

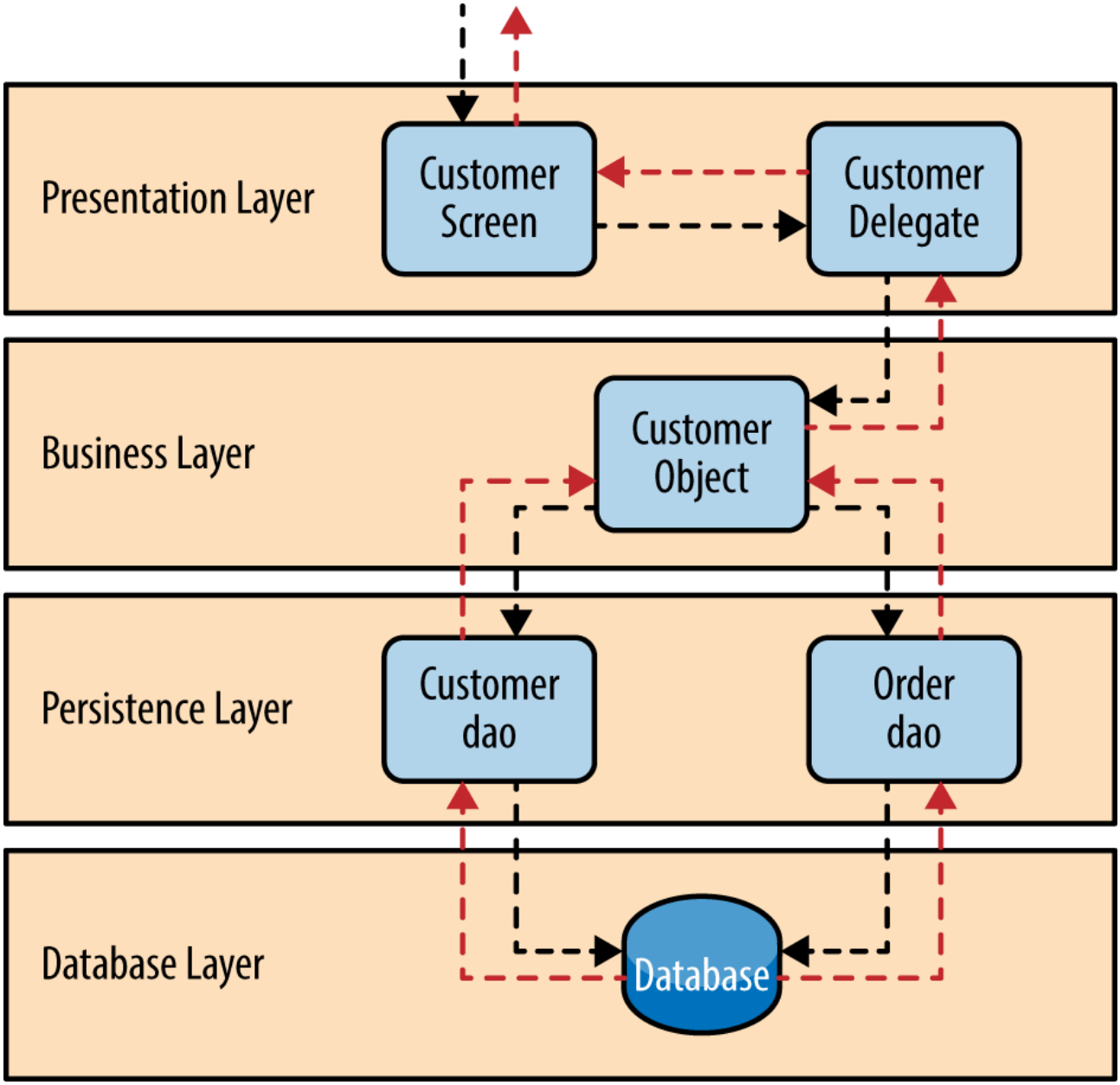



Концепція ізоляції Шарів означає, що зміни, зроблені в одному шарі архітектури, як правило, не впливають або впливають мінімально на компоненти в інших шарах: зміна ізольовані в компонентах всередині цього шару, і, можливо, на іншому пов'язаному Слока (наприклад, збереження шару, що містить SQL). Якщо ви дозволите шару уявлення прямий доступ до шару управління даними, то зміни, внесені в SQL при збереженні шару будуть впливати на бізнес-рівень і рівень представлення, тим самим створюючи дуже тісно пов'язані додатки з великою кількістю взаємозалежностей між компонентами. Цей тип архітектури, стає дуже важко і дорого змінити.

Концепція ізоляції Шарів також означає, що кожен шар не залежить від інших верств, тим самим маючи мало або взагалі не знають про внутрішню роботу інших верств в архітектурі.

*Слайд 7. Приклад.*

Щоб проілюструвати, як працює багаторівнева архітектура, розглянемо запит від бізнес-користувачів, для отримання інформації про клієнта для конкретної людини. Чорні стрілки показують запит, що входить в базу даних, щоб відновити дані про клієнтів, а червоні стрілки показують реакцію вихідну назад на екран, щоб відобразити дані. У цьому прикладі, інформація про клієнта складається з даних клієнта і даних замовлення (замовлення, розміщені замовником).

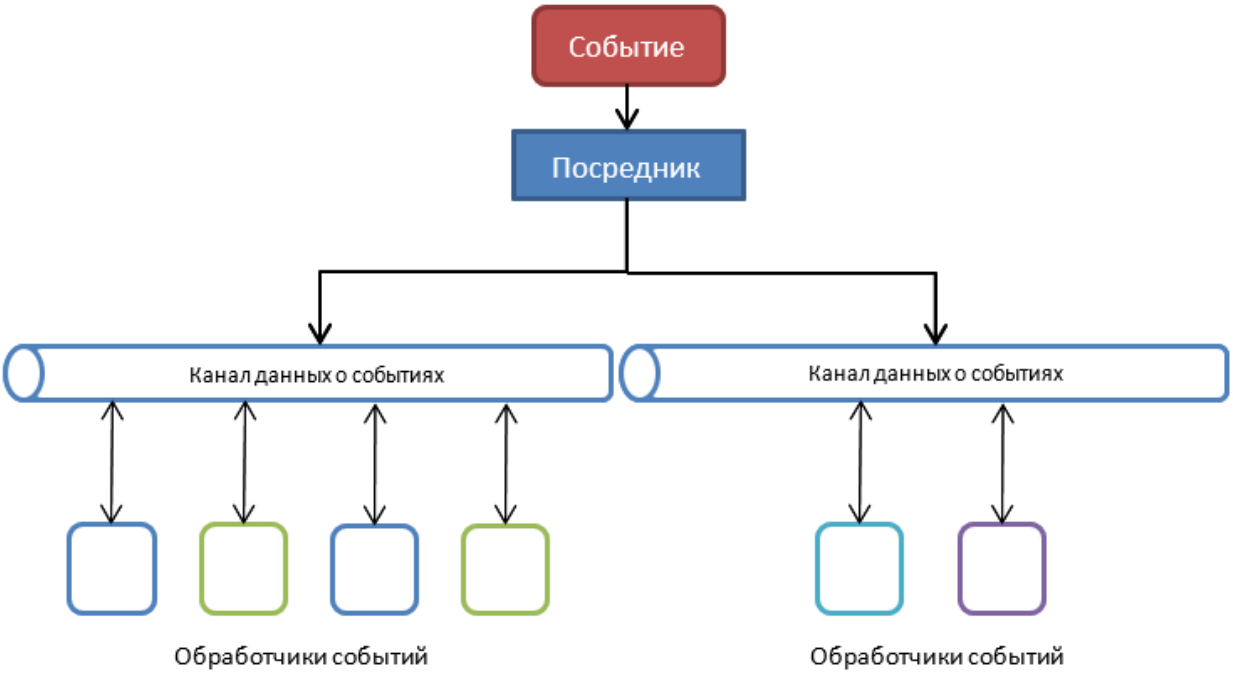



*Слайд 8.* Екран клієнт відповідає за прийом запиту і відображення інформації про клієнта. Він не знає, де ці дані, як вони витягуються, або скільки таблиць бази даних повинні бути в запиті, щоб отримати дані. Після того, як екран клієнт отримує запит на отримання інформації про клієнтів для конкретного індивідуума, він пересилає, що запитують дані на модуль делегата клієнта. Цей модуль відповідає за знання, які модулі в бізнес-шарі можуть обробити цей запит, а також, як дістатися до цього модуля і які дані йому потрібно переслати. Об'єкт клієнт в бізнес-шарі відповідає за агрегацію всієї інформації, необхідної по бізнес запитом (в даному випадку, щоб отримати інформацію про клієнта). Цей модуль звертається до модуля DAO клієнта (об'єкт доступу до даних) в шарі управління даних, щоб отримати дані про клієнтів, а також модуль DAO для того, щоб отримати інформацію про замовлення.

Ці модулі, в свою чергу виконувати оператори SQL, щоб отримати відповідні дані і передати його назад до об'єкта клієнт в бізнес-рівні. Після того, як об'єкт клієнт отримує дані, він агрегує дані і передає цю інформацію назад делегатом клієнта, який потім передає ці дані на екран клієнта, де вони будуть представлені користувачеві.

*Слайд 9.* **2.3 Архітектура, керована подіями (EDA)**

Це популярний адаптивний патерн, широко використовуваний для створення масштабованих систем.




*Слайд 10* Якщо говорити про програмне забезпечення, то в цій схемі існує два варіанти подій: инициализирующее подія і подія, на яке реагують обробники. Обробники є ізольованими незалежними компонентами, що відповідають (в ідеалі) за якусь одну задачу, і містять бізнес-логіку, необхідну для роботи.

**Посередник** може бути реалізований декількома способами. Самий просто спосіб - це скористатися фреймворками для інтеграції Apache Camel, Spring Integration або Mule ESB. Для великих додатків, яким потрібна більш складні функції управління, ви можете реалізувати посередника, використовуючи концепцію управління бізнес-процесами (наприклад движок jBPL).

**Архітектура, керована подіями** - це відносно складний патерн. Причиною тому - його розподілена і асинхронна природа. Вам доведеться вирішувати проблеми фрагментації мережі, обробляти помилки черзі подій і так далі. Плюсами цієї архітектури можуть служити висока продуктивність, легкість розгортки і вражаючі можливості масштабування. Однак можливо ускладнення процесу тестування системи.

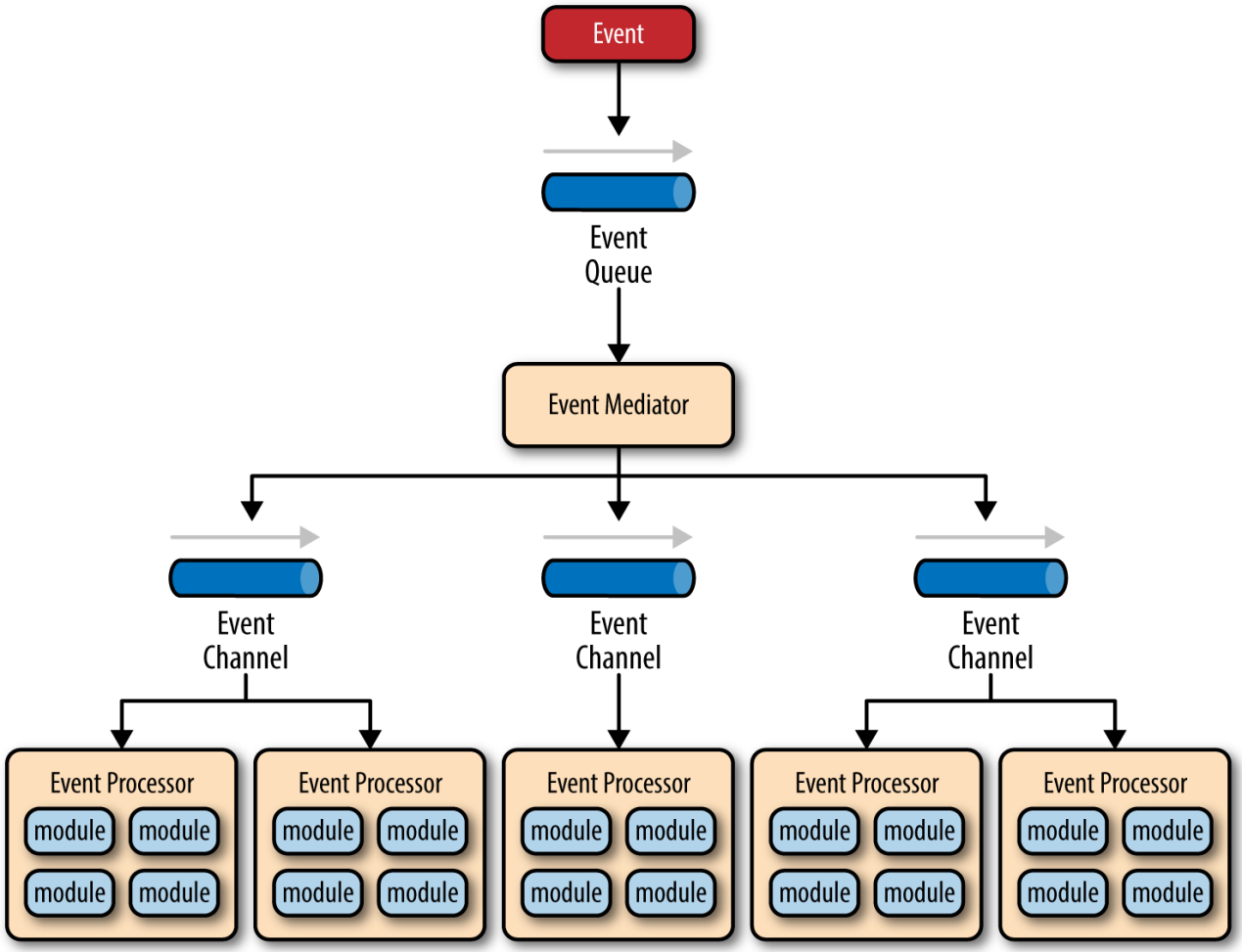
*Слайд 11.* EDA складається з двох основних топологій, посередник і брокер. Топологія посередник зазвичай використовується, коли вам потрібно організувати кілька кроків у події через центрального посередника, в той час як топологія брокера використовується, коли ви хочете організувати ланцюг подій разом без використання центрального посередника. Оскільки характеристики і стратегії здійснення розрізняються між цими двома топологій, важливо розуміти, кожен з них, щоб знати, який найкраще підходить для вашої конкретної ситуації.

*Слайд 12.* **Топологія Посередник**

Топологія посередник використовується для подій, які мають кілька кроків і вимагають деякого рівня узгодженості для обробки події. Наприклад, одна подія для розміщення біржової торгівлі може зажадати, щоб спочатку перевірили угоду, а потім перевірили відповідність цієї біржової торгівлі щодо дотримання різних правил, призначити торгівлю брокеру, порахувати комісію, і, нарешті, місце торгівлі з брокером. Всі ці кроки зажадають деякого рівня узгодженості, щоб визначити порядок кроків і які з них можна робити послідовно а які паралельно.

*Слайд 13* Є чотири основні типи компонентів архітектури в рамках топології Посередник: черга подій, посередник подій, канали подій і процесори подій. Потік подій починається з клієнта посилає подія в чергу подій, яка використовується для транспортування події до посередника подій. Посередник події отримує вихідне подія і погодить цю подію шляхом відправки додаткових асинхронних подій на канали подій, щоб виконати кожен крок процесу. Процесори подій, які очікують на каналах подій, отримавши подія від посередника подій і виконують певну бізнес логіку для обробки події.

*Слайд 14.*




*Слайд 15.* Звичайно є від десятків до декількох сотень черг подій. Шаблон не визначає реалізацію компонента черзі подій; це може бути чергу повідомлень, веб-сервіс, або будь-яка їх комбінації.

Посередник-подія може бути реалізований різними способами. Як архітектор, ви повинні розуміти, кожен з цих варіантів здійснення, щоб гарантувати, що рішення, яке ви вибираєте для посередника подій відповідає вашим потребам і вимогам.

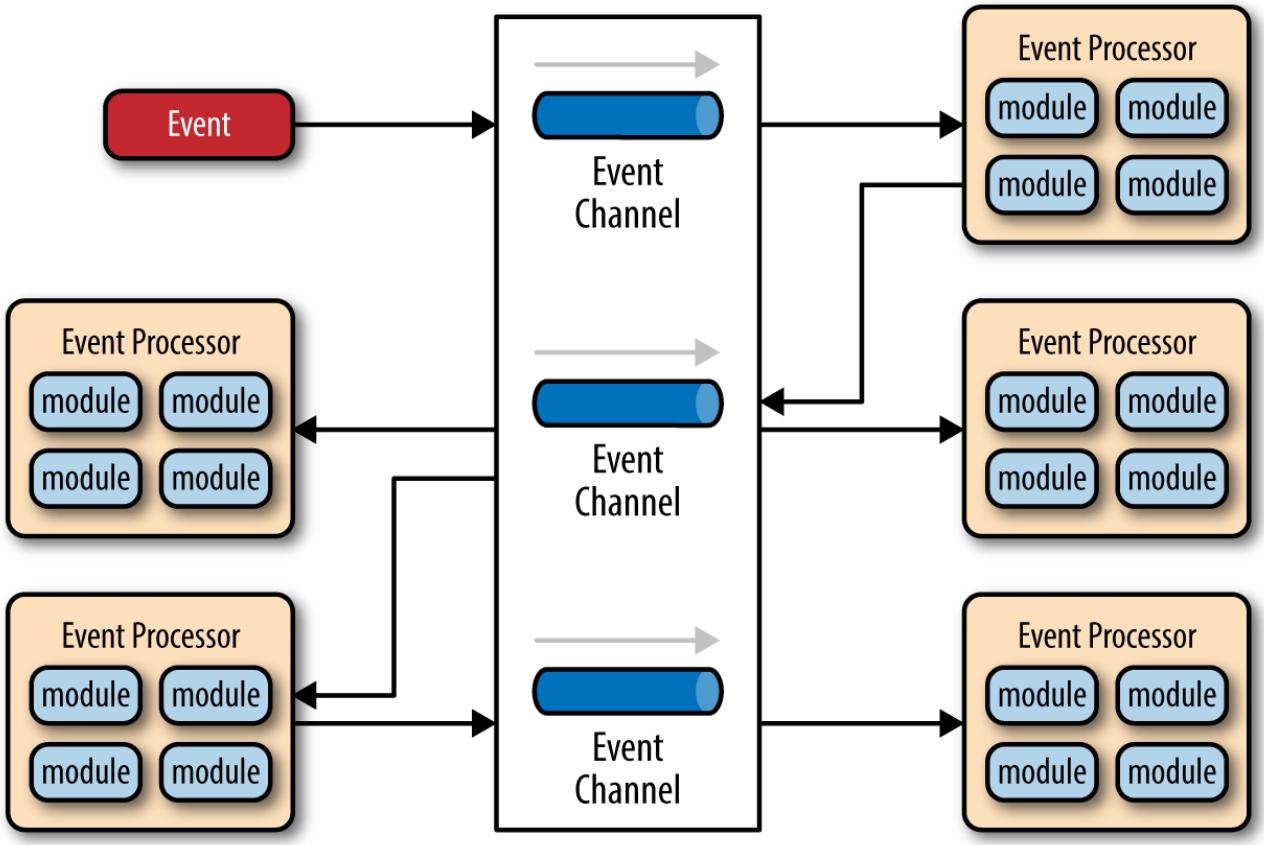
Найпростіший і найбільш поширеною реалізацією посередника подій є реалізація через вузли інтеграції з відкритим вихідним кодом, такі як Spring Integration, Apache Camel, або Mule ESB. Потоки подій в цих вузлах інтеграції з відкритим вихідним кодом, як правило, реалізується через Java-код або DSL (предметно-орієнтована мова). Для більш складного посередництва і регулювання, ви можете використовувати BPEL (бізнес-процес мовного виконання) в поєднанні з BPEL engine. BPEL є стандартний XML-подібна мова, який описує дані і кроки, необхідні для обробки вихідної події. Для дуже великих додатків, що вимагають набагато більш складного узгодження (включаючи кроки з участю людини), ви можете реалізувати посередник події, використовуючи менеджер бізнес-процесів (BPM), наприклад jBPM.

*Слайд 16.* **Топологія Брокер**

Топологія брокер відрізняється від топології посередника в тому, що немає ніякого центрального посередника подій; а потік повідомлень розподіляється між компонентами процесора подій в каналах через прості повідомлення брокера. Ця топологія корисна, якщо у вас є відносно простий потік обробки подій, і ви не хочете (або не потрібно) відповідно подій через єдиний центр.

Є два основних типи компонентів архітектури в рамках топології брокера: компонент брокера і компонент процесора подій. Компонент брокер може бути централізованим або федеративною і містить всі канали подій, які використовуються в потоці подій. Канали подій, що містяться в компоненті брокера можуть бути чергами повідомлень, темами повідомлень, або комбінацією обох.

*Слайд 17.*



Як ви можете бачити з діаграми, немає ніякого центрального компонента подій посередника управління і узгодження початкового події; швидше, кожен компонент процесор подій відповідає за обробку події та публікації нового події, вказуючи дію, яке він щойно виконав. Наприклад, процесор подій, який врівноважує портфель акцій може отримати початкове подія під назвою дроблення акцій. На підставі цього початкового події, процесор подій може зробити деякий зміна балансу портфеля, а потім опублікувати нову подію для брокера під назвою перебалансировка портфеля, який потім буде підібраний іншим процесором подій. Зверніть увагу, що бувають випадки, коли подія публікується процесором подій, але не підбирається будь-яким процесором іншої події. Це часто зустрічається, коли додаток еволюціонує або для забезпечення майбутньої функціональності і розширень.

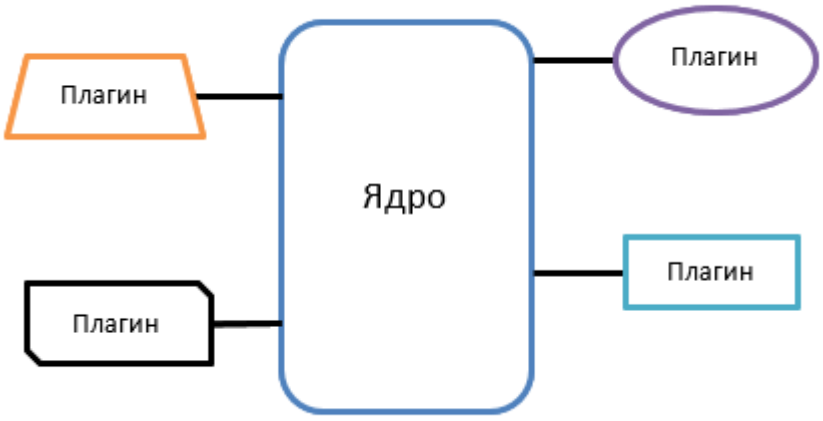
*Слайд 18.* ***Плюси мінуси***

Керована подіями моделі архітектура є відносно складною для реалізації моделлю, в першу чергу через її асинхронного розподіленого характеру. При реалізації цього шаблону, ви повинні вирішувати різні питання, розподіленої архітектури, такі як віддалений доступ процесу, відсутність чуйності і брокер зміни логіки відмови в разі брокера або посередника.

Однією зі складностей даного патерну, є відсутність атомарних операцій для одного бізнес-процесу. Оскільки компоненти процесора подій сильно розв'язані і розподілені, то дуже важко підтримувати транзакційних єдність роботи з ними. З цієї причини при розробці вашого застосування, використовуючи цей шаблон, ви повинні постійно думати про те, які події можуть і не можуть працювати незалежно один від одного і планувати деталізацію ваших процесорів подій відповідно.

Можливо, одним з найбільш складних аспектів подієвого шаблону архітектури є створення, підтримка і управління контрактами компонентів подій процесора. Кожна подія, як правило, має конкретну інформацію, пов'язану з ним (наприклад, значення даних і формат даних, що передаються по відношенню до процесора подій). Це життєво важливо при використанні цього шаблону вибрати стандартний формат даних (наприклад, XML, JSON, об'єкт Java і т.д.) і розробити політику договору управління версіями з самого початку.

*Слайд 19.* ***2.4 Мікроядерна архітектура***



Патерн складається з двох компонентів: основної системи (ядра) і плагінів. Ядро містить мінімум бізнес-логіки, але керує завантаженням, вивантаженням і запуском необхідних плагінів. Таким чином, плагіни виявляються непов'язаними між собою.

Оскільки плагіни можуть розроблятися незалежно один від одного, такі системи мають дуже високу гнучкість і, як наслідок, легко тестуються. Продуктивність додатки, побудованого на основі такої архітектури, безпосередньо залежить від кількості підключених і активних модулів.

Можливо найкращим прикладом микроядерной архітектури буде Eclipse IDE. Завантажуючи Eclipse без надбудов, ви отримуєте абсолютно порожній редактор. Однак з додаванням плагінів порожній редактор почне перетворюватися в корисний і легко настроюється продукт. Ще один хороший приклад - це браузер: додаткові плагіни дозволяють розширити його функціональність.

*Слайд 20.* Модулі являють собою автономні, незалежні компоненти, які містять спеціальну обробку, додаткові функції, а також призначений для користувача код, який призначений для посилення або розширення базової системи для отримання додаткових можливостей. Як правило, модулі повинні бути незалежними від інших модулів, але ви, звичайно, можна розробити плагіни, які вимагають присутності інших плагінів. У будь-якому випадку, важливо, щоб зв'язок між плагінами була зведена до мінімуму, щоб уникнути проблем, пов'язаних з залежностями.

Ядро системи має знати про те, які модулі доступні, і як дістатися до них. Одним з поширених способів реалізації цього через якийсь плагін в реєстрі. Цей реєстр містить інформацію про кожному модулі плагіна, в тому числі такі речі, як його ім'я, формат даних, і деталі протоколу віддаленого доступу (в залежності від того, як плагін підключений до базової системи).

Модулі можуть бути підключені до основної системи за допомогою різних способів. Сама архітектура не визначає будь-яку з цих деталей реалізації, тільки те, що модулі повинні залишатися незалежними один від одного.

*Слайд 21.* ***2.5 Мікросервісна архітектура***

У цьому випадку додаток розбивається на безліч невеликих сервісів, які називаються мікросервісамі. Кожен мікросервіс включає в себе бізнес-логіку і являє собою абсолютно незалежний компонент. Сервіси однієї системи можуть бути написані на різних мовах програмування і спілкуватися один з одним, використовуючи різні протоколи.

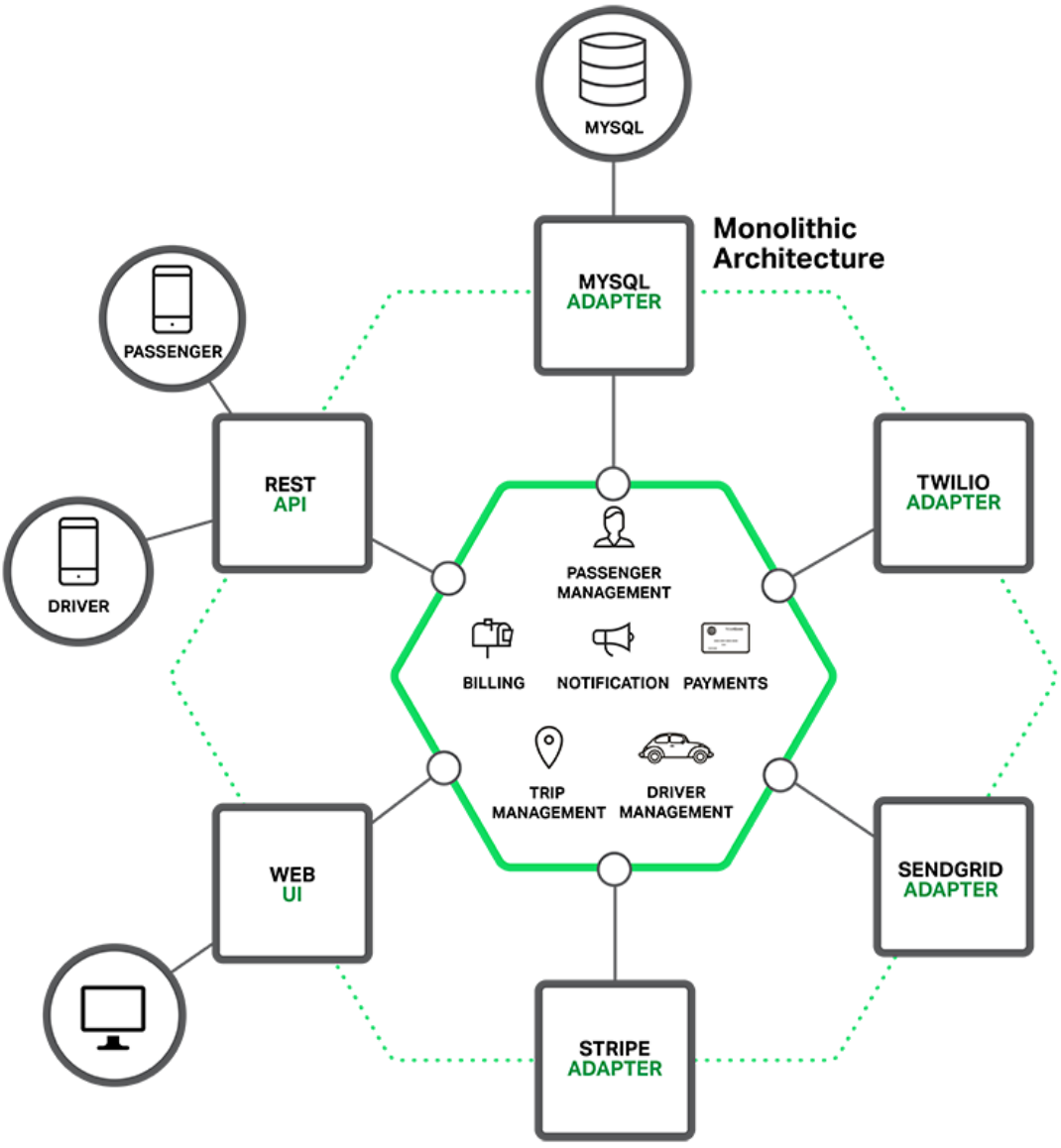
Оскільки кожен мікросервіс є окремим проектом, ви можете розподілити роботу над ними між командами розробників, тобто над системою можуть одночасно працювати кілька десятків програмістів. Мікросервісная архітектура дозволяє з легкістю масштабувати додаток - якщо вам знадобилося запровадити нову функцію (розгортати кожен мікросервіс можна окремо), просто напишіть новий сервіс, а якщо якийсь функцією ніхто не користується - відключіть сервіс.

Очевидним недоліком цього патерну є необхідність передачі великої кількості даних між мікросервісамі. Якщо накладні витрати на обмін повідомленнями занадто великі, потрібно або оптимізувати протокол, або об'єднати мікросервіси. З тестуванням таких систем теж не все просто. Наприклад, якщо ви пишете клас на Spring Boot, який повинен протестувати все

REST API сервісу, то він повинен запустити перевіряється мікросервіс і мікросервіси з ним пов'язані. Це не ядерна фізика, але недооцінювати складність процесу все ж не варто.

*Слайд 22. Приклад.*

Необхідно написати абсолютно новий додаток для виклику таксі, яке буде конкурувати я такими додатками як Uber і Hailo. Після кількох попередніх зустрічей та збору вимог, ви можете створити новий проект, або вручну, або за допомогою генератора, який поставляється з Rails, Spring Boot, Play, або Maven. Це новий додаток буде мати модульну архітектуру гексагональную:




*Слайд 23.* В основі програми є бізнес-логіка, яка реалізується за допомогою додаткових модулів, які визначають послуги, доменних об'єктів і подій. Навколо сердечника адаптери, які взаємодіють із зовнішнім світом. Приклади адаптерів включають в себе компоненти доступу до бази даних, компоненти обміну повідомленнями, які виробляють і споживають повідомлення, а також веб-компоненти, які або виставляють API, або реалізують користувальницький інтерфейс.

Незважаючи на наявність логічно модульну архітектуру, додаток запакований і розгортається як моноліт. Фактичний формат залежить від мови додатки і рамок.

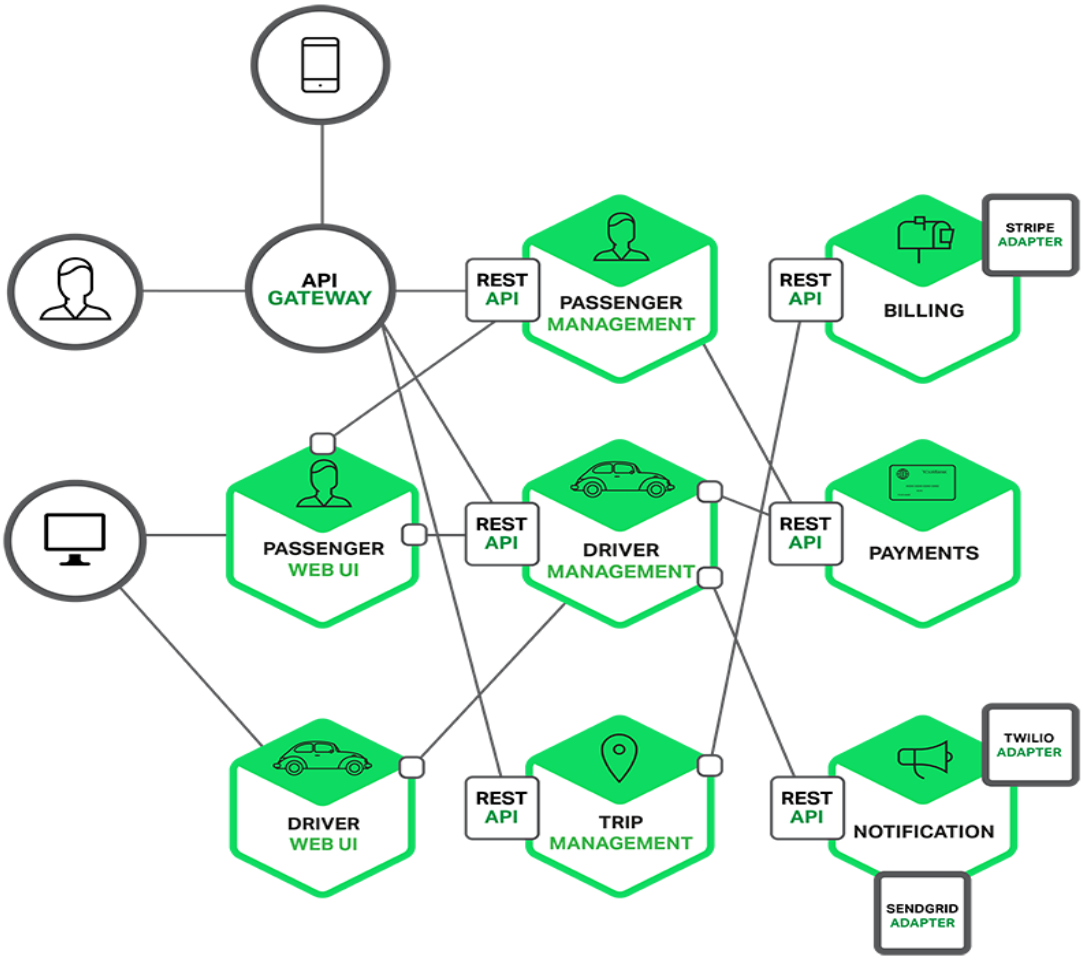
Програми, написані в цьому стилі надзвичайно поширені. Вони прості в розробці. Такі додатки також просто перевірити. Ви можете реалізувати від початку до кінця тестування просто запустити додаток і тестувати призначеного для користувача інтерфейсу з Selenium. Монолітні додатки також прості в розгортанні. Ви просто повинні скопіювати упаковане додаток на сервер. Ви можете також масштабувати додаток, запустивши декілька копій. На ранніх стадіях проекту він працює добре.

На жаль, цей простий підхід має величезне обмеження. Успішне застосування має звичку рости з плином часу і в кінці кінців стає величезним. Під час кожного запуску, ваша команда розробників реалізує ще кілька історій, які, звичайно ж, означає додавання рядків коду. Через кілька років, ваше маленьке, просте додаток переросте в жахливий моноліт.

Багато організацій, такі як Amazon, eBay і Netflix, вирішили цю проблему, прийнявши те, що зараз відомо як шаблон Мікросервісной архітектури. Замість того щоб будувати одне жахливе, монолітне додаток, ідея полягає в тому, щоб розбити додаток на безліч дрібніших, пов'язаних між собою сервісів.

*Слайд 24.* Сервіс, як правило, реалізує безліч різних функцій або функціональних можливостей, таких як управління замовленнями, управління клієнтами і т.д. Кожен мікросервіс є міні-додаток, яке має свою власну гексагональную архітектуру, що складається з бізнес-логіки, а також різних перехідників.

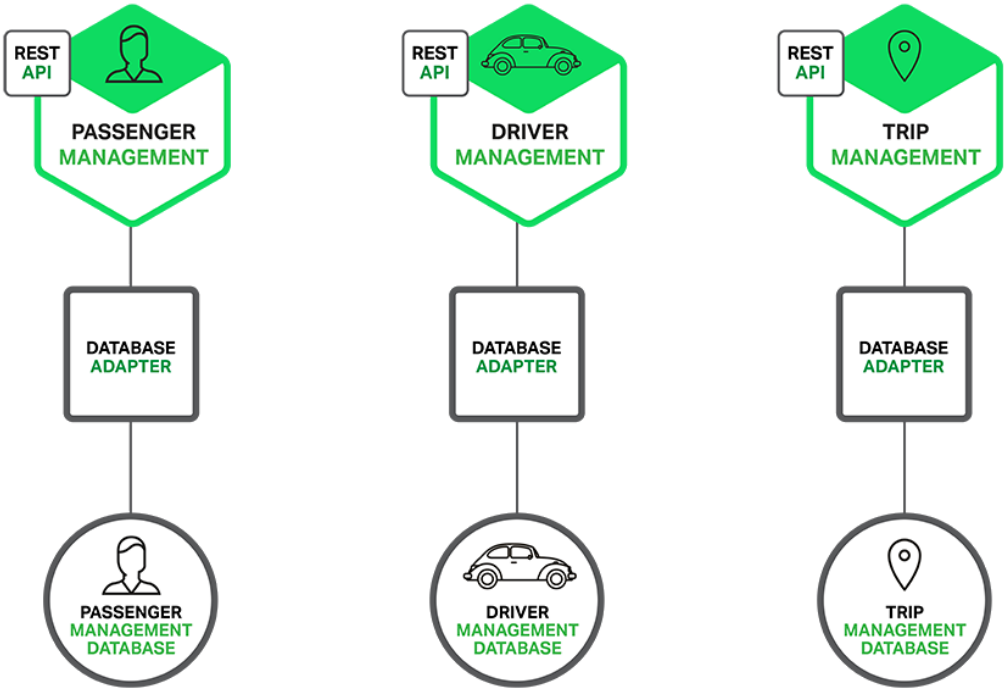
*Наприклад*, можлива архітектура системи, описаної раніше показано на наступній схемі:



*Слайд 25* Кожна функціональна сфера застосування в даний час здійснюється його власним мікросервісом. Крім того, веб-додаток розбивається на набір більш простих веб-додатків (наприклад, один для пасажирів і один для водіїв в нашому таксі-град, наприклад). Це полегшує розгортання особливий досвід для конкретних користувачів, пристроїв або спеціалізованих випадків використання.

Моделі мікросервісної Архітектури істотно впливає на взаємозв'язок між додатком і базою даних. Замість того, щоб ділити одну схему бази даних з іншими сервісами, кожен сервіс має свою власну схему бази даних. З одного боку, такий підхід суперечить ідеєю моделі даних в масштабах підприємства. Крім того, це часто призводить до дублювання деяких даних. Однак, окремі схеми бази даних для кожного сервісу мають важливе значення, якщо ви хочете, отримати вигоду з мікросервісної архітектури, оскільки це забезпечує слабкий зв'язок. На наступній діаграмі показана архітектура бази даних для прикладу програми.

*Слайд 26*




Кожен із сервісів має свою власну базу даних. Крім того, сервіс може використовувати тип бази даних, яка найкраще підходить для його потреб. Наприклад, сервіс водія, який знаходить пропозиції, близькі до потенційного пасажиру, необхідно використовувати базу даних, яка підтримує ефективні гео-запити.

Запитання до лекції 2

1. Багаторівнева архітектура як одна з найвідоміших архітектур.
2. Яким чином організовані шари (компоненти) в багаторівневій архітектурі?
3. Ключові ідеї багаторівневої архітектури.
4. В чому полягає концепція ізоляції шарів?

5. Проілюструйте на прикладі, як працює багаторівнева архітектура.
6. За що відповідає модуль Екрану клієнта?
7. Архітектура, керована подіями (EDA) Наведіть схему двох варіантів подій які ініціалізують події.
8. Охарактеризуйте основні типи компонентів архітектури в рамках топології Посередник
9. Шаблони: черги подій, черги повідомлень, веб-сервіси, або будь-які їх комбінації.
10. Найпростіші і найбільш поширені реалізації посередника подій.
11. Охарактеризуйте основні типи компонентів архітектури в рамках топології Брокер. Визначте плюси і мінуси цієї топології.
12. Особливості мікроядерної архітектури.
13. Особливості та бізнес-логіка, яка реалізується в мікросервісній архітектурі.