О. С. Яшина, Т. С. Пісклова

ПРОЄКТУВАННЯ ІНФОРМАЦІЙНИХ СИСТЕМ

2024

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
Національний аерокосмічний університет ім. М. Є. Жуковського
«Харківський авіаційний інститут»

О. С. Яшина, Т. С. Пісклова

ПРОЄКТУВАННЯ ІНФОРМАЦІЙНИХ СИСТЕМ

Навчальний посібник

Харків «ХАІ» 2024

УДК 004.9:004.415.2(075.8)
Я96

Рецензенти: д-р техн. наук, проф. І. П. Гамаюн,
д-р техн. наук, проф. М. В. Євланов

Яшина, О. С.

Я96 Проєктування інформаційних систем [Електронний ресурс] : навч. посіб. / О. С. Яшина, Т. С. Пісклова. – Харків : Нац. аерокосм. ун-т ім. М. Є. Жуковського «Харків. авіац. ін-т», 2024. – 68 с.

Розглянуто питання проєктування сучасних інформаційних систем та особливості трирівневої та мікросервісної архітектурних моделей. Наведено способи забезпечення якості, надійності та безпеки сучасних інформаційних систем. Описано основні архітектурні моделі, підходи та методи до проєктування архітектури програмного забезпечення.

Для студентів спеціальності 122 «Комп'ютерні науки» під час вивчення курсу «Проектування інформаційних систем» та спеціальності 126 «Інформаційні системи та технології» під час вивчення курсу «Створення інформаційних систем та технологій».

Іл. 18. Табл. 2. Бібліогр.: 20 назв

УДК 004.9:004.415.2(075.8)

ЗМІСТ

Передмова 5
1 Основи проєктування ІС..... 6
1.1 Поняття системи…… ……6
1.2 Принципи системного аналізу в проєктуванні ІС……6
2 Огляд методів проєктування ІС 7
2.1 Підходи до проєктування архітектури……7
2.2 Методи аналізу та проєктування інформаційних систем……8
3 Методи детального проєктування ІС..... 9
3.1 Загальна характеристика методів проєктування ІС……9
3.2 Підходи до проєктування програмного забезпечення……9
3.3 Формально-математичні методи в проєктуванні ІС……11
3.4 Методи формального опису ІС……12
4 Аналіз та веріфікація проєктних рішень..... 14
4.1 Експериментальна перевірка характеристик програмного забезпечення – тестування……14
4.2 Методи статичного аналізу……15
5 Моделі архітектури інформаційної системи..... 17
5.1 Мікроархітектура і макроархітектура……17
5.2 Платформні архітектури інформаційних систем……18
5.3 Централізовані архітектурні моделі……19
5.4 Розподілені архітектурні моделі……23
6 Основні принципи проєктування програмного забезпечення..... 26
6.1 Принципи SOLID……26
6.2 Принцип KISS – робіть речі простіше……29
6.3 Принцип DRY……30
6.4 Принцип YAGNI……31
7 Багатошарова архітектура програмного забезпечення..... 31
7.1 Загальна характеристика багатошарової архітектури……31
7.2 Загальні принципи проєктування багатошарової архітектури32
7.3 Логічний поділ на шари……34
7.4 Сервіси і шари……36
7.5 Розподіл шарів і компонентів……36
7.6 Визначення правил взаємодії між шарами……37
8 Мікросервісна архітектура..... 38
8.1 Види мікросерсвісних архітектур……38
8.2 Типова мікросервісна архітектура……39
8.3 Структура мікросервісів……39
8.4 Оброблення розподілених транзакцій у мікросервісній архітектурі……40
9 Архітектурний шаблон MVC..... 44
9.1 Загальна характеристика MVC……44
9.2 Різновиди MVC……45
9.3 Архітектурний MVC……48

10 Проєктування наскрізної функціональності…….....50
10.1 Питання, які потребують особливої уваги під час проєктування……...50
10.2 Проєктування стратегії керування винятками…….....51
10.3 Проєктування стратегії валідації введення і даних….....55
10.4 Протоколювання й інструментування.....57
11 Аутентифікація та авторизація в мікросервісних додатках…….58
11.1 Складові поділу прав доступу…….....58
11.2 Сувора аутентифікація……....59
11.3 Протоколи аутентифікації……62
11.4 Протокол OAuth 2.0……..62
11.5 Сценарій авторизації за протоколом OAuth 2.0……....64
Бібліографічний список……66

ПЕРЕДМОВА

Мета вивчення дисципліни «Проектування інформаційних систем» **–** надати студентам знання, уміння, навички, методичні прийоми та засоби, що необхідні для розроблення та створення інформаційних систем (ІС) різноманітного призначення.

Вивчення основ і принципів системного підходу під час створення інформаційних систем, зокрема здобуття знань про архітектурні моделі, інструментальні засоби і програмні платформи створення ІС, є завданням курсу.

Після вивчення навчальної дисципліни студент повинен:

знати*:*

– основні положення системного аналізу, які використовують під час проєктування складних систем;

– основні моделі архітектури інформаційної системи;

– основні інструментальні засоби проєктування складної системи;

– засоби і методи пошуку та ухвалення рішень під час проєктування інформаційних систем;

вміти:

– вибирати модель архітектури інформаційної системи;

– проєктувати розподіл функцій та організовувати взаємодію між рівнями інформаційної системи;

– формувати вхідні дані та визначати основні технічні характеристики;

– розробляти комп'ютерні програми з використанням сучасних технологій обміну даними.

1 ОСНОВИ ПРОЄКТУВАННЯ ІС

1.1 Поняття системи

Система – сукупність компонентів і зв'язків між ними, що має властивості єдності та цілісності. Система функціонує як єдине ціле і має властивості, яких немає в кожного компонента окремо і які проявляються лише в разі їх взаємодії. Такі властивості називаються системними ефектами.

Системні ефекти поділяють на такі:

– корисні – необхідні для вирішення основних завдань системи;
– шкідливі – різні дефекти, конфлікти, баги, «глюки» тощо;
– нейтральні.

Основним завданням проєктування є отримання системи, що має необхідні корисні ефекти, властивості, а також нейтралізація шкідливих ефектів, що виникають при цьому.

Проєктуванням системи вважають процес пошуку, верифікації та ухвалення технічних рішень, а також оформлення відповідної документації, необхідної для створення системи відповідно до заданих вимог. Будь-який процес проєктування має ітераційний характер, зумовлений тим, що створюються прототипи, у яких реалізуються вироблені проєктні рішення й оцінюється якість.

1.2 Принципи системного аналізу в проєктуванні ІС

Декомпозиція. Це поділ системи на окремі підсистеми, які поділяються на компоненти, і проєктування компонентів відповідно до поставлених їм завдань.

Під час проєктування складної системи вона поділяється на більш прості підсистеми, які так само поділяються на компоненти, модулі, елементи реалізують окремі функції. Підсистема являє собою порівняно самостійний продукт, здатний до автономного функціонування. Компоненти і модулі функціонують тільки в межах більшого програмного продукту, але можуть розроблятися незалежно один від одного. Створені незалежно компоненти повинні бути зібрані в єдину систему, тобто в процесі проєктування має бути виконано композицію компонентів, а після їх програмної реалізації проведено інтеграцію системи.

Стратифікація. Під час проєктування як системи, так і окремих її компонентів виділяються страти, які відповідають різним аспектам розгляду системи, а саме функціональна, алгоритмічна, математична, інформаційна, організаційна, технічна та ін. Стратифікація дає змогу ефективно організувати роботу фахівців різної кваліфікації.

Субоптимальність. У процесі проєктування окремих компонентів і ухвалення рішень щодо окремих страт повинні забезпечуватися загальносистемні вимоги й оптимізація окремих компонентів відповідно до загальносистемних критеріїв оптимальності. Значною складністю під час реалізації цього принципу є недостатність методик дезагрегації вимог, тобто вираження вимог до системи через характеристики компонентів нижніх рівнів.

2 ОГЛЯД МЕТОДІВ ПРОЄКТУВАННЯ ІС

2.1 Підходи до проєктування архітектури

Можна виділити завдання проєктування архітектури (високорівневе) та детальне (низькорівневе) проєктування.

Основним завданням проєктування архітектури є визначення основних складових частин або компонентів системи, їх функцій і способів взаємодії між ними і з навколишнім середовищем.

Виділяють три методологічних підходи:

1 Проєктування зверху вниз.
2 Проєктування знизу вгору.
3 Еволюційне проєктування.

Проєктування зверху вниз (спадне проєктування / проєктування від цілей до засобів). Цей метод відображає класичний підхід до проєктування, при якому найперше формулюється глобальна або генеральна мета системи, будується цільова і функціональна декомпозиція, на основі якої будується структурна декомпозиція системи. Переходячи за деревом декомпозиції зверху вниз, проєктувальники деталізують і уточнюють рішення, отримані на верхніх рівнях.

Перевагами проєктування зверху вниз є високий ступінь відповідності вимогам і цілям проєктування, а також висока ефективність продуктивності системи.

Недоліками цього підходу вважають значну тривалість і високу вартість розроблення.

Область застосування проєктування зверху вниз – розроблення складних систем із високим рівнем новизни.

Проєктування знизу вгору (висхідне проєктування, або проєктування від досягнутого). Типове рішення методу – повторне використання й адаптація наявних рішень, як архітектурних, так і програмних.

Архітектура системи будується на основі однієї з відомих типових моделей (клієнт-серверна, багаторівнева модель тощо). Основна увага приділяється проєктуванню підсистем і компонентів, при цьому широко застосовуються адаптація і рефакторинг наявних рішень.

Перевагами проєктування знизу вгору вважають зниження трудомісткості розроблення та підвищення надійності системи.

Недоліками цього методу є зниження ефективності системи, а також можливі проблеми під час адаптації компонентів.

Область застосування проєктування знизу вгору – модернізація наявних систем, розроблення нових систем на основі типових рішень.

Еволюційне проєктування (метод розширюваного ядра). Під час еволюційноого проєктування так само, як і під час проєктування знизу вгору, основна увага приділяється проєктуванню підсистем, а не системі в цілому. Немає чіткого формулювання кінцевої мети проєктування, а також єдиної команди розробників. Основою архітектури є ядро, яке надає інфраструктуру для підключення окремих компонентів і модулів. Система розвивається шляхом підключення компонентів, створених різними командами розробників незалежно один від одного.

Серед переваг еволюційного проєктування – гнучкість в урахуванні потреб користувачів і тривалий життєвий цикл системи.

Недоліком методу є складність обслуговування.

Область застосування еволюційного проєктування – системи хмарних обчислень, open source розроблення та системи для власних потреб.

2.2 Методи аналізу та проєктування інформаційних систем

Процес визначення архітектури, компонентів, інтерфейсів та інших характеристик системи або її компонентів називається проєктуванням. Проєктування є інженерною діяльністю в межах життєвого циклу системи, у якій належним чином аналізуються вимоги для створення опису внутрішньої структури програмного забезпечення (ПЗ), що є основою для конструювання програмного забезпечення.

Відповідно до стандарту SWEBOK (ISO/IEC TR 19759:2015) проєктування програмних систем можна розглядати як діяльність, результат якої складається з двох складових частин:

– архітектурний / високорівневий дизайн (software architectural design, top-level design) – опис високорівневої структури й організації компонентів системи;

– деталізована архітектура (software detailed design) – опис кожного компонента в тому обсязі, який необхідний для конструювання.

Проєктування програмного забезпечення SWEBOK об’єднує 15 галузей знань (Knowledge Areas, KAs):

1 Вимоги до програмного забезпечення (Software Requirements).

2 Проєктування програмного забезпечення (Software Design).

3 Побудова програмного забезпечення (Software Construction).

4 Тестування програмного забезпечення (Software Testing).

5 Обслуговування програмного забезпечення (Software Maintenance).

6 Керування конфігурацією програмного забезпечення (Software Configuration Management).

7 Керування інженерією програмного забезпечення (Software Engineering Management).

8 Процес інженерії програмного забезпечення (Software Engineering Process).

9 Моделі та методи інженерії програмного забезпечення (Software Engineering Models and Methods).

10 Якість програмного забезпечення (Software Quality).

11 Професійна практика з інженерії програмного забезпечення (Software Engineering Professional Practice).

12 Економіка інженерії програмного забезпечення (Software Engineering Economics).

13 Основи обчислювальної техніки (Computing Foundations).

14 Математичні основи (Mathematical Foundations).

15 Інженерні основи (Engineering Foundations).

3 МЕТОДИ ДЕТАЛЬНОГО ПРОЄКТУВАННЯ ІС

3.1 Загальна характеристика методів проєктування ІС

Методи проєктування інформаційних систем поділяють на два підходи: структурний і об'єктно-орієнтований. Для структурного підходу характерні розділене подання та моделювання структур даних і процедур роботи з ними. Для об'єктно-орієнтованого підходу характерне подання системи у вигляді об'єктів, що взаємодіють шляхом обміну повідомленнями. При цьому об'єкт являє собою сукупність даних (стан об'єкта) і процедур (методів) їх оброблення. Утім, деякі сучасні методи проєктування неможливо однозначно вважати структурним або об'єктно-орієнтованим підходами, тому що ці методи оперують поняттями класів і об'єктів, але основна увага приділяється міжоб'єктній і міжкомпонентній взаємодії, а не застосуванню інкапсуляції та поліморфізму під час проєктування класів.

3.2 Підходи до проєктування програмного забезпечення

Серед методів проєктування можна виділити кілька категорій.

Процедурно-орієнтоване проєктування (ПОП, Procedural-oriented design, POD) – зразок класичного підходу до проєктування ІС, основним завданням якого є ідентифікація функцій системи, їх детальне опрацювання та створення специфікацій, що забезпечують можливість реалізації окремих функцій.

Об'єктно-орієнтоване проєктування (ООП, Object Oriented Design, OOD) являє собою стратегію проєктування ПЗ з об'єктів. Об'єкт – це

сутність, яка здатна перебувати в різних станах і має безліч операцій. Стан визначається як набір атрибутів об'єкта. Операції, пов'язані з об'єктом, надають їх іншим об'єктам для виконання певних обчислень. Об'єкти об'єднуються в класи, кожен із яких має опис усіх атрибутів і операцій. ПЗ складається з об'єктів, що взаємодіють, мають локальний стан і здатні виконувати набір операцій, який визначається станом об'єкта. Об'єкти інкапсулюють інформацію про їх стан і обмежують до них доступ. ООП використовує можливості структурного підходу, наприклад декомпозиція в діаграмах використання і станів, а також діаграми потоків даних широко використовуються під час логічного і фізичного проєктування бази даних (БД). На стадії проєктування вимог в ООП простежується зв'язок між діаграмами «сутність – зв'язок» і діаграмами класів. У процесі проєктування переважне значення має моделювання предметної області в термінах об'єктів, що взаємодіють.

Функціонально-орієнтоване розроблення (ФОР, Feature Driven Development, FDD) основане на моделі предметної області, що дає змогу виявити ключові функції та властивості системи, що відображають сценарії взаємодії системи з користувачем. Досвідчені розробники формують мінікоманди, що реалізують окремі функції системи. Процес розроблення є ітеративним. ФОР, на відміну від процедурного підходу, оснований на гнучкому (Agile) підході до розроблення і не потребує детального попереднього проєктування. Однак саме ФОР, на відміну від інших Agile-технологій, передбачає широке використання моделей і формальних засобів проєктування.

Проєктування на основі даних (ПОД, Data-Structure-Centered Design, DSCD) передбачає, що основою для проєктування системи має бути модель потоків даних, що описує основні структури даних, їх джерела й характеристики. Визначаються метадані, що являють собою абстрактні описи застосовуваних структур і форматів даних. Потім проєктувальники описують входи і виходи кожного процесу, а також самі процеси перетворення даних і керування потоками даних.

Проєктування на основі тестів (ПОТ, Test Driven Design, TDD) основане на тестах, що демонструють окремі сценарії роботи системи. По суті, вимоги до системи специфікують за допомогою набору тестів, кожен із яких подає окремий прецедент (test case) як коректного, так і помилкового використання системи. Розробники додають функціонал, що забезпечує успішне виконання тестів. Зазвичай тести виконуються в автоматичному режимі. Цей підхід популярний в Agile-методології розроблення інформаційних систем. Упереджувальне розроблення тестів прискорює настроювання та полегшує подальший супровід системи, однак може збільшити трудомісткість розроблення.

Компонентне проєктування (КП, Component-Based Design, CBD). Програмні компоненти є незалежними одиницями, які мають однозначно-визначені інтерфейси і залежності, можуть збиратися й розгортатися

незалежно один від одного. Цей підхід покликаний вирішити завдання використання, розроблення й інтеграції таких компонентів із метою підвищення повторного використання активів (як архітектурних, так і у формі коду). При цьому компонент надходить найчастіше у вигляді відкомпільованого коду і для клієнтської системи відіграє роль «чорного ящика», що ускладнює його адаптацію для конкретних вимог.

Сервіс-орієнтоване проєктування (СОП, Service-oriented design, SOD) – це метод проєктування систем на основі сервіс-орієнтованої архітектури (Service-Oriented Architecture, SOA). SOA передбачає безліч слабопозв’язаних сервісів (служб), які об'єднані загальним комунікаційним механізмом. Кожен із сервісів займається вирішенням тільки одного завдання. Така концепція дає змогу використовувати сервіси безліч разів у різних додатках. Так само забезпечується легка розширюваність системи, оскільки сервіси не впливають один на інший, а додавання нових сервісів не стосується вже наявних. У класичній моделі SOA шина сервісів (Enterprise Service Bus, ESB) реалізує концепцію єдиної точки доступу до додатка. Більш сучасним способом побудови сервіс-орієнтованих додатків є мікросервісна архітектура, яка передбачає ще більшу ізоляцію сервісів.

Аналіз наведених вище підходів до проєктування ІС дає змогу зробити висновок про те, що наразі немає універсальної методології проєктування, що можна застосовувати під час розроблення систем будь-якого призначення. Колись універсальна структурна методологія втратила своє значення з огляду на старіння як теоретичної бази, так і інструментальних засобів структурного проєктування. Об'єктно-орієнтована методологія, що претендує на універсальність, найбільш корисна під час розроблення нижніх рівнів декомпозиції ІС (проєктування окремих компонентів і модулів) і малопридатна в процесі розроблення верхніх архітектурних рівнів. Решта методологічних підходів вирішують обмежене коло завдань проєктування окремих видів ІС.

Тому на практиці доцільно не робити однозначний вибір на користь однієї певної методології, а застосовувати комплексну методологію, основану на спільному застосуванні декількох підходів.

3.3 Формально-математичні методи в проєктуванні ІС

Програмування та математика мають тісний зв'язок, проте зараз немає методів синтезу ІС або хоча б її програмного забезпечення, основаних на застосуванні формально-математичних моделей. Математичне моделювання широко застосовується для вирішення допоміжних завдань проєктування, таких як оцінка ефективності, продуктивності, надійності системи, розрахунок її економічних характеристик тощо.

Спроби розроблення математичних моделей для формального опису архітектури системи й аналізу її властивостей продовжуються. Зазвичай у них використовується теоретико-множинний підхід.

Більшість наявних формальних методів і моделей проєктування ІС, однак, орієнтовані не на розроблення проєктних рішень, а на їх формальний опис, найчастіше за допомогою будь-якої графічної нотації.

3.4 Методи формального опису ІС

Нотація є угодою про подання моделі. Часто нотацію розуміють як візуальне (графічне) подання. Деякі нотації використовують текстові описи і, по суті, є формальними мовами. Нотація може визначатися:

– стандартом, наприклад OMG UML – Unified Modeling Language, що розвивається консорціумом OMG (Object Management Group);

– загальноприйнятою практикою, наприклад, у eXtreme Programming часто використовуються картки функціональної відповідальності та зв'язків класу – Class Responsibility Collaborator або CRC Card (CRC є текстовою, тобто невізуальною нотацією);

– внутрішнім методом проєктної команди.

Певні нотації використовуються на стадії концептуального проєктування, низка нотацій орієнтована на створення детального дизайну, багато з них може використовуватися на обох стадіях. Крім того, нотації найчастіше використовують у контексті застосовуваної методології або підходу. Розглянемо нотації з огляду на опис структурного (статичного) або поведінкового (динамічного) подання.

Структурний опис, статичне подання (Structural Descriptions, static view) описують структурні аспекти програмного дизайну, найчастіше стосуються основних компонент і зв'язків між ними.

Мови опису архітектури (Architecture description languages, ADLs) – текстові мови, часто формальні, використовувані для опису програмної архітектури в термінах компонентів і конекторів (спеціалізованих компонентів, що забезпечують взаємозв'язок функціональних компонентів між собою та із «зовнішнім світом»).

Діаграми класів і об'єктів (Class and object diagrams) використовуються для подання набору класів і статичних зв'язків між ними (наприклад, успадкування).

Діаграми компонентів / компонентні діаграми (Component diagrams) певною мірою схожі на діаграми класів, проте через специфіку концепції або поняття компонента зазвичай видаються в іншій візуальній формі. Компонент розглядається як фізично реалізований елемент програмного забезпечення, що має самодостатню логіку й реалізований як конгломерат інтерфейсу і його реалізації (часто у вигляді комплексу класів).

Картки функціональної відповідальності та зв'язків класу (Class responsibility collaborator card, CRC) використовуються для позначення

імені класу, його відповідальності (що клас має робити) та інших сутностей (класів, компонентів, акторів / ролей тощо), з якими пов'язаний клас; часто їх називають картками «клас-обов'язок-кооперація».

Діаграми розгортання (Deployment diagrams) використовуються для подання фізичних вузлів, зв'язків між ними і моделювання інших фізичних аспектів системи.

Діаграми «сутність-зв'язок» (Entity-relationship diagram, ERD / ER) використовуються для подання концептуальної моделі даних, що зберігаються в процесі роботи інформаційної системи.

Мови опису / визначення інтерфейсу (Interface Description Languages, IDLs) – мови, подібні до мов програмування, але не передбачають можливостей опису логіки системи і призначені для визначення інтерфейсів програмних компонентів (імен і типів, що експортуються або публікованих операцій).

Структурні діаграми Джексона (Jackson structure diagrams) використовуються для опису структур даних у термінах послідовності, вибору й ітерацій (повторень).

*Структурні схеми (*Structure charts) описують структуру викликів у програмах (який модуль викликає, ким і як викликається).

Поведінковий опис, динамічне подання (Behavioral Descriptions, dynamic view) використовуються для опису динамічної поведінки програмних систем та їх компонентів.

Діаграми діяльності / операцій (Activity diagrams), використовуються для опису потоків робіт і керування.

Діаграми співробітництва (Collaboration diagrams) показують динамічну взаємодію, що відбувається в групі об'єктів і приділяють особливу увагу об'єктам, зв'язкам між ними і повідомленням, якими обмінюються об'єкти за допомогою цих зв'язків.

Діаграми потоків даних (Data flow diagrams, DFDs) описують потоки даних усередині набору процесів (не в термінах процесів операційного середовища, але з огляду на обмін інформацією в бізнес-контексті).

Таблиці та діаграми ухвалення рішень (Decision tables and diagrams) використовуються для подання складних комбінацій умов і дій (операцій).

Блок-схеми та структуровані блок-схеми (Flowcharts and structured flowcharts) – застосовуються для подання потоків керування (контролю) та пов'язаних операцій.

Діаграми послідовності (Sequence diagrams) використовуються для показу взаємодій всередині групи об'єктів з акцентом на часовій послідовності повідомлень / викликів.

Діаграми переходу і карти станів (State transition and statechart diagrams) застосовуються для опису потоків керування переходами між станами.

Формальні мови специфікації (Formal specification languages) – текстові мови, що використовують основні поняття з математики

(наприклад, множини) для суворого й абстрактного визначення інтерфейсів і поведінки програмних компонентів, часто в термінах перед- і постумов.

Псевдокод і програмні мови проєктування (Pseudocode and program design languages, PDLs) – мови, які використовуються для опису поведінки процедур і методів, переважно на стадії детального проєктування; подібні до структурних мов програмування.

Таким чином, формально-математичні методи і моделі застосовуються в процесі проєктування ІС для вирішення різноманітних завдань опису й аналізу проєктних рішень.

4 АНАЛІЗ ТА ВЕРІФІКАЦІЯ ПРОЄКТНИХ РІШЕНЬ

4.1 Експериментальна перевірка характеристик програмного забезпечення – тестування

Тестування застосовується для вивчення реальних властивостей програмного забезпечення (ПЗ) шляхом проведення серії експериментів. Об'єктом тестування є не сама програма, а її поведінка, що спостерігається.

Модульне тестування – перевірка коректності окремих фрагментів програмного коду. Виконується в автономному режимі. Існує безліч видів тестування, що вирішують різні завдання:

1 Тестування цілісності – перевірка коректності взаємодії компонентів.

2 Тестування після інтеграції в системі – перевірка коректності виконання окремих функцій.

3 Тестування бізнес-циклів – перевірка коректності спільного виконання комплексу функцій, які являють собою бізнес-завдання користувачів. Необхідно для виявлення спотворення даних та інших побічних ефектів, що виникають під час взаємодії різних функцій.

4 Навантажувальне і стресове тестування, мета якого – вивчення поведінки системи за умови підвищених навантажень, пов'язаних з обслуговуванням великої кількості користувачів, високою частотою запитів або великими обсягами інформації, що надходить. Під час навантажувального тестування вивчається поведінка системи за умови тривалих штатних або короткочасних пікових навантажень, а під час стресового – деградація властивостей системи за умови вкрай високих навантажень. Вивчається також коректність засобів відновлення системи. Обидва ці види тестування виконуються в автоматизованому режимі.

5 Тестування інтерфейсу користувача, завдання якого – вивчення коректності призначеного для користувача інтерфейсу і його зручності (usability). Основна проблема при цьому – суб'єктивність користувачів. Під час дослідження ПЗ, призначеного для масового використання, може

застосовуватися тестування в лабораторії, що дає змогу об'єктивно оцінити використання окремих елементів, призначених для користувача інтерфейсу, а так само реєструвати ознаки втоми або роздратування користувачів. Однак лабораторне тестування досить дороге.

Тестування є основним методом дослідження ПЗ і його верифікації, тобто підтвердження його коректності та відповідності специфікації. Основними недоліками тестування є висока трудомісткість, що значно перевищує трудомісткість розроблення вихідного коду, зокрема, неможливість застосування на ранніх стадіях розроблення.

4.2 Методи статичного аналізу

Статичний аналіз являє собою дослідження ПЗ без його запуску. На відміну від тестування, об'єктом дослідження є не поведінка програми, що спостерігається, а її код і супроводжувана документація. Це дає змогу виявити деякі види дефектів, що не виявляються під час тестування.

Існує низка усталених донині заходів обміну знаннями, що проводяться в софтверних ІТ-компаніях. Такі заходи є як технічними (націлені на поліпшення коду), так і навчальними (дають змогу обмінюватися знаннями). Розглянемо деякі з них.

Аналіз проєкту (Design review). Спільне обговорення розробниками проєктних рішень, пов'язаних з архітектурою системи, організацією взаємодії компонентів, алгоритмами вирішення окремих завдань, а так само застосовуваними технологіями й інструментальними засобами. Аналіз проєкту схожий на «мозковий штурм», але його метою є не пошук нових рішень, а аналіз запропонованих. Застосовується на всіх стадіях проєкту. Крім вирішення технічних завдань, аналіз проєкту сприяє поліпшенню клімату в колективі.

Перегляд коду (Code review). Цей метод ще називають інспекцією або ревізією коду. Його сенс полягає в регулярному перегляді й обговоренні написаного коду. На обговоренні присутній автор коду й інші програмісти, які виступають у ролі рецензентів. Така процедура, крім виявлення помилок і усунення неоптимальних рішень, дає змогу обмінюватися досвідом, вдалими рішеннями і підходами, а також приводить код проєкту до загальних стандартів. Якщо проєкт великий і програмісти спеціалізуються на окремих його частинах, перегляд і обговорення чужого коду дає змогу краще зрозуміти логіку й особливості роботи системи.

Під час перегляду коду учасники процесу висловлюють думки щодо доцільності тих чи інших рішень, пропонують свої та отримують зворотний зв'язок від колег. Очевидно, що таке обговорення має бути добре організованим і спочатку модерується, щоб критика подавалася конструктивно й доброзичливо. Пізніше, коли захід стане частиною

корпоративної культури, люди звикнуть до нього і знайдуть прийнятні форми критики, від модерування можна відмовитися.

Різні варіанти інспекції коду виключають публічне обговорення або роботу в парі: автор – рецензент. Такий варіант знижує стрес для автора, але, відповідно, зменшує і навчальний ефект: інші програмісти компанії не можуть учитися на чужих помилках, а сам автор отримує неповний спектр думок. Найчастіше для перегляду коду використовуються інструменти систем контролю версій.

«Парне програмування» – техніка програмування, при якій весь вихідний код створюється парами людей, що програмують одну задачу, сидячи за одним робочим місцем. Один програміст («пілот») керує комп'ютером і переважно думає про кодування в деталях. Інший програміст («штурман») зосереджений на картині загалом і безперервно переглядає код, розроблений першим програмістом. Час від часу вони міняються ролями, зазвичай кожні півгодини.

Така взаємодія дає змогу не тільки підвищити якість коду, але й організувати неформальний обмін знаннями. У разі значної різниці в досвіді та кваліфікації учасників процесу ця методика трансформується в наставництво.

Застосування цієї техніки може дати ще один важливий результат – спільне володіння кодом. Це поняття означає, що зона відповідальності програміста поширюється на код, написаний іншими. Така практика привчає вдосконалювати чужий код: виправляти помилки, покращувати й оптимізувати його. Згодом імовірність того, що програміст виправить чужий код, навіть не працюючи в певний момент у парі з його автором, значно підвищується.

Необов'язково працювати в парі весь час, таку діяльність можна чергувати з традиційним одиночним програмуванням. Також періодично корисно змінювати склад пар, щоб в ідеалі кожен програміст попрацював з усіма іншими учасниками проєкту. Такий підхід може значно посилити робочі та міжособистісні відносини, сприяти так званому спрацьовуванню команди.

Порівняння характеристик видів інспекцій коду наведено в таблиці 4.1.

Таблиця 4.1 – Порівняння видів інспекцій коду

Вид інспекції	Головна думка	Переваги	Недоліки
Інспекція «за плечем»	За автором спостерігають інші члени команди під час написання коду	Можна спостерігати за процесом розроблення і перебігом думок автора коду	Необхідно погоджувати час для спільної роботи, інші програмісти простоюють

Кінець таблиці 4.1

Вид інспекції	Головна думка	Переваги	Недоліки
Обмін кодом електронною поштою	Автор коду розсилає свої зміни розробникам, які так само надсилають свої зауваження після перегляду змін	Немає необхідності погоджувати час, код може дивитися дуже велика кількість людей	Складно обговорювати питання стосовно коду
Парне програмування	Два розробники програмують разом на одній фізичній машині	Можна спостерігати за процесом розроблення і перебігом думок, на процес може впливати напарник	Триває довше ніж одного розробника, не завжди є можливість використовувати віддалено
Використання спеціальних засобів публікації коду (систем контролю версій)	Автор публікує код на спеціальний ресурс і надає доступ усім зацікавленим особам	Спеціальне програмне забезпечення для інспекції коду дає змогу не тільки дивитися на зміни, але часто зберігає обговорення в доступній для сприйняття формі, а також забезпечує навігацію між знайденими помилками	Не можна спостерігати за процесом розроблення і перебігом думок автора

5 МОДЕЛІ АРХІТЕКТУРИ ІНФОРМАЦІЙНОЇ СИСТЕМИ

5.1 Мікроархітектура і макроархітектура

У технічному аспекті можна розглядати архітектуру як високорівневу абстрактну модель, у якій немає подробиць реалізації.

Архітектуру інформаційної системи можна описати як концепцію, визначальну модель, структуру, виконувані функції і взаємозв'язок компонентів інформаційної системи.

Терміни «мікроархітектура» і «макроархітектура» більшою мірою застосовуються для опису програмних систем. Відповідно до розглянутої

моделі рівнів архітектур корпоративних інформаційних систем микроархітектуру можна вважати рівнем програмної архітектури й архітектури даних, а макроархітектуру – рівнем IT-архітектури.

Мікроархітектура описує внутрішню структуру конкретного компонента або підсистеми, а макроархітектура – структуру всієї ІС як сукупності її компонентів або підсистем.

Існують два принципи, що дають змогу оцінити взаємний вплив компонентів системи один на одного:

1 Low coupling (слабка зв'язність).

2 High cohesion (сильна пов'язаність).

Принцип *Low coupling* сприяє розподіленню функцій між компонентами системи таким чином, щоб ступінь зв'язності між ними залишався низьким.

Ступінь зв'язності (coupling) – це *міра взаємозалежності* підсистем. Цей принцип пов'язаний з одним з основних принципів системного підходу, який потребує мінімізації інформаційних потоків між підсистемами.

Підсистема з низьким ступенем зв'язності має такі властивості: невелика кількість залежностей між підсистемами; слабка залежність однієї підсистеми від змін в інший; високий ступінь повторного використання підсистем.

Принцип *High cohesion* задає властивість сильної пов'язаності всередині підсистеми. У результаті підсистеми виходять сфальцьованими, керованими і зрозумілими.

Пов'язаність (функціональна пов'язаність) – це *міра пов'язаності* та сфокусованості функцій підсистеми. Підсистема має високий ступінь пов'язаності, якщо її функції тісно пов'язані між собою і якщо вона не виконує великих обсягів роботи.

Підсистема з низьким ступенем зв'язності виконує безліч різних функцій, жодним чином не пов'язаних між собою. Такі підсистеми створювати небажано, оскільки це може призвести до таких проблем: труднощі розуміння, складність під час повторного використання, складність підтримки, ненадійність, постійно зазнає змін.

Підсистеми з низьким ступенем зв'язності не мають чіткого функціонального призначення та мають занадто різнопланові функції, які можна легко розподілити між іншими підсистемами.

Слід зауважити, що зв'язність є характеристикою системи цілком, а пов'язаність характеризує окремо взяту підсистему.

5.2 Платформні архітектури інформаційних систем

Архітектурний підхід до проєктування інформаційних систем можна вважати найбільш довершеним. Його ключовим аспектом є створення фреймворку, тобто каркаса, адаптація якого до потреб конкретної системи буде легко здійсненна. Відповідно до цього завдання проєктування

поділяється на два підзавдання: розроблення багаторазово використовуваного каркаса і створення системи на його основі. Слід зазначити, що такі підзавдання можуть вирішуватися різними групами фахівців. У разі використання каркасів з'являється можливість досить швидко змінювати функціональність системи завдяки ітеративному процесу проєктування.

Можна виділити три напрями розвитку платформних архітектур:

1 Автономні.

2 Централізовані.

3 Розподілені.

Автономна архітектура передбачає наявність усіх функціональних компонентів системи на одному фізичному пристрої, наприклад комп'ютері, і не повинна мати зв'язків із зовнішнім середовищем. Прикладом таких систем можуть бути системні утиліти, текстові редактори і досить прості корпоративні програми. Слід зазначити, що в процесі побудування корпоративної інформаційної системи зазвичай не має формуватися незв'язаних вузлів або модулів, поява яких може бути зумовлена певними вимогами до безпеки або надійності.

5.3 Централізовані архітектурні моделі

Забезпечують виконання всіх необхідних завдань на спеціально відведеному вузлі, потужності якого достатньо, щоб задовольнити потреби всіх користувачів. Такий тип архітектури був популярний на момент появи комп'ютерної техніки (70-ті роки XX століття), однак і зараз залишається затребуваним. Компоненти системи в цьому випадку розподіляються між обчислювальним вузлом, котрий називається сервером (мейнфреймом), і термінальною станцією, з якою працює користувач. Термінал містить компоненти подання, а сервер – прикладні компоненти і компоненти керування ресурсами. Слід зазначити, що термінал використовується лише як пристрій введення-виведення і не має інших функціональних можливостей.

Перевагами такої архітектури можна вважати:

– відсутність необхідності адміністрування робочих місць;

– простоту обслуговування й експлуатації системи, оскільки всі ресурси зосереджені в одному місці.

Недоліками такої архітектури є:

– функціонування всієї системи цілком залежить від головного вузла (сервера);

– усі ресурси і програмні засоби є колективними і не можуть бути змінені для потреб конкретних користувачів.

Щоб позбутися від останнього недоліку, у сучасних інформаційних системах застосовуються технології віртуалізації, завдяки яким стає

можливим виділити кожному користувачеві необхідну кількість ресурсів і встановити необхідне програмне забезпечення.

Існують такі види централізованих архітектур:

– архітектура «файл-сервер»;
– архітектура «клієнт-сервер»;
– архітектура багаторівневих вебдодатків.

Файл-серверна архітектура передбачає наявність виділеного мережевого ресурсу для зберігання даних. Такий ресурс називається файловим сервером. При такій архітектурі всі функціональні компоненти системи містяться на комп'ютері користувача, який називається клієнтом, а самі дані – на сервері.

Така організація системи має переваги: багато користувачів у режимі роботи з даними, що зберігаються на сервері; централізоване керування правами доступу до загальних даних; низька ціна розроблення; висока швидкість розроблення.

Недоліки файл-серверної архітектури: послідовний доступ до загальних даних і відсутність гарантії їх цілісності; продуктивність (залежить від характеристик мережі, клієнта і сервера).

Класичне подання файл-серверної архітектури зображено на рисунку 5.1.

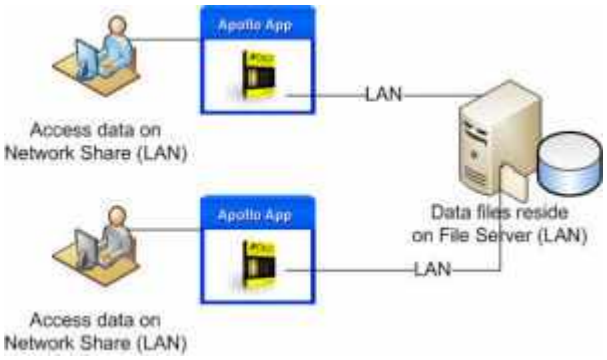

Рисунок 5.1 – Файл-серверна архітектура

Сервер є сховищем інформації, організованим у вигляді файлового масиву. Усе оброблення здійснюється на клієнтських комп'ютерах. Недоліком файл-серверної архітектури також є високе навантаження на мережу.

Застосування файл-серверної архітектури доцільне в системах оброблення слабоструктурованої інформації та у файлообмінних системах.

Архітектура «клієнт-сервер» – мережева інфраструктура, у якій сервери є постачальниками певних сервісів (послуг), а клієнтські комп'ютери – їх споживачами. Класичне подання клієнт-серверної архітектури передбачає наявність у мережі сервера і кількох під’єднаних

до нього клієнтів. У таких системах сервер в основному відіграє роль постачальника послуг із використання бази даних.

Сервер зберігає дані, організовані у вигляді реляційної моделі даних, і виконує запити від користувача. Здійснюється підготовка запитів, оброблення отриманих даних, їх візуалізація. Недоліком архітектури «клієнт-сервер» є необхідність обробляти інформацію з боку клієнта, а її перевагами – зниження навантаження на всі ланки передання даних та раціональний розподіл обчислювального навантаження між сервером і клієнтом.

Застосовуватися архітектура «клієнт-сервер» може великою кількістю користувачів, а також корпоративними системами.

Ця архітектурна модель називається дволанковою (two-tier architecture). Дволанкову архітектуру зображено на рисунку 5.2.

Переваги дволанкової архітектури – це підтримка багатокористувацької роботи, гарантія цілісності даних, наявність механізмів керування правами доступу до ресурсів сервера, а також можливість розподілу функцій між вузлами мережі.

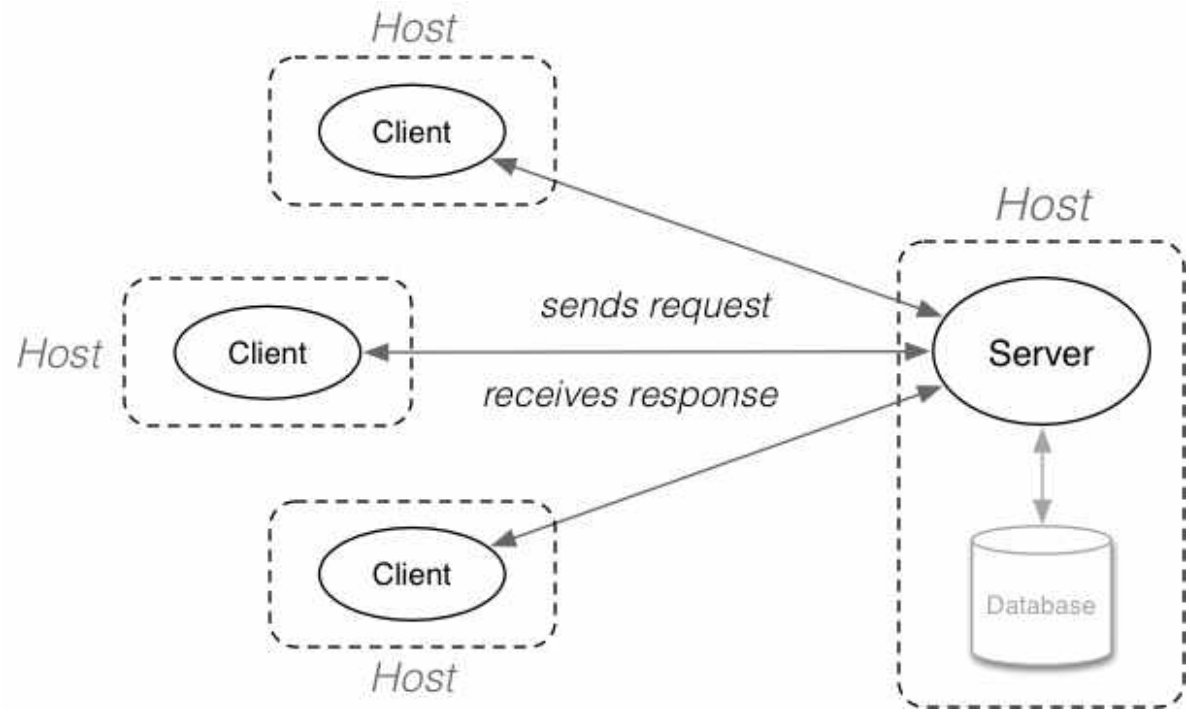

Рисунок 5.2 – Дволанкова клієнт-серверна архітектура

Недоліками цієї архітектури вважають непрацездатність усієї системи в разі виходу з ладу сервера, необхідність високого рівня технічного персоналу та висока вартість обладнання.

Багаторівнева архітектура. У разі збільшення масштабів системи виникає необхідність синхронізації версій великої кількості додатків. Для розв’язання цієї проблеми використовують багаторівневі та багатоланкові

архітектури (три і більше рівнів). Частина загальних додатків переноситься на спеціально виділений сервер, тим самим знижуються вимоги до продуктивності клієнтських машин. Клієнтів з низькою обчислювальною потужністю називають «тонкими клієнтами», а з високою потужністю – «товстими клієнтами». При багатоланковій архітектурі з виділеним сервером додатків існує можливість використання портативних пристроїв. Багатоланкову архітектуру показано на рисунку 5.3.

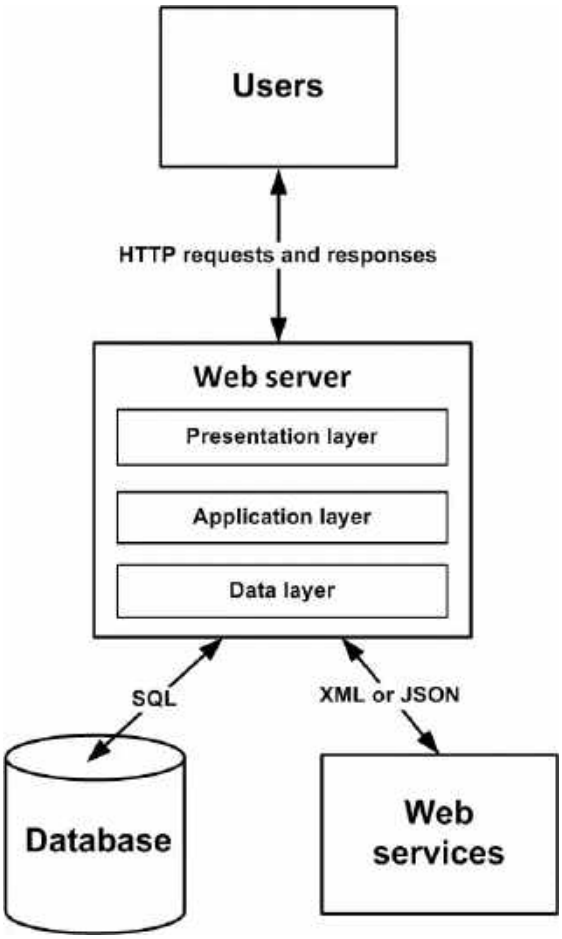

Рисунок 5.3 – Багатоланкова клієнт-серверна архітектура

Використання такого типу архітектури обумовлено високими вимогами додатка до ресурсів. У такому випадку доцільно розмістити його на окремому сервері і цим знизити вимоги до продуктивності робочих станцій.

Правильний добір характеристик сервера додатків, сервера баз даних і клієнтських робочих станцій дасть змогу створити інформаційну систему з прийнятною вартістю. Слід зауважити, що розподіл функціональних компонентів системи при використанні клієнт-серверної архітектури може здійснюватися кількома способами.

5.4 Розподілені архітектурні моделі

Поява і розвиток розподілених архітектур пов'язані з інтенсивним розвитком технічних і програмних засобів. У цьому типі архітектури функціональні компоненти інформаційної системи розподіляються за наявними вузлами залежно від поставлених цілей і завдань.

Можна виділити шість основних характеристик архітектури розподілених систем:

– спільне використання ресурсів (як апаратних, так і програмних);

– відкритість – можливість збільшення типів і кількості ресурсів;

– паралельність – можливість виконання декількох процесів на різних вузлах системи (при цьому вони можуть взаємодіяти);

– масштабованість – можливість додавати нові властивості та методи;

– відмовостійкість – здатність системи підтримувати часткову функціональність завдяки можливості дублювання інформації, апаратної та програмної складових.

Недоліками розподілених систем слід вважати такі характеристики: структурна складність; складно забезпечити достатній рівень безпеки; на керування системою потрібно багато зусиль; непередбачувана реакція на зміни.

Модель Peer-to-Peer, P2P (точка-точка). Система складається з однорангових вузлів, кожен із яких виконує функції як клієнта, так і сервера. Усі вузли взаємодіють один з одним, утворюючи зв'язну топологію. Проблемою моделі є те, що пошук вузла надає потрібні ресурси.

Способами розв’язання проблеми є використання спеціальних протоколів, що дадуть змогу вузлам обмінюватися інформацією один з одним, або перехід до гібридної архітектури із центральним вузлом (брокер ресурсів), у якому ведеться реєстр, який не є сервером і не надає жодних ресурсів.

Перевагами P2P є можливість використання обчислювальних та інформаційних ресурсів при раціональному розподілі навантаження між вузлами системи. Наявні також недоліки – складність архітектури та нестабільність характеристик. Застосовується модель у торентах, системах зв'язку тощо.

Сервіс-орієнтована та мікросервісна архітектура. Система складається з безлічі вузлів, що надають послуги або сервіси. Це дає змогу будувати слабкозв’язні системи, у яких окремі функції виконуються різними вузлами. Вузли взаємодіють між собою безпосередньо, проте для встановлення зв'язку між ними використовується брокер, на якому ведеться реєстр сервісів.

Перевагами сервіс-орієнтованої та мікросервісної архітектури є гнучкість (можливість конфігурувати систему з різних сервісів), надійність

(у разі виходу з ладу окремих сервіс-провайдерів система втрачає частину функцій, але не перестає працювати), швидкість розроблення (використання наявних сервісів замість створення нових компонентів).

Сервіс-орієнтована архітектура (service-oriented architecture, SOA) являє собою підхід до реалізації інформаційних систем, у яких основну увагу приділено створенню і використанню служб (service). SOA є переважно інтеграційною архітектурою, що надає безліч механізмів для гнучкої інтеграції як елементів однієї системи, так і різних інформаційних систем (рисунок 5.4). Принцип функціонування полягає у виклику одними додатками сервісів, що входять до складу інших додатків. Служби (сервіси) можуть бути об'єднані для створення бізнес-процесів, які будуть у подальшому використовувати як сервіси.

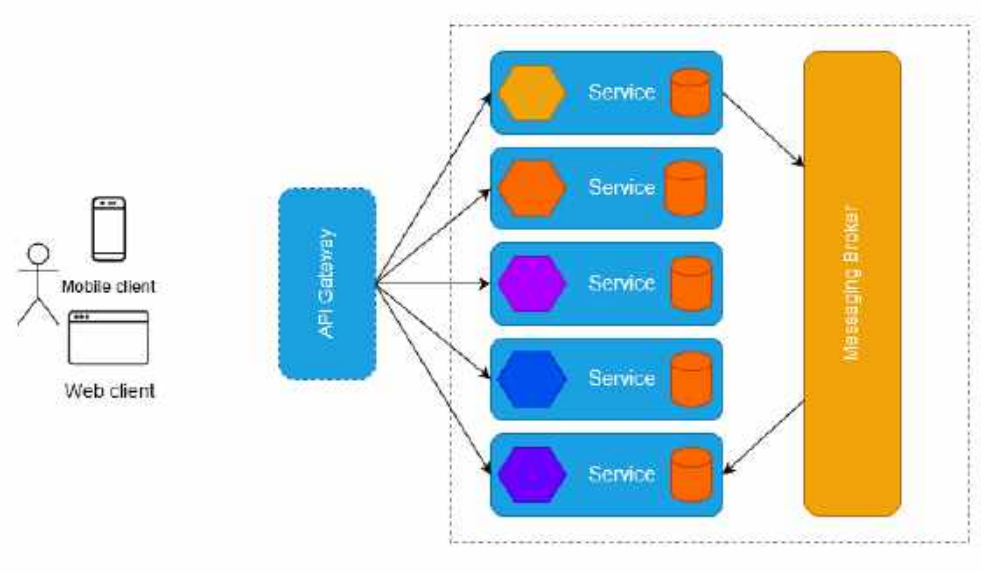

Рисунок 5.4 – Типова структура мікросервісної системи

SOA дає змогу вирішувати такі завдання:

– зменшення термінів упровадження нових елементів систем;
– швидке створення нових систем на основі наявних рішень;
– зменшення вартості інформаційної системи;
– зменшення вартості інтеграції;
– збільшення терміну життя системи;
– реалізація бізнес-процесів абстраговано від додатків і платформ.

Служби, що входять до складу SOA, мають відповідати таким властивостям:

– являти собою багаторазово використовувані бізнес-функції;
– визначатися за допомогою формальних інтерфейсів;
– мати протоколи зв'язку, незалежні від мови і платформи реалізації, і забезпечувати доступність інформації про місцезнаходження.

Кожен сервіс розташовується в певному місці і віддалено викликається клієнтами. Отже, якщо треба внести зміни, то їх потрібно робити в одному місці.

Основна концепція SOA полягає в описі сервісу незалежно від наявних протоколів. Служба визначається один раз і повинна мати кілька реалізацій для роботи з різними протоколами.

GRID дає змогу використовувати ресурси домашніх і офісних комп'ютерів, що простоюють, для вирішення складних обчислювальних завдань (рисунок 5.5). Pool ресурсів формується з великої кількості персональних комп'ютерів (ПК). TaskManager отримує завдання від клієнтів, поділяє їх на невеликі пакетні завдання і розподіляє між комп'ютерами, що простоюють. Після отримання результатів здійснює їх складання і відправлення клієнту. Сукупна обчислювальна потужність великої системи GRID порівнянна з характеристиками суперкомп'ютера.

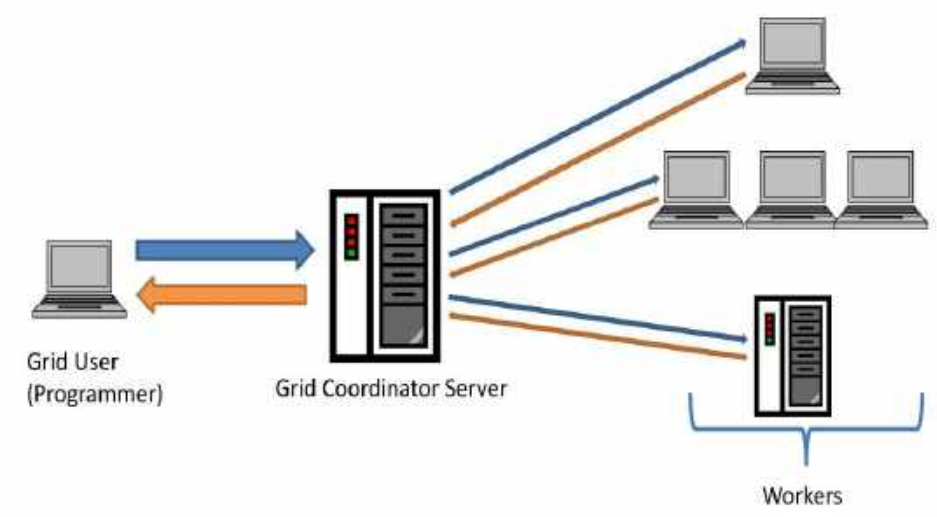

Рисунок 5.5 – Функціонування системи на основі архітектури GRID

Можливість задіяти значну обчислювальну потужність за умови порівняно невеликих додаткових витрат і раціональне використання недовантажених комп'ютерів є перевагами GRID.

Недоліками цієї системи вважають нестабільність характеристик і, як наслідок, складне планування завантаження комп'ютерів, а також невизначений час вирішення окремих завдань.

Використовується GRID у науковиих проєктах, що потребують великих обсягів розрахунків, але не критичні щодо тривалості їх виконання.

Хмарні архітектури. Клієнти працюють із віртуальним сервером, який приховує від них реальну обчислювальну архітектуру. Це дає змогу відмовитися від використання власного парку серверів і замінити їх орендою необхідних потужностей. У межах хмарної архітектури пропонується три основні концепції: SaaS – використання орендованого ПЗ; PaaS – використання для розроблення і запуску програм, орендованих

засобів розроблення, зокрема СУБД; IaaS – оренда віртуального комп'ютера, що має певні характеристики дискового простору, оперативної пам'яті тощо.

Перевагами хмарних архітектур є економічна вигода (оренда сервера вигідніша, ніж його купівля) та можливість гнучко розпоряджатися орендованими ресурсами.

Неможливість контролю користувачем фізичного стану наданої йому інфраструктури та законодавче обмеження на використання конфіденційної інформації вважають недоліками хмарних архітектур.

Загалом розподілені архітектурні моделі мають на багато ширші можливості, ніж централізовані, однак вони набагато складніші як у побудові, так і в обслуговуванні. Тому більшість наявних інформаційних систем побудовані на основі ПЗ централізованих моделей архітектури.

6 ОСНОВНІ ПРИНЦИПИ ПРОЄКТУВАННЯ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

6.1 Принципи SOLID

Починаючи роботу з архітектурою додатка, необхідно пам'ятати про основні принципи проєктування. Це допоможе створити архітектуру, яка буде відповідати перевіреним підходам, забезпечить мінімізацію витрат, простоту обслуговування, зручність використання і розширюваність. Розглянемо основні принципи.

SOLID – це п’ять принципів об'єктно-орієнтованого програмування, що описують архітектуру програмного забезпечення. Абревіатура SOLID розшифровується як:

– Single Responsibility Principle (принцип єдиної відповідальності);
– Open Closed Principle (принцип відкритості / закритості);
– Liskov's Substitution Principle (принцип підстановки Барбари Лісков);
– Interface Segregation Principle (принцип поділу інтерфейсу);
– Dependency Inversion Principle (принцип інверсії залежностей).

Принцип єдиної відповідальності (Single Responsibility Principle, SRP). Відповідно до цього принципу ніколи не має бути більше однієї причини змінити клас. На кожен об'єкт покладається один обов'язок, повністю інкапсульований у клас. Усі сервіси класу спрямовані на забезпечення цього обов'язку.

Це не означає, що ваші класи повинні містити тільки один метод або властивість. Може бути багато членів, якщо вони належать до єдиної відповідальності.

Поширені випадки порушення принципу SRP. Нерідко принцип єдиної відповідальності порушується в разі змішування в одному класі функціональності різних рівнів. Наприклад, клас виконує обчислення і подає їх користувачеві, тобто поєднує в собі бізнес-логіку і роботу з

призначеним для користувача інтерфейсом. Або клас керує збереженням / отриманням даних і виконанням над ними обчислень, що також небажано. Клас слід застосовувати тільки для одного завдання – або бізнес-логіка, або обчислення, або робота з даними.

Інший поширений випадок – наявність у класі або його методах абсолютно непов'язаного між собою функціонала.

Принцип відкритості / закритості (Open Closed Principle, OCP). Цей принцип змістовно описують так: програмні сутності (класи, модулі, функції тощо) повинні бути відкриті для розширення, але закриті для зміни. Це означає, що має бути можливість змінювати зовнішню поведінку класу і не вносити фізичні зміни в сам клас. Дотримуючись цього принципу, класи розробляють так, щоб для пристосування класу до конкретних умов застосування було досить розширити його і перевизначити деякі функції. Тому система має бути гнучкою, з можливістю роботи в змінних умовах без зміни вихідного коду.

Суть цього принципу полягає в тому, що систему має бути побудовано таким чином, що всі її подальші зміни мають бути реалізовані за допомогою додавання нового коду, а не зміни вже наявного.

«Закрито для модифікації» означає, що вже розроблено клас, що пройшов модульне тестування. Не треба змінювати клас, поки не буде знайдено помилки.

Принцип підстановки Барбари Лісков (Liskov's Substitution Principle, LSP). Це варіація принципу відкритості / закритості, про який зазначалося раніше. Його можна описати так: об'єкти в програмі можна замінити їх спадкоємцями без зміни властивостей програми. Це означає, що клас, розроблений шляхом розширення на підставі базового класу, повинен перевизначати його методи так, щоб не порушувалася функціональність з погляду клієнта. Інакше кажучи, якщо розробник розширює ваш клас і використовує його в додатку, він не повинен змінювати очікувану поведінку перевизначених методів. Підкласи мають перевизначати методи базового класу так, щоб не порушувалася функціональність з погляду клієнта.

Початкове визначення цього принципу, яке було дано Барбарою Лісков у 1988 році, мало такий вигляд:

Якщо для кожного об'єкта o1 типу S існує об'єкт o2 типу T, такий, що для будь-якої програми P, визначеної в термінах T, поведінка P не змінюється при заміні o2 на o1, то S є підтипом T.

Інакше кажучи, клас S може вважатися підкласом T, якщо заміна об'єктів T на об'єкти S не приведе до зміни роботи програми.

Загалом цей принцип можна сформулювати так:

Має бути можливість замість базового типу підставити будь-який його підтип.

Принцип підстановки Барбари Лісков фактично допомагає чіткіше сформулювати ієрархію класів, визначити функціонал для базових і похідних класів та уникнути можливих проблем у разі застосування поліморфізму.

Принцип поділу інтерфейсу (Interface Segregation Principle, ISP). Характеризується таким твердженням: клієнти не мусять реалізовувати методи, які вони не будуть використовувати. Інакше кажучи, клієнти не мусять залежати від методів, якими не користуються.

Порушуючи цей принцип, клієнт, який використовує певний інтерфейс з усіма його методами, залежить від методів, якими не користується, і тому виявляється сприйнятливий до змін у цих методах. У підсумку виявляється сувора залежність між різними частинами інтерфейсу, які можуть бути не пов'язані під час його реалізації.

У цьому випадку інтерфейс класу поділяється на окремі частини, які становлять окремі інтерфейси. Потім ці інтерфейси незалежно один від одного можуть застосовуватися і змінюватися. Унаслідок застосування принципу поділу інтерфейсів система стає слабкозв’язаною, тому її легше модифікувати й оновлювати.

Принцип інверсії залежностей (Dependency Inversion Principle, DIP). Цей принцип описують так: залежності всередині системи будуються на основі абстракцій. Модулі верхнього рівня не залежать від модулів нижнього рівня. Абстракції не повинні залежати від деталей. Деталі мають залежати від абстракцій.

Програмне забезпечення потрібно розробляти так, щоб різні модулі були автономними й з'єднувалися один з одним за допомогою абстракції.

Існує три базові принципи впровадження залежностей:

1 Упровадження залежностей за допомогою конструктора (Constructor Injection):

```
public class Notification
{
 private IMessenger _messenger;
 public Notification (IMessenger mes)
 {
 _messenger = mes;
 }

 public void DoNotify ()
 {
 _messenger.Send ();
 }
}
```

2 Упровадження залежностей за допомогою властивості (Property Injection) public class Notification:

```
{
 private IMessenger _messenger;
 public Notification ()
 {
```

```
 }

 public IMessenger Messanger
 {
 set
 {
 _messenger = value;
 }
 }

 public void DoNotify ()
 {
 _messenger.Send ();
 }
}
```

3 Упровадження залежностей за допомогою методу (Method Injection):

```
public class Notification
{
 public void DoNotify (IMessenger mes)
 {
 mes.Send ();
 }
}
```

6.2 Принцип KISS – робіть речі простіше

Абревіатура KISS утворена від таких варіантів: «Keep it simple, stupid»,«Keep it short and simple», «Keep it simple and straightforward», «Keep it small and simple». Принцип проєктування, прийнятий у ВМС США 1960 року, пов’язаний з авіаконструктором Кларенсом Джонсоном (1910–1990).

KISS – це принцип проєктування і програмування, при якому простота системи декларується як основна мета або цінність.

Чимало програмних систем необґрунтовано перевантажені майже непотрібними функціями, що погіршує зручність їх використання кінцевими користувачами, а також ускладнює їх підтримку і розвиток розробниками. Дотримання принципу KISS дає змогу розробляти рішення, які прості у використанні та в супроводі.

У проєктуванні дотримання принципу KISS виражається в такому:

– немає сенсу реалізовувати додаткові функції, які не будуть використовуватися зовсім або їх використання вкрай малоймовірне, адже

зазвичай більшості користувачів достатньо базового функціонала, а ускладнення тільки шкодить зручності додатка;

– не слід перевантажувати інтерфейс опціями, які не будуть потрібні більшості користувачів, набагато простіше передбачити для них окремий «розширений» інтерфейс (або зовсім відмовитися від цього функціонала);

– безглуздо робити реалізацію складної бізнес-логіки, яка враховує абсолютно всі можливі варіанти поведінки системи, користувача і навколишнього середовища, – по-перше, це просто неможливо, а по-друге, ірраціонально з комерційного погляду.

У програмуванні дотримання принципу KISS можна описати так:

– немає сенсу безмежно збільшувати рівень абстракції;

– безглуздо закладати в проєкт функції «про запас», які, ймовірно, коли-небудь кому-небудь знадобляться (у цьому випадку KISS подібний до принципу YAGNI);

– декомпозиція на прості складові – це архітектурно правильний підхід (у цьому випадку KISS подібний до принципу DRY);

– абсолютна математична точність або гранична деталізація потрібні не завжди, дані можна і потрібно обробляти з тією точністю, яка достатня для якісного вирішення завдання, а деталізацію видавати в потрібному користувачеві, а не в максимально можливому обсязі.

6.3 Принцип DRY

Абревіатура DRY утворена від «Do not Repeat Yourself» / «Не повторюй себе».

Дотримання принципу програмування DRY дає змогу домогтися високого супроводу проєкту, простоти внесення змін і якісного тестування.

Якщо код не дублюється, то для зміни логіки досить внести виправлення лише в одному місці, оскільки простіше тестувати одну (нехай і більш складну) функцію, а не набір із десятків однотипних. Дотримання принципу DRY завжди приводить до декомпозиції складних алгоритмів на прості функції. А декомпозиція складних операцій на більш прості (і повторно використовувані) значно спрощує розуміння програмного коду.

У межах одного програмного класу (або модуля) дотримуватися DRY і не повторюватися зазвичай досить просто. Проте у великих проєктах ситуація з DRY трохи складніша – повтори найчастіше з'являються через відсутність у розробників цілісної картини або неузгодженості дій у межах команди.

У проєктуванні DRY означає, що доступ до конкретного функціонала має бути доступний в одному місці, уніфікований і згрупований, а не «розкиданий» ПЗ системи в довільних варіаціях. Цей підхід подібний до принципу єдиної відповідальності SOLID.

6.4 Принцип YAGNI

Абревіатура YAGNI означає «You Is not Gonna Need It» / «Вам це не знадобиться».

Дотримування цього принципу полягає в тому, що можливості, які не описані у вимогах до системи, не повинні реалізовуватися.

Основна проблема, яку розв’язує принцип YAGNI, – це запобігання зайвому абстрагуванню експериментам «із цікавості» і реалізації функціонала, який зараз не потрібен, але, на думку розробника, може або незабаром знадобитися, або просто буде корисним, хоча в реальності такого найчастіше не відбувається.

Будь-які «бонусні» можливості ускладнюють супровід, збільшують імовірність помилок і ускладнюють взаємодію з продуктом – між обсягом кодової бази й описаними характеристиками є пряма залежність.

Поки функціональність дійсно не потрібна, важко повністю передбачити, що вона має робити, і протестувати її. Якщо нову функціональність ретельно не протестовано, вона може неправильно працювати, коли згодом знадобиться.

7 БАГАТОШАРОВА АРХІТЕКТУРА ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

7.1 Загальна характеристика багатошарової архітектури

Терміни шар (layer) і ланка (tier) часто плутають. Шар, або рівень, позначає логічний поділ функціональності, а ланка – фізичне розгортання на різних системах.

Багаторівнева архітектура забезпечує угруповання пов'язаної функціональності додатка в різних шарах, що вибудовуються вертикально, один поверх одного. Функціональність кожного шару об'єднана спільною роллю або відповідальністю. Шари слабозв'язані, і між ними здійснюється обмін даними. Правильний поділ додатка на шари допомагає підтримувати чіткий поділ функціональності, що забезпечує гнучкість, а також зручність і простоту обслуговування.

Багатошарова архітектура описана як перевернута піраміда повторного використання, у якій кожен шар агрегує відповідальності й абстракції рівня, розташованого безпосередньо під ним (рисунок 7.1).

За умови чіткого поділу на шари компоненти одного шару можуть взаємодіяти тільки з компонентами того самого шару або компонентами шару, розташованого під цим шаром. Більш вільний поділ на шари дає змогу компонентам взаємодіяти з компонентами всіх нижчих шарів.

Шари додатка можуть розміщуватися фізично на одному комп'ютері (на одній ланці) або бути розподілені між різними комп'ютерами (n-ланок), і зв'язок між компонентами різних ланок здійснюється через чітко

визначені інтерфейси. Наприклад, типовий вебдодаток складається з шару подання (функціональність, пов'язана з UI), бізнес-шару (оброблення бізнес-правил) і шару даних (функціональність, пов'язана з доступом до даних, часто майже повністю реалізована за допомогою високорівневих інфраструктур доступу до даних).

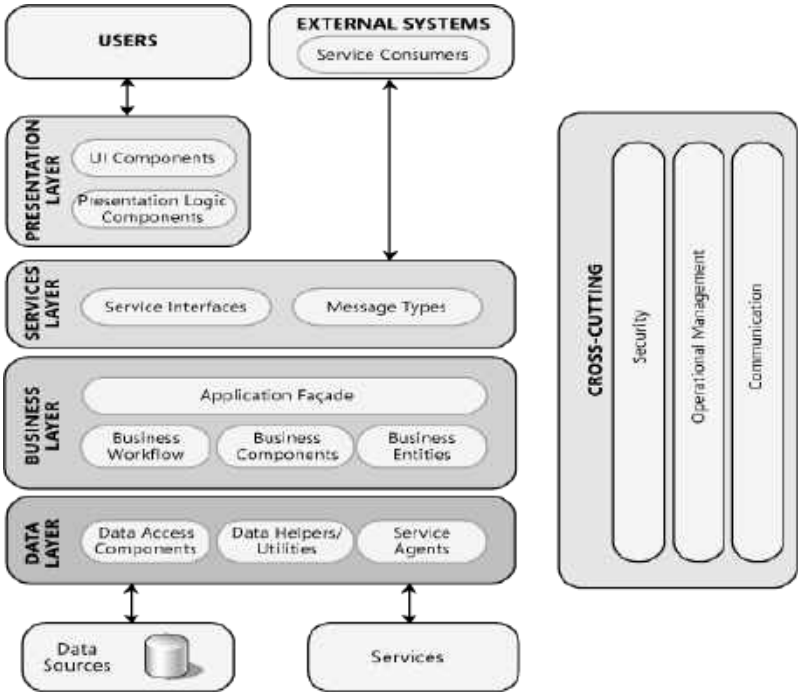

Рисунок 7.1 – Багаторівнева архітектура інформаційної системи

7.2 Загальні принципи проєктування багатошарової архітектури

Розглянемо загальні принципи проєктування з використанням багатошарової архітектури.

Абстракція. Багатошарова архітектура являє собою систему як єдине ціле, забезпечуючи при цьому досить деталей для розуміння ролей і відповідальностей окремих шарів і зв’язків між ними.

Інкапсуляція. Під час проєктування немає необхідності робити будь-які припущення про типи даних, методи і властивості або реалізації, оскільки всі ці деталі приховано в межах шару.

Чітко визначені функціональні шари. Поділ функціональності між шарами дуже чіткий. Верхні шари, такі як шар подання, посилають команди нижнім рівням, як-от бізнес-шар і шар даних, і можуть реагувати на події, що виникають у цих шарах, забезпечуючи можливість передавання даних між шарами вгору і вниз.

Висока пов'язаність (high cohesion). Чітко визначені межі відповідальності для кожного шару і гарантоване включення в шар тільки

функціональності, безпосередньо пов'язаної з його завданнями, допоможуть забезпечити максимальну пов'язаність у межах шару.

Можливість повторного використання. Відсутність залежностей між нижніми і верхніми шарами забезпечує потенційну можливість їх повторного використання в інших сценаріях.

Слабка зв'язність (low coupling). Для забезпечення слабкого зв'язування між шарами зв'язок між ними ґрунтується на абстракції та подіях.

Прикладами багатошарових додатків можуть бути бізнес-додатки (line-of-business, LOB), такі як системи бухгалтерського обліку й управління замовниками; вебдодатки та вебсайти підприємств; настільні або смартклієнти підприємств із централізованими серверами додатків для розміщення бізнес-логіки.

Багатошарова архітектура підтримується низкою шаблонів проєктування. Наприклад, під назвою *Separated Presentation* (Розділене подання) об'єднується кілька шаблонів, які поділяють взаємодію користувача з UI, подання, бізнес-логіку і дані додатка, з якими працює користувач. Розділене подання дає змогу створювати UI в графічних дизайнерів, тоді як розробники пишуть керівний код. Такий поділ функціональності на ролі підвищує можливість тестування поведінки окремих ролей. Розглянемо основні принципи шаблонів Separated Presentation.

Поділ функціональності. Шаблони Separated Presentation поділяють функціональність UI на ролі. Наприклад, у шаблоні MVC є три ролі: Model (Модель), View (Подання) і Controller (Контролер). *Модель* подає дані (можливо, модель предметної області, яка включає бізнес-правила). *Подання* відповідає за зовнішній вигляд UI. *Контролер* обробляє запити, працює з *Моделлю* і виконує інші операції.

Повідомлення на основі подій. Шаблон Observer (Спостерігач) зазвичай використовується для повідомлення *Подання* про зміни даних, керованих *Моделлю*.

Делегування оброблення подій. *Контролер* обробляє події, що формуються елементами керування UI в *Поданні*.

Як приклади шаблонів Separated Presentation розглянемо шаблон Passive View (Пасивне подання) і шаблон Supervising Presenter (Спостерігач-презентатор), також званий Supervising Controller (Спостерігач-контролер).

Основними перевагами багатошарового архітектурного стилю і застосування шаблона Separated Presentation є:

1 Абстракція. Рівні забезпечують можливість внесення змін на абстрактному рівні. Використовуваний рівень абстракції кожного шару може бути збільшений або зменшений.

2 Ізоляція. Оновлення технологій можуть бути ізольовані в окремих шарах, що допоможе зменшити ризик і мінімізувати вплив на всю систему.

3 Керованість. Поділ основних функцій допомагає ідентифікувати залежності й організувати код у секції, що підвищує керованість.

4 Продуктивність. Розподіл шарів за декількома фізичними рівнями може поліпшити масштабованість, відмовостійкість і продуктивність.

5 Можливість повторного використання. Ролі підвищують можливість повторного використання. Наприклад, у MVC *Контролер* часто може використовуватися повторно з іншими сумісними *Поданнями* для забезпечення характерного для ролі або настроєного для користувача подання тих самих даних і функціональності.

6 Поліпшення тестованості. Є результатом наявності строго визначених інтерфейсів шарів, а також можливості перемикання між різними реалізаціями інтерфейсів шарів. Шаблони Separated Presentation дають змогу створювати під час тестування фіктивні об'єкти, що імітують поведінку реальних об'єктів, таких як *Модель*, *Контролер* або *Подання*.

Можливість застосування багатошарової архітектури необхідно розглянути, якщо наявні вже готові рівні, відповідні для повторного використання в інших додатках; якщо є додатки, що надають відповідні бізнес-процеси через інтерфейси сервісів; або якщо створюється складний додаток і попереднє проєктування потребує поділу, щоб групи могли зосередитися на різних ділянках функціональності. Багатошарова архітектура також буде доречною, якщо програма має підтримувати різні типи клієнтів і різні пристрої або якщо потрібно реалізувати складні і/або настроювати бізнес-правила і процеси.

Зверніть увагу на шаблон Separated Presentation, якщо хочете отримати поліпшену тестованість і спростити обслуговування функціональності UI або відокремити завдання створення дизайну UI від розроблення керівного коду. Ці шаблони також будуть корисні, якщо подання UI не містить жодного коду оброблення запитів і не реалізує жодної бізнес-логіки.

7.3 Логічний поділ на шари

Незалежно від типу проєктованого додатка і того, чи є в нього призначений для користувача інтерфейс або він є сервісним додатком, який просто надає послуги (не слід плутати із шаром сервісів додатка), його структуру можна поділити на логічні групи програмних компонентів. Ці логічні групи називаються шарами. Шари допомагають поділити різні типи завдань, які здійснюються цими компонентами, що спрощує створення дизайну та підтримує можливість повторного використання компонентів. Кожен логічний шар включає низку окремих типів компонентів, згрупованих у підшари, кожен із підшарів виконує певний тип завдань.

Після визначення універсальних типів компонентів, які наявні в більшості рішень, можна створити схему програми або сервісу і потім використовувати цю схему як ескіз створюваного дизайну. Поділ програми

на шари, що виконують різні ролі та функції, допомагає максимально підвищити зручність і простоту обслуговування коду, оптимізувати роботу програми при різних схемах розгортання і забезпечує чітке розмежування областей застосування певної технології або ухвалення певних проєктних рішень.

На найвищому і найбільш абстрактному рівні логічне подання архітектури системи може розглядатися як набір компонентів, що взаємодіють, згруповані в шари. На рисунку 7.2 показано спрощене високорівневе подання цих шарів і їх відношень / зв'язків із користувачами, іншими додатками, що викликають сервіси, реалізовані в бізнес-шарі додатка, джерелами даних, такими як реляційні бази даних або вебсервіси, що забезпечують доступ до даних, і зовнішніми або віддаленими сервісами, що використовуються додатком.

Ці шари фізично можуть розміщуватися на одному або різних рівнях. Якщо вони розміщуються на різних рівнях або розділені фізичними межами, дизайн повинен забезпечувати це.

Типовий тришаровий дизайн, наведений на рисунку 7.2, містить такі шари:

1 *Шар подання*. Цей шар містить орієнтовану на користувача функціональність, яка відповідає за реалізацію взаємодії користувача із системою, і зазвичай містить компоненти, що забезпечують загальний зв'язок з основною бізнес-логікою, інкапсульованою в бізнес-шарі.

2 *Бізнес-шар (шар бізнес-логіки)*. Цей шар реалізує основну функціональність системи і інкапсулює пов'язану з нею бізнес-логіку. Зазвичай цей шар складається з компонентів, деякі з яких надають інтерфейси сервісів, доступні для використання іншими учасниками взаємодії.

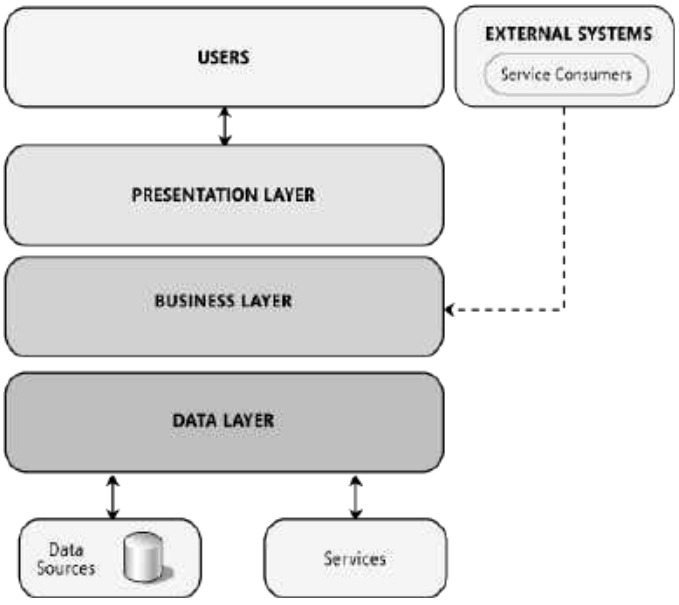

Рисунок 7.2 – Логічне подання архітектури багатошарової системи

3 *Шар доступу до даних*. Цей шар забезпечує доступ до даних, що зберігаються в межах системи, і даних, що надаються іншими мережевими системами. Доступ може здійснюватися через сервіси. Шар даних надає універсальні інтерфейси, які можуть використовуватися компонентами бізнес-шару.

7.4 Сервіси і шари

Рішення, основане на сервісах, можна розглядати як набір сервісів, що взаємодіють один з одним шляхом передання повідомлень. Концептуально ці сервіси можна вважати компонентами рішення в цілому. Однак кожен сервіс утворений програмними компонентами, як будь-який інший додаток, і ці компоненти можуть бути логічно згруповані в шар подання, бізнес-шар і шар даних. Інші додатки можуть використовувати сервіси, незважаючи на спосіб їх реалізації. Принципи багатошарового дизайну, про які йшлося в попередньому розділі, однаково застосовуються і до основаних на сервісах рішеннях.

Звичайним підходом під час створення програми, яка повинна забезпечувати сервіси для інших додатків, а також реалізовувати безпосередню підтримку клієнтів, є використання шару сервісів, який надає доступ до бізнес-функціональності програми. Шар сервісів забезпечує альтернативне подання, що дає змогу клієнтам використовувати інший механізм для доступу до додатка.

У цьому сценарії користувачі можуть виконувати доступ до додатка через шар подання, що обмінюється даними з компонентами бізнес-шару або безпосередньо, або через фасад додатка в бізнес-шарі, якщо методи зв'язку потребують композиції функціональності. Тим часом зовнішні клієнти й інші системи можуть виконувати доступ до додатка і використовувати його функціональність шляхом взаємодії з бізнес-шаром через інтерфейси сервісів. Це покращує можливості додатка для підтримки безлічі типів клієнтів, сприяє повторному використанню і більш високому рівню композиції функціональності в додатках.

У деяких випадках шар подання може взаємодіяти з бізнес-шаром через шар сервісів. Але це не є обов'язковою умовою. Якщо фізично шар подання і бізнес-шар розміщуються на одному рівні, вони можуть взаємодіяти безпосередньо.

7.5 Розподіл шарів і компонентів

Шари і компоненти мають розподілятися за різними фізичними рівнями, тільки якщо в цьому є необхідність. Типовими причинами реалізації розподіленого розгортання вважають політику безпеки, фізичні обмеження, спільно використані бізнес-логіку і масштабованість.

Якщо компоненти подання вебдодатка здійснюють синхронний доступ до компонентів бізнес-шару, а обмеження безпеки не потребують наявності межі довіри між шарами, слід розглянути можливість розгортання компонентів бізнес-шару і шару подання на одному рівні, це забезпечить максимальну продуктивність і керованість.

У клієнтських додатках, у яких оброблення UI виконується на клієнтському комп'ютері, варіант розгортання компонентів бізнес-шару на окремому бізнес-рівні може бути вибраний із міркувань безпеки і для підвищення керованості.

Розгортаються бізнес-сутності на одному рівні з кодом, що їх використовує. Це може означати їх розгортання в декількох місцях, наприклад, розміщення копій на окремому рівні (подання або даних), логіка якого використовує або посилається на ці бізнес-сутності. Розгортаються компоненти агентів сервісу на тому самому рівні, що й код, що викликає ці компоненти, якщо обмеження безпеки не потребують наявності межі довіри між ними.

Розгляньте можливість розгортання асинхронних компонентів бізнес-шару, компонентів робочого процесу і сервісів з однаковими характеристиками завантаження і введення / виведення на окремому рівні. Це дає змогу настроїти інфраструктуру для забезпечення максимальної продуктивності та масштабованості.

7.6 Визначення правил взаємодії між шарами

Коли справа доходить до стратегії поділу на шари, необхідно визначити правила взаємодії шарів один з одним. Основна мета завдання правил взаємодії – мінімізація залежностей і виключення циклічних посилань. Наприклад, якщо два шари мають залежності від компонентів третього шару, з'являється циклічна залежність. Загальним правилом, якого слід дотримуватися в цьому випадку, є дозвіл тільки односпрямованої взаємодії між шарами через застосування одного з таких підходів:

1 *Взаємодія зверху вниз*. Шари можуть взаємодіяти із шарами, розміщеними нижче, але нижні шари ніколи не можуть взаємодіяти із шарами, що розміщені вище. Це правило дає змогу уникнути циклічних залежностей між шарами. Використання подій дасть змогу сповіщати компоненти розміщених вище шарів про зміни в нижніх шарах без введення залежностей.

2 *Сувора взаємодія*. Кожен шар повинен взаємодіяти тільки із шаром, розміщеним безпосередньо під ним. Це правило забезпечить чіткий поділ, при якому кожен шар знає тільки про шари під ним. Позитивний ефект від цього правила в тому, що зміни в інтерфейсі шару будуть впливати тільки на шар, розміщений безпосередньо над ним. Слід застосовувати цей підхід під час проєктування програми, яка передбачає розширювати новою

функціональністю в майбутньому, якщо треба максимально скоротити вплив цих змін, або під час проєктування програми, для якої необхідно забезпечити можливість розподілу на різні рівні.

3 *Вільна взаємодія*. Більш високі шари можуть взаємодіяти з розміщеними нижче шарами безпосередньо, обходячи інші верстви. Це може підвищити продуктивність, але також збільшить залежності. Інакше кажучи, зміни в нижньому шарі можуть впливати на кілька розміщених вище шарів. Цей підхід рекомендується застосовувати під час проєктування програми, яка гарантовано буде розміщуватися на одному рівні (наприклад, клієнтську програму), або під час проєктування невеликого додатка, для якого внесення змін, які зачіпають безліч шарів, не потребує великих зусиль.

8 МІКРОСЕРВІСНА АРХІТЕКТУРА

8.1 Види мікросерсвісних архітектур

Найпростіший і популярний варіант архітектури – монолітна, у якій немає ізоляції та розподіленості. Один моноліт обробляє всі запити, але при цьому виникають такі проблеми:

– недостатня відмовостійкість;
– складність горизонтального масштабування;
– необхідність застосування однієї технології або мови, зокрема невигідність оновлювати або модифікувати величезну кодову базу;
– складність рефакторинга через зберігання коду в одному місці та багато застарілих кодів (legacy);
– труднощі роботи в команді розробників;
– ділення, щоб використовувати повторно.

Другий за популярністю вид архітектури – сукупність монолітів, мікс із моноліту і сервісів або навіть мікросервісів. Отож зберігається моноліт, а доопрацювання виконується з використанням мікросервісів.

Це частково розв’язує проблеми відмовостійкості, масштабованості й одного стеку технологій. Більш сучасний підхід до створення розподілених систем надає мікросервісна архітектура.

Мікросервісна архітектура не нова ідея, а різновид сервіс-орієнтованої архітектури (SOA), яка передбачає модульність розробки і слабку зв'язність компонентів і дає змогу отримати ізольовану й розподілену систему. Головний мінус класичної SOA – загальна шина даних Enterprise Service Bus із величезними специфікаціями і труднощами роботи з абстракціями і фасадами. Сучасні технології створення мікросервісних систем позбавлені цього недоліку.

8.2 Типова мікросервісна архітектура

Мікросервісна архітектура успадкувала від попередньої ізоляцію і розподіленість (рисунок 8.1). У цій архітектурі не використовується як шина даних, за винятком окремих випадків на користь продуктивності. За класичною схемою компоненти ізолюються і на рівні коду, і на рівні бази. Сервіси можуть бути написані різними мовами і використовувати різні технології зберігання даних.

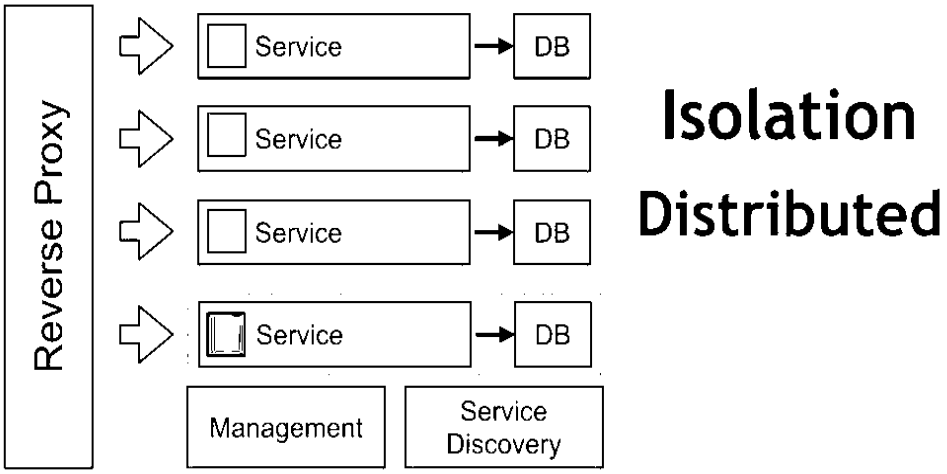

Рисунок 8.1 – Структура мікросервісної архітектури

Поділяють мікросервіси з погляду або бізнесу, або програміста для перевикористання. Перешкоджають поділу дві речі:

– внутрішні зв'язки – у разі тісної взаємодії мікросервіси об'єднують;

– транзакції – у різних мікросервісів бази даних ізольовані, а потрібна одна спільна.

API Gateway – це центральні «двері» для всіх публічних API. Це єдина точка входу для клієнтських програм і є важливою складовою найкращих практик керування API (API Management).

8.3 Структура мікросервісів

Мікросервіси складаються з трьох шарів: невеликих обробників, бізнес-логіки й маперів даних. Послідовність виконання запиту має такий вигляд (рисунок 8.2).

У сервісному шарі зосереджується 99 % усього коду. Оскільки в мікросервісі кілька обробників, слід використовувати Data Transfer Object (DTO), до якого буде здійснювати GET-запит. Це полегшує оброблення та валідацію.

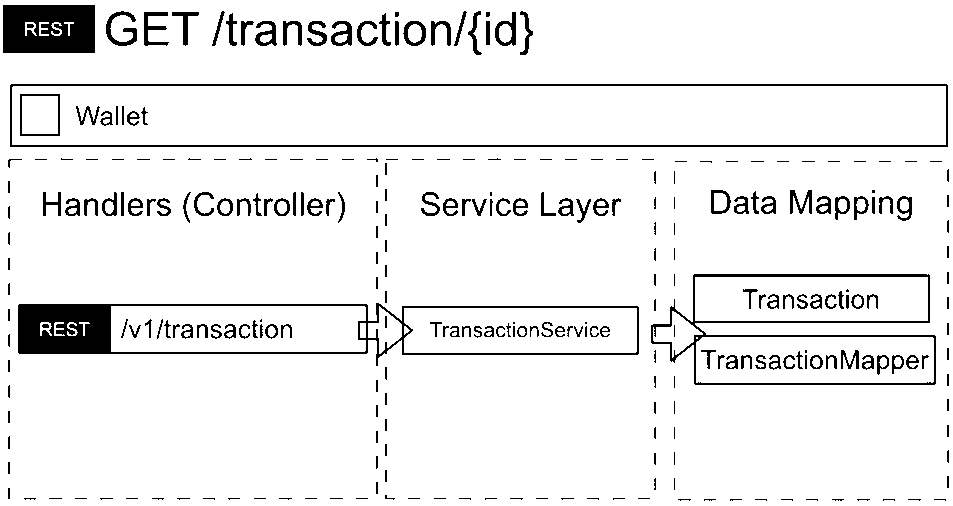

Рисунок 8.2 – Структура мікросервісів

8.4 Оброблення розподілених транзакцій у мікросервісній архітектурі

Загальноприйнята практика полягає в тому, що кожна база даних має належати лише одному мікросервісу. Інші сервіси не повинні мати можливості безпосереднього доступу до бази даних, лише через API сервісу-власника. Це дає змогу кожному сервісу керувати консистентністю та структурою бази даних, до якої сервіс має доступ. Також це дає змогу вибирати найбільш відповідну базу даних для конкретних цілей без будь-якої залежності від вимог системи в цілому. Але при цьому виникає проблема розподілених транзакцій.

Транзакції, що охоплюють кілька фізичних систем або комп'ютерів у мережі, називаються розподіленими транзакціями.

З упровадженням мікросервісної архітектури бази даних втрачають свою ACID-природу, оскільки транзакції поширюються між безліччю мікросервісів і, отже, баз даних. Для розв’язання проблем оброблення конкурентних запитів застосовуються два такі підходи:

– двофазна фіксація;
– узгодженість у результаті та компенсація / SAGA.

Двофазна фіксація. Як зрозуміло з назви, такий спосіб оброблення транзакцій передбачає два етапи: фазу підготовки і фазу фіксації. Важливу роль у цьому випадку відіграє координатор транзакцій, що організує життєвий цикл транзакції.

На підготовчому етапі всі мікросервіси, які беруть участь у роботі, готуються до фіксації та повідомляють координатора (Transaction Coordinator), що готові завершити транзакцію. Потім на наступному етапі

або відбувається фіксація, або координатор транзакції видає всім мікросервісам команду виконати відкат. Розглянемо для прикладу систему інтернет-магазину.

У наведеному прикладі (рисунок 8.3), коли користувач надсилає запит на замовлення, координатор *Transaction Coordinator* спершу починає глобальну транзакцію, маючи повну інформацію про контекст.

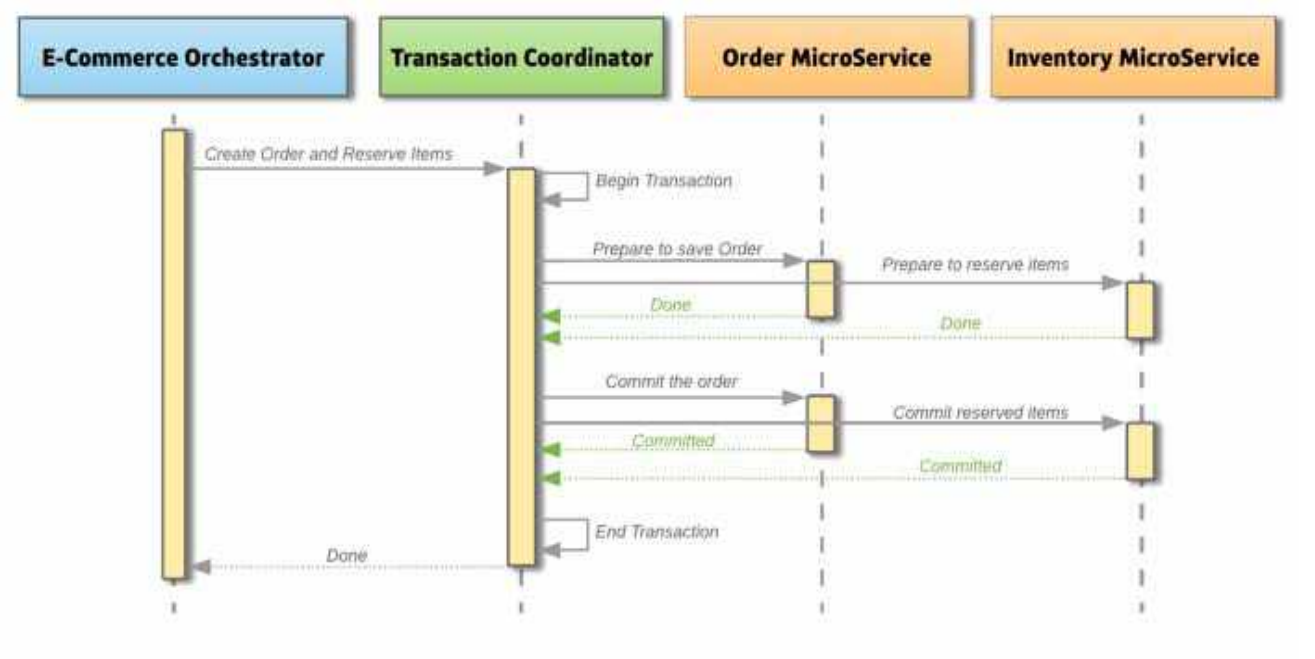

Рисунок 8.3 – Успішна двофазна фіксація в мікросервісній системі

Спочатку *Transaction Coordinator* відправляє команду *Prepare* мікросервісу *Order Microservice*, щоб створити замовлення. Потім відправляє команду *Prepare* до *Inventory Microservice*, щоб зарезервувати товари. Коли обидва сервіси готові внести зміни, вони блокують об'єкти від подальших змін і повідомляють про це *Transaction Coordinator*. Щойно *Transaction Coordinator* підтвердить, що всі мікросервіси готові застосувати свої зміни, він накаже цим мікросервісам зберегти їх, надіславши запит на фіксацію транзакції. У цей момент усі об'єкти будуть розблокованими.

У сценарії відмови (рисунок 8.4) (якщо в будь-який момент окремо взятий мікросервіс не зможе виконати запит) *Transaction Coordinator* скасує транзакцію і почне процес відкату. На схемі *Order Microservice* з якоїсь причини не зміг створити замовлення, але *Inventory Microservice* відгукнувся, що готовий створити замовлення. Координатор *Transaction Coordinator* викликає скасування на *Inventory Microservice*, після чого сервіс виконає команду відкату всіх зроблених змін і розблокує об'єкти бази даних.

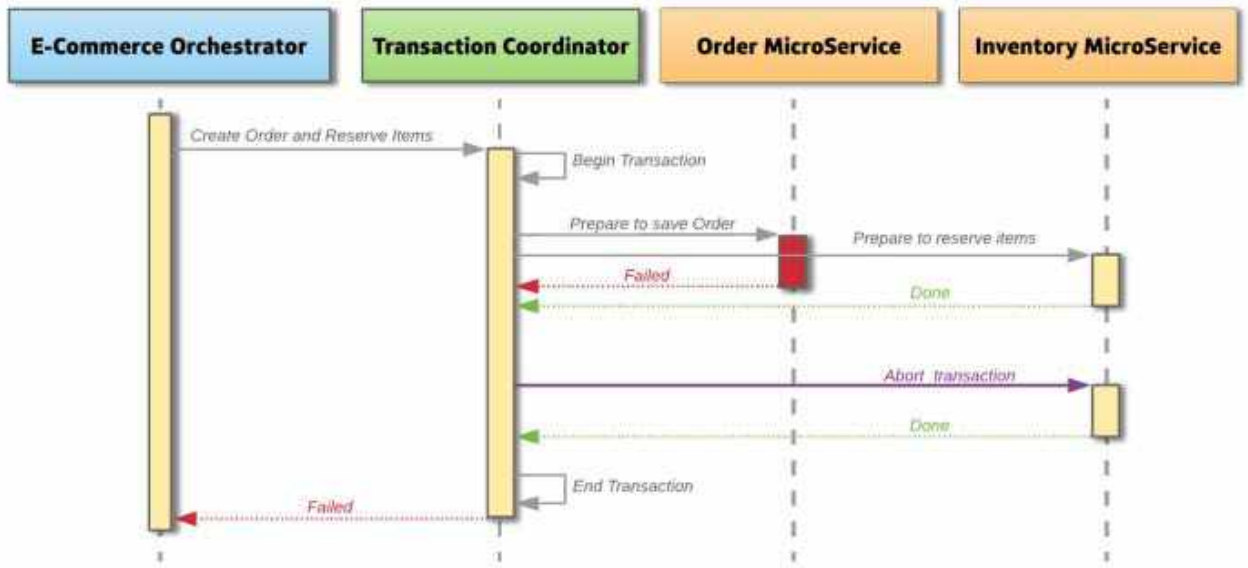

Рисунок 8.4 – Невдала двофазна фіксація під час роботи з мікросервісами

Перевагами двофазної фіксації є:

1 Атомарність транзакції. Транзакція завершиться або тоді, коли обидва мікросервіси спрацюють успішно, або тоді, коли мікросервіси не внесуть жодних змін.

2 Можливість ізолювати читання від запису, оскільки зміни в об'єктах не помітні доти, доки координатор транзакцій зафіксує ці зміни.

3 Забезпечення синхронного виклику, при якому клієнт буде повідомлений про успіх чи невдачу.

Недоліками цього підходу можна вважати:

1 Досить повільне здійснення двофазної фіксації, як порівняти з операціями над одним мікросервісом. Існує сильна залежність від координатора транзакцій, що може значно уповільнювати роботу системи в період високої завантаженості.

2 Тривале блокування рядків баз даних. Блокування може стати вузьким місцем, що знижує продуктивність, причому може виникнути взаємне блокування, де дві транзакції уповільнюють одна одну.

Узгодженість у результаті та компенсація / SAGA. Одне з кращих визначень узгодженості в результаті подано на сайті microservices.io: кожен сервіс публікує подію щоразу, коли оновлює свої дані. Інші сервіси підписуються на події. Отримавши події, сервіс оновлює свої дані. При такому підході розподілена транзакція виконується як сукупність асинхронних локальних транзакцій на відповідних мікросервісах. Мікросервіси обмінюються інформацією через шину подій.

Розглянемо як приклад систему, що працює в інтернет-магазині (рисунок 8.5).

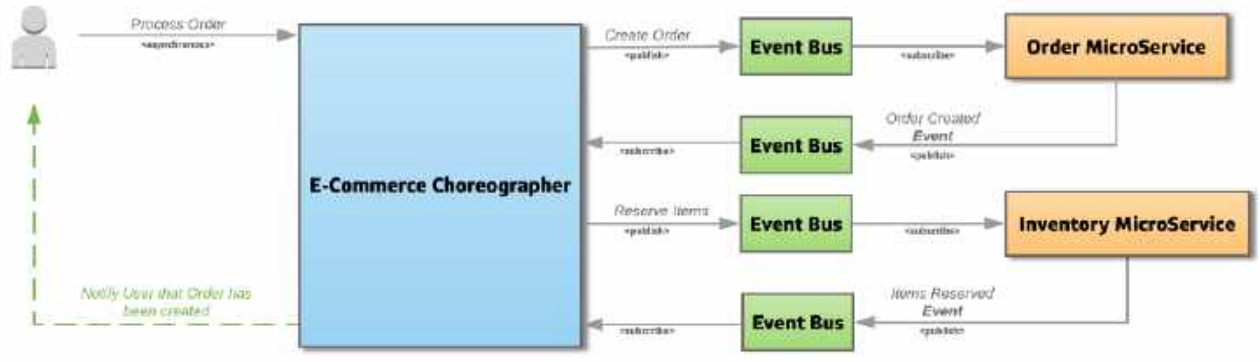

Рисунок 8.5 – Узгодженість у результаті / SAGA, успішний результат

У наведеному вище прикладі (див. рисунок 8.5) клієнт вимагає, щоб система обробила замовлення. При цьому запиті *Choreographer* створює подію *Create Order* (створити замовлення), що починає транзакцію. Мікросервіс *Order Microservice* аналізує цю подію і створює замовлення – якщо ця операція пройшла успішно, то він створює подію *Order Created* (замовлення створено). Координатор *Choreographer* аналізує цю подію і переходить до замовлення товарів, створюючи подію *Reserve Items* (зарезервувати товари). Мікросервіс *Inventory Microservice* аналізує цю подію і замовляє товари; якщо цю подію здійснено успішно, то створюється подія *Items Reserved* (товари зарезервовані). У цьому прикладі це означає, що транзакцію закінчено.

Уся комунікація на основі подій між мікросервісами відбувається через шину подій, а за її організацію відповідає інша система – так розв’язується проблема із зайвою складністю.

Якщо з певної причини *Inventory Microservice* не вдалося зарезервувати товари (рисунок 8.6), він створює подію *Failedto ReserveItems* (не вдалося зарезервувати товари). Координатор *Choreographer* аналізує цю подію і запускає компенсувальну транзакцію, створюючи подію *Delete Order* (видалити замовлення). Мікросервіс *Order Microservice* аналізує цю подію і видаляє раніше створене замовлення.

Серйозна перевага такого підходу полягає в тому, що кожен мікросервіс зосереджується лише на власній атомарній транзакції. Робота мікросервісів не блокується, якщо на роботу іншого сервісу потрібно порівняно багато часу. Це також означає, що не потрібно блокувати і базу даних. За допомогою такого підходу можна забезпечити хорошу масштабованість системи під час роботи з високим системним навантаженням, оскільки пропоноване рішення асинхронне і основане на роботі з подіями.

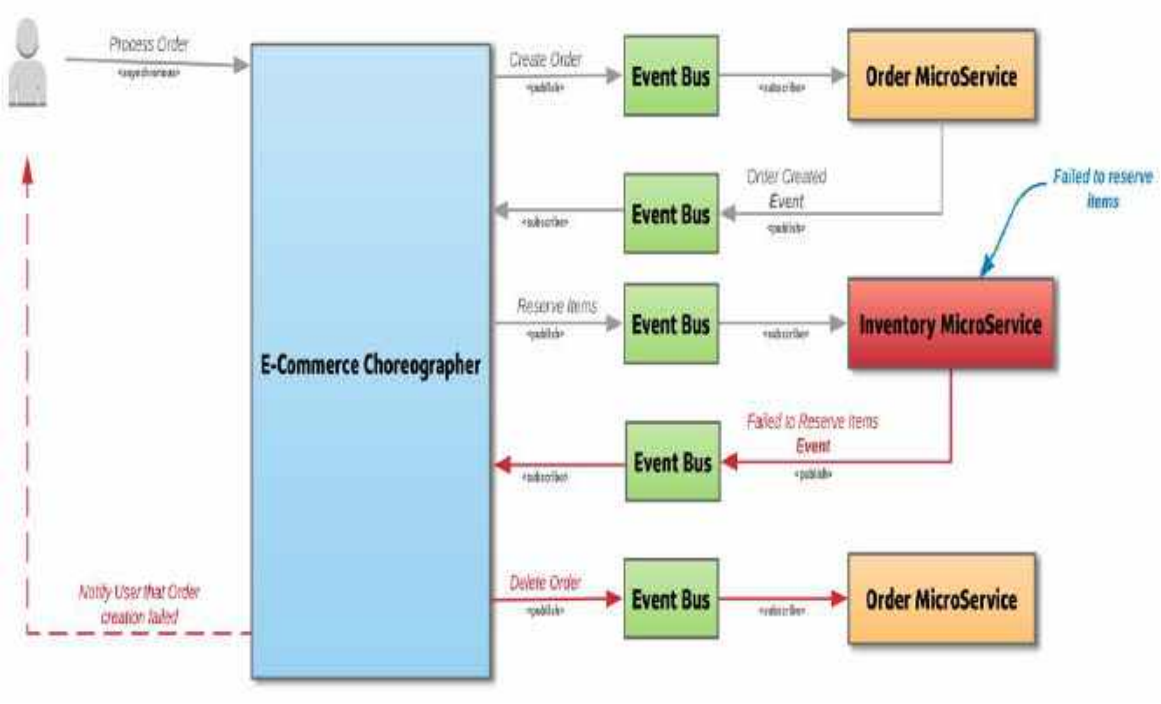

Рисунок 8.6 – Узгодженість у результаті / SAGA, невдалий результат

Основний недолік цього підходу полягає в тому, що тут не забезпечується ізоляція під час читання. Таким чином, у наведеному вище прикладі клієнт побачить, що замовлення було створено, але вже через секунду замовлення буде видалено під час компенсації транзакції. Крім того, коли збільшується кількість мікросервісів, ускладнюються їх настроювання й підтримка.

Якщо необхідно оновити дані відразу у двох місцях у результаті однієї події, то підхід з узгодженістю в результаті / SAGA кращий під час оброблення розподілених транзакцій порівняно з двофазним підходом. Основна причина в тому, що двофазний підхід у розподіленому середовищі не масштабується.

9 АРХІТЕКТУРНИЙ ШАБЛОН MVC

9.1 Загальна характеристика MVC

Model-View-Controller (MVC) – це фундаментальний патерн, який застосовується в багатьох технологіях. У класичному варіанті MVC складається з трьох частин, які й дали йому назву.

Model (Модель). *Моделлю* зазвичай вважають частину, яка містить у собі функціональну бізнес-логіку програми. *Модель* має бути повністю незалежною від інших частин продукту. *Модель* нічого не знає ні про *Подання*, ні про *Контролер*, що робить можливим її розроблення і тестування як незалежного компонента – це є головною особливістю MVC.

Модель може бути пасивною або активною. Пасивна *Модель* жодним чином не вплине ні на *Подання,* ні на *Контролер*. У цьому випадку всі зміни *Моделі* відслідковуються *Контролером*, який відповідає за перерисовку *Подання*, коли це необхідно. Але зазвичай під MVC все-таки мають на увазі варіант з активною *Моделлю*.

View (Подання). *Подання* відображає *Модель*. Це означає, що воно якимось чином має отримувати з неї потрібні для відображення дані. Найбільш поширені такі два варіанти: активне *Подання*, яке знає про *Моделі* і само бере з неї потрібні дані; пасивне *Подання*, якому дані поставляє *Контролер*. У цьому випадку *Подання* з *Моделлю* жодним чином не пов'язане.

Controller (Контролер). Контролер є, мабуть, найбільш неоднозначним компонентом. Проте спільним є те, що *Контролер* завжди знає про *Моделі* та може її змінювати (зазвичай унаслідок дій користувача).

9.2 Різновиди MVC

Найбільш поширеними різновидами MVC-патерну є:
– Model-View-Controller (*Модель-Подання-Контролер*);
– Model-View-Presenter (*Модель-Подання-Пред'явник*);
– Model-View-View Model (*Модель-Подання-Подання Моделі*).

Model-View-Controller. Основна ідея цього патерну (рисунок 9.1) в тому, що і *Контролер* і *Подання* залежать від *Моделі*, але *Модель* не залежить від цих двох компонентів.

Контролер визначає, які *Подання* має бути відображено в певний момент.

Події *Подання* можуть вплинути тільки на *Контролер*. *Контролер* може вплинути на *Модель* і визначити інше *Подання*.

Можливо кілька подань тільки для одного *Контролера*.

Контролер перехоплює подію ззовні та відповідно до закладеної в нього логіки реагує на цю подію, змінюючи *Модель* за допомогою виклику відповідного методу. Після зміни *Модель* надсилає подію про те, що вона змінилася, і всі підписані на цю подію *Подання*, отримавши виклик, звертаються до *Моделі* за оновленими даними, потім здійснюється відображення оновлених даних.

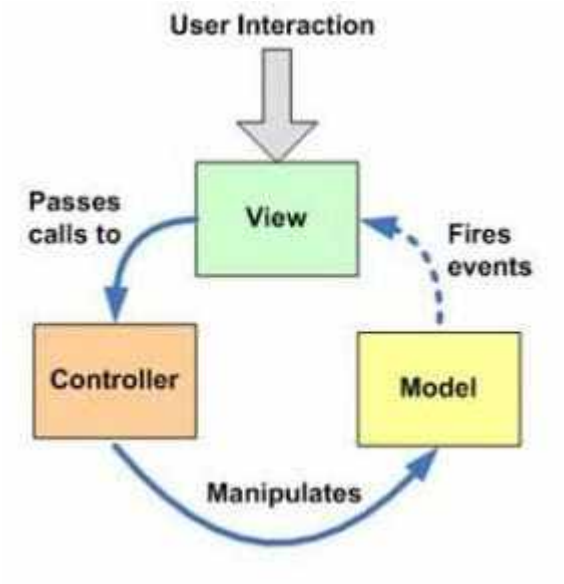

Рисунок 9.1 – Структура Model-View-Controller

Model-View-Presenter. Цей підхід (рисунок 9.2) дає змогу створювати абстракцію *Подання*. Для цього необхідно виділити інтерфейс *Подання* з певним набором властивостей і методів. *Пред'явник* отримує посилання на реалізацію інтерфейсу, підписується на події і за запитом змінює *Модель*.

Пред'явник має двосторонню комунікацію з *Поданням*.

Подання взаємодіє безпосередньо з *Пред'явником* шляхом виклику відповідних функцій або подій примірника *Пред'явника*.

Пред'явник взаємодіє з *Поданням* шляхом використання спеціального інтерфейсу, реалізованого *Поданням*.

Один екземпляр *Пред'явника* пов'язаний з одним *Поданням*.

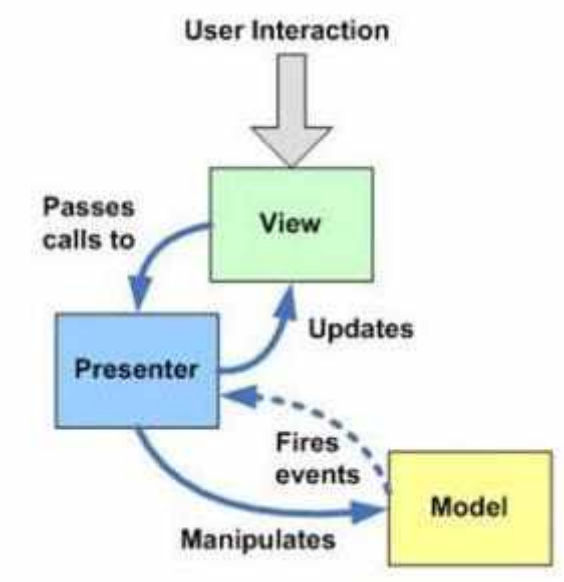

Рисунок 9.2 – Структура Model-View-Presenter

Кожне *Подання* має реалізовувати відповідний інтерфейс. Інтерфейс *Подання* визначає набір функцій і подій, необхідних для взаємодії з користувачем. У *Пред'явника* має бути посилання на реалізацію відповідного інтерфейсу, яке зазвичай передають у конструкторі. Логіка *Подання* повинна мати посилання на екземпляр *Пред'явника*. Усі події *Подання* передаються для оброблення в *Пред'явник* і майже ніколи не обробляються логікою *Подання* (зокрема створення інших *Подань*).

Model-View-View Model. Цей підхід (рисунок 9.3) дає змогу пов'язувати елементи *Подання* з властивостями і подіями *Подання Моделі*. Можна стверджувати, що кожен шар цього патерну не знає про існування іншого шару.

Подання Моделі має двосторонню комунікацію з *Поданням*.

Подання Моделі – це абстракція подання. Зазвичай означає, що властивості подання збігаються з властивостями *Подання Моделі* / *Моделі.*

Подання Моделі не має посилання на інтерфейс *Подання*. Зміна стану *Подання Моделі* автоматично змінює *Подання* і навпаки, оскільки використовується механізм об'єднання даних.

Один екземпляр *Подання Моделі* пов'язаний з одним відображенням.

Під час використання цього патерну *Подання* не реалізує відповідний інтерфейс. У *Поданні* має бути посилання на джерело даних, яким у цьому випадку є *Подання Моделі*. Елементи *Подання* пов'язані з відповідними властивостями і подіями *Подання Моделі*. Зі свого боку *Подання Моделі* реалізує спеціальний інтерфейс, який використовується для автоматичного оновлення елементів *Подання*.

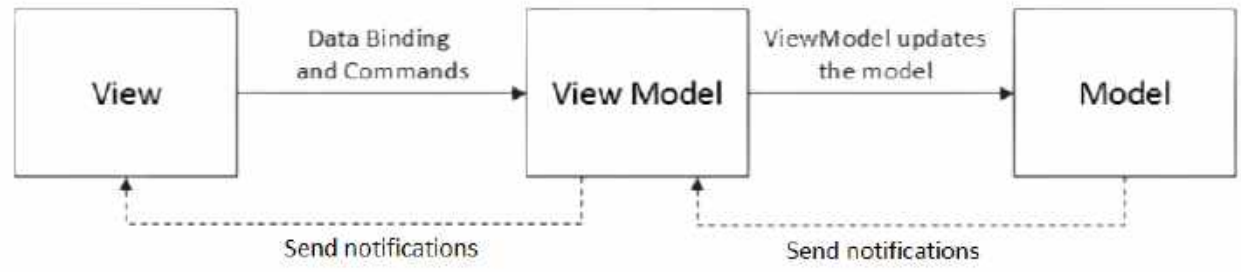

Рисунок 9.3 – Структура Model-View-View Model

Загальні правила вибору патерну:

– MVVM використовується в ситуації, коли можна зв'язувати дані без необхідності введення спеціальних інтерфейсів *Подання*;

– MVP використовується в ситуації, коли неможливе зв'язування даних;

– MVC використовується в ситуації, коли зв'язок між *Поданням* та іншими частинами програми є неможливим (немає можливості використовувати MVVM або MVP).

9.3 Архітектурний MVC

В основі MVC лежать три досить прості ідеї:

1 Відокремлення моделі предметної області (бізнес-логіки) додатка від призначеного для користувача інтерфейсу.

2 Незалежність *Моделі* і синхронізація користувацьких інтерфейсів за допомогою шаблона *Спостерігача*.

3 Поділ користувацького інтерфейсу на *Подання* і *Контролер*.

Відокремлення моделі предметної області (бізнес-логіки) додатка від призначеного для користувача інтерфейсу. Перша й основна ідея MVC полягає в тому, що будь-який користувацький додаток можна поділити на два модулі – один з яких забезпечує основний функціонал додатка, його бізнес-логіку, а другий відповідає за взаємодію з користувачем.

Таким чином, отримується можливість розробляти модель предметної області, що містить бізнес-логіку системи і є функціональним ядром програми, незважаючи на те, як саме модель буде взаємодіяти з користувачем.

Завдання взаємодії з користувачем розміщується на окремому модулі, а користувацький інтерфейс може вирішуватися порівняно незалежно.

Саме модель предметної області (доменна модель від англійського Domain Model) вважається моделлю в архітектурному MVC (звідси і термін). Тому так важливо, щоб модель була незалежною і могла незалежно розроблятися і тестуватися.

Незалежність *Моделі* і синхронізація користувацьких інтерфейсів за допомогою шаблона *Спостерігача*. Друга ключова ідея полягає в тому, що для того, щоб мати можливість розробляти *Модель* незалежно, необхідно послабити її залежність від призначеного для користувача інтерфейсу. Робиться це за допомогою шаблона *Спостерігач*.

Модель розсилає повідомлення про зміни. Інтерфейс підписується на ці оповіщення і таким чином знає, коли потрібно заново зчитати дані з моделі й оновитися. Завдяки цьому отримується майже незалежна *Модель*, яка нічого не знає про пов'язані з нею призначені для користувача інтерфейси, крім того, що вони реалізують інтерфейс *Спостерігача*.

Поділ користувацького інтерфейсу на *Подання* і *Контролер*. Третя ідея – це просто другий крок ієрархічної декомпозиції. Після первинного поділу додатка на бізнес-модель і інтерфейс, декомпозиція триває на наступному ієрархічному рівні й вже призначений для користувача інтерфейс поділяється на *Подання* і *Контролер*.

Модулі один для одного мають бути «чорними ящиками». За жодних умов один модуль не повинен звертатися до об'єктів іншого модуля безпосередньо або знати про його внутрішню структуру. Модулі повинні взаємодіяти один з одним лише на рівні абстрактних інтерфейсів

(Dependency Inversion Principle). Реалізує інтерфейс модуля зазвичай спеціальний об'єкт – *Фасад*.

Модель в MVC зовсім не подібна до доменної моделі (яка може будь-якої складності та містити безліч об'єктів), а є лише її інтерфейсом і фасадом.

Таким чином, ні *Подання*, ні *Контролер* не повинні знати про те, як влаштована модель предметної області (доменна модель), де і в якому форматі там зберігаються дані, як саме здійснюється керування. Вони взаємодіють лише з інтерфейсом і реалізують його об'єктом-фасадом, який надає всі потрібні дані в потрібному форматі й зручний набір високорівневих команд для керування підсистемою, а також реалізує шаблон *Спостерігача* для сповіщення про значні зміни в підсистемі. Якщо треба змінити базу даних, використовувати хмарне сховище або взагалі збирати потрібні дані з різних джерел у мережі, але при цьому залишити незмінним інтерфейс-фасад, то ні *Подання*, ні *Контролера* це не буде стосуватися. Ця архітектура стійка до змін (рисунок 9.4).

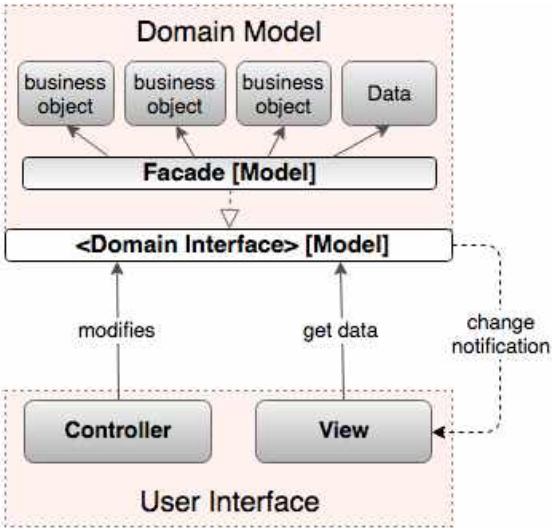

Рисунок 9.4 – Архітектура системи на основі доменної моделі

Моделі – це не дані, а виключно інтерфейси / об'єкти-посередники / фільтри (Models as Proxies, Models as Filters), що забезпечують зручний доступ до даних, які можуть міститься будь-де – на різних машинах, у різних форматах. Плутанина виникає через те, що одне і те саме слово «модель» використовується в різних контекстах. Коли йдеться про декомпозицію і відділення бізнес-логіки від призначеного для користувача інтерфейсу, то *Моделлю* дійсно вважають саме доменну модель, яка містить дані й логіку роботи з ними і забезпечує основний функціонал програми. Але в контексті шаблонів і схем *Модель* – це найперше інтерфейс, і реалізує його об'єкт-посередник (фасад, адаптер, проксі), що забезпечує зручний і безпечний доступ до доменних даних, які можуть міститися будь-де.

10 ПРОЄКТУВАННЯ НАСКРІЗНОЇ ФУНКЦІОНАЛЬНОСТІ

10.1 Питання, які потребують особливої уваги під час проєктування

Наскрізна функціональність – це аспекти дизайну, які можуть застосовуватися до всіх шарів, компонентів і рівнів. Також це ті області, у яких найчастіше робляться помилки, які мають великий вплив на дизайн. Наведемо приклади наскрізної функціональності.

Аналіз параметрів якості та наскрізної функціональності у зв'язку з наявними вимогами дає змогу зосередитися на конкретних функціональних областях. Наприклад, безпека, безсумнівно, є життєво важливим фактором під час проєктування і наявна в багатьох шарах і в багатьох аспектах архітектури. Наскрізна функціональність, що належить до безпеки, є орієнтиром, що вказує на області, на яких слід акцентувати увагу. Категорії наскрізної функціональності можуть використовуватися для поділу архітектури додатка для подальшого аналізу і виявлення «вразливих місць» докладання. Такий підхід забезпечує створення дизайну з оптимальним рівнем безпеки. Під час обговорення наскрізної функціональності, яка пов’язана з безпекою, рекомендується звернути увагу на такі питання:

1 Аудит і протоколювання. Хто, що зробив і коли? Додаток функціонує в нормальному режимі? Аудит займається питаннями реєстрації подій, пов'язаних із безпекою. Протоколювання стосується того, як додаток публікує дані про свою роботу.

2 Аутентифікація. Хто ви? Аутентифікація – це процес, під час якого одна сутність чітко й однозначно ідентифікує іншу сутність, зазвичай це робиться за допомогою таких облікових даних, як ім'я користувача і пароль.

3 Авторизація. Що ви можете робити? Авторизація визначає, як додаток керує доступом до ресурсів та операцій.

4 Керування конфігурацією. У якому контексті використовується додаток? До яких баз даних під’єднується? Як виконується адміністрування програми? Як захищені ці настройки? Керування конфігурацією визначає, як додаток реалізує ці операції та завдання.

5 Шифрування. Як реалізований захист секретів (конфіденційних даних)? Як здійснюється захист від несанкціонованого доступу до даних і бібліотек? Як передаються випадкові значення, які мають бути криптографічно стійкими? Шифрування і криптографія займаються питаннями реалізації конфіденційності та цілісності.

6 Оброблення винятків. Що робить додаток у разі невдалої спроби здійснити виклик його методу? Наскільки повні дані про помилку надаються? Чи забезпечуються зрозумілі для кінцевих користувачів повідомлення про помилки? Чи повертаються цінні відомості про винятки?

Чи виконується коректне оброблення події збою? Чи надає додаток адміністраторам необхідну інформацію для проведення аналізу основних причин збою? Оброблення винятків стосується того, як винятки обробляються в додатку.

7 Валідація вхідних даних. Як визначити, що дані, які надходять у додаток, дійсні та безпечні? Чи виконується обмеження введення через точки входу і кодування виведення через точки виходу? Чи можна довіряти таким джерелам даних, як бази даних і загальні файли? Перевірка введення стосується питань фільтрації, очищення або відхилення, які вводяться в додаток даних перед їх додатковим обробленням.

8 Конфіденційні дані. Як додаток працює з конфіденційними даними? Чи забезпечується захист конфіденційних даних користувачів і додатків? Тут вирішуються питання оброблення додатком будь-яких даних, які мають бути захищеними або під час зберігання в пам'яті, або під час передавання ПЗ мережею, або в разі зберігання в постійних сховищах.

9 Керування сеансами. Як додаток обробляє і захищає сеанси користувачів? Сеанс – це низка взаємопов'язаних взаємодій користувача і додатка.

Ці питання допоможуть ухвалити основні проєктні рішення із захисту програми і задокументувати їх як частину архітектури.

10.2 Проєктування стратегії керування винятками

Проєктування ефективної стратегії керування винятками має велике значення з погляду забезпечення безпеки і надійності додатка. Неправильний вибір стратегії дуже ускладнить діагностування та розв’язання проблем додатка, зробить його вразливим для атак на зразок «відмова в обслуговуванні» (DoS, Denial of Service), а також може привести до розголошення конфіденційних і важливих відомостей. Формування і оброблення винятків є ресурсними процесами, тому важливо, щоб під час проєктування були також враховані питання продуктивності. Вдалим підходом є проєктування централізованого механізму керування винятками для програми та надання точок доступу до системи керування винятками (таких як події WMI) для забезпечення підтримки систем моніторингу рівня підприємства, як-от Microsoft System Center.

Надійна і ретельно продумана стратегія керування винятками може спростити дизайн програми й підвищити безпеку та керованість. Вона сприятиме полегшенню завдання з розроблення програми, скороченню часу і зменшенню витрат на розроблення. Наведені далі рекомендації допоможуть правильно виробити стратегію керування винятками.

Крок 1 – вибір оброблюваних винятків. Під час проєктування стратегії керування винятками для додатка важливо визначитися з тим, які винятки потрібно обробляти. Повинні оброблятися винятки системи або додатка, які формуються в разі спроб доступу до системних ресурсів

користувачами, які не мають для цього дозволів, а також системні збої, що виникають через проблеми із жорстким диском або пам'яттю. Також необхідно позначити оброблювані винятки бізнес-логіки, тобто винятки, зумовлені такими діями, як порушення бізнес-правил.

Крок 2 – вибір стратегії виявлення винятків. Створюваний дизайн має забезпечувати однакове оброблення винятків ПЗ всім додатком. Це зробить додаток більш стійким до помилок, зменшиться ймовірність виникнення неузгодженого стану. Структуроване оброблення винятків є засобом для керування винятками за допомогою блоків try, catch і finally, їх своєчасного виявлення і відповідного реагування на них.

Перехоплювати виявлені винятки слід тільки якщо необхідно зібрати дані про винятки для протоколювання, додати у виняток деякі значущі дані, очистити ресурси, використовувані в блоці коду або повторити операцію для виходу зі стану винятку. Чи не перехоплювати винятки з подальшим їх передаванням вгору за стеком викликів, якщо не треба виконувати жодне із зазначених вище завдань.

Крок 3 – вироблення стратегії поширення винятків. Розглянемо стратегії поширення винятків. Залежно від вимог контексту додаток може (і повинен) використовувати поєднання будь-яких або всіх цих стратегій:

1 Дозволити поширення винятків. Ця стратегія є вдалою, оскільки не потребує збору даних про винятки для протоколювання, додавання даних у виняток, очищення будь-яких ресурсів у блоці коду або повтору операції для виходу зі стану винятку. Користувач дозволяє винятку розповсюджуватися вгору за стеком коду.

2 Перехоплення і повторне формування винятків. Така стратегія передбачає перехоплення, обробляння певним чином і повторне формування винятку. Зазвичай у цьому випадку дані винятку залишаються незмінними. Треба застосовувати цю стратегію, якщо потрібно очищати ресурси, протоколювати дані винятку або виконати спробу виходу зі стану помилки.

3 Перехоплення, обгортання і формування винятків. Ця стратегія забезпечує перехоплення універсальних винятків із подальшим очищенням ресурсів або будь-який інший відповідного оброблення. Якщо не вдається обробити помилку, виняток передається в інший виняток, більш доречний для сторони, яка викликає, і після цього формується новий виняток для оброблення кодом, розташованим вище в стеку коду. Слід застосовувати таку стратегію, якщо треба зберегти доречність винятку і/або забезпечити додатковими відомостями код, що обробляє виняток.

4 Перехоплення й анулювання винятків. Ця стратегія є нерекомендованою, але може застосовуватися в особливих випадках. Виняток перехоплюється, і додаток виконується у звичайному порядку. У разі потреби можна зареєструвати виняток і виконати очищення ресурсів. Така стратегія підходить для винятків системи, що не роблять впливу на інші операції, такі як винятки, що формуються під час заповнення журналу.

Крок 4 – вироблення стратегії використання власних винятків. Подумайте, чи існує необхідність у проєктуванні власних винятків або досить буде стандартних типів винятків .NET Framework. Не слід використовувати власні винятки, якщо в ієрархії винятків або в .NET Framework уже є відповідний виняток. Але створюйте власні винятки, якщо програма має виявляти й обробляти спеціальну виняткову ситуацію, щоб уникнути застосування умовної логіки, або якщо необхідно включити додаткові дані для виконання певного вимоги.

Якщо все-таки доводиться створювати власні класи винятків, слід застосовувати в них стандартні конструктори, включаючи конструктор серіалізації, і обов'язково треба закінчувати ім'я класу словом «Exception» (Виняток). Це важливо для забезпечення інтеграції зі стандартним механізмом винятків. Реалізуйте власний виняток шляхом успадкування від відповідного більш загального винятку і його спеціалізації відповідно до конкретних вимог.

Загалом під час проєктування стратегії керування винятками слід створювати ієрархію винятків і організовувати власні винятки в її межах. Це допоможе користувачам швидко аналізувати і простежувати проблеми, що виникають. У власних винятках має бути зазначено шар, у якому було сформовано виняток, компонент, у якому, можливо, виник виняток, і тип сформованого винятку (як-от виняток безпеки, бізнес-логіки або системний виняток).

Зберігайте ієрархію дозволених програм у програмі в окремій збірці, на яку може посилатися код програми. Це допоможе централізувати керування і розгортання класів винятків. Також продумайте, як буде виконуватися передавання винятків фізичними межами, межами процесів або AppDomain. Класи .NET Framework Exception підтримують серіалізацію. Під час проєктування власних класів винятків треба забезпечити, щоб вони також підтримували серіалізацію.

Крок 5 – вибір відповідних даних для збору. Один із найбільш важливих аспектів під час оброблення винятків – правильний вибір стратегії збору даних винятку. Перехоплювач відомостей повинен точно подавати умови виникнення винятку. Також відомості мають бути значущими й інформативними для аудиторії. Можна виділити три категорії аудиторії: кінцеві користувачі, розробники програми й оператори. На підставі сценарію та індивідуального контексту встановіть, кому адресовано виняток.

Для кінцевих користувачів потрібен осмислений і добре поданий опис. Під час збору даних винятків для кінцевих користувачів подбайте, щоб вони були подані у вигляді звичного для користувачів повідомлення, що описує природу помилки і пропонує шляхи відновлення після такої помилки, якщо це можливо. Розробникам додатка необхідні більш детальні відомості, які допоможуть у діагностиці проблеми.

Розробникам додатка необхідні дані про точне місце виникнення винятків в коді, а також такі відомості, як тип винятку і стан системи в момент виникнення винятку. Операторам служби підтримки має надаватися відповідна інформація, яка дасть змогу їм реагувати відповідним чином і зробити необхідні кроки з відновлення системи після помилки. Під час збору даних винятків для операторів служби підтримки забезпечте відомості, які допоможуть їм знайти людей, яких вони повинні повідомити про винятки, і інформацію, яку вони муситимуть передати для розв’язання проблеми.

Незалежно від цільової аудиторії завжди корисно забезпечувати вичерпну інформацію про винятки. Зберігайте ці відомості у файлі журналу для подальшої перевірки й аналізу частоти виникнення винятків і їх даних. За стандартною настройкою мають фіксуватися щонайменше дата і час, ім'я комп'ютера, повідомлення про винятки, дані про стек і виклики, доменне ім'я додатка, ім'я та версія збірки, ID-потоку і відомості про користувача.

Крок 6 – вироблення стратегії протоколювання винятків. Існує низка можливих варіантів протоколювання даних винятків. Наведені нижче основні рекомендації допоможуть зробити правильний вибір:

– використовуйте Windows Event Log (Журнал реєстрації подій Windows) або Windows Eventing 6.0, якщо додаток розгорнуто на одному комп'ютері, якщо необхідно використовувати наявні інструменти для перегляду журналу, або якщо надійність є основною вимогою;

– використовуйте SQL Database, якщо додаток розгорнуто у фермі або в кластері, якщо необхідно забезпечити централізоване протоколювання, або якщо потрібна гнучкість у структуруванні та протоколюванні даних винятків;

– використовуйте власний файл журналу, якщо додаток розгорнуто на одному комп'ютері, якщо необхідна абсолютна гнучкість вибору формату протоколювання, або якщо хочете просто і легко реалізувати сховище журналу реєстрації; контролюйте обсяг файлу журналу, періодично очищаючи або архівуючи журнал, щоб його обсяг не збільшувався;

– вибирайте Message Queuing як механізм доставки для передавання повідомлень про винятки в точку їх остаточного призначення, якщо надійність є головною вимогою, якщо додатки розгорнуті у фермі або в кластері, або якщо потрібно централізувати протоколювання.

У будь-якому додатку може використовуватися кілька з наведених варіантів залежно від сценарію та політики оброблення винятків. Наприклад, винятки безпеки можуть протоколюватися в Security Event Log, а винятки бізнес-логіки – у базу даних.

Крок 7 – вибір стратегії повідомлення про винятки. Під час вироблення стратегії керування винятками необхідно також ухвалити

рішення про стратегію повідомлення. Часто в корпоративних програмах одного протоколювання мало.

Необхідно передбачити також повідомлення, щоб забезпечити інформування про винятки адміністраторів і операторів, що виникають. Для цього можуть використовуватися такі технології, як події WMI, електронні листи SMTP, текстові SMS або інші системи сповіщення.

Розгляньте можливість використання зовнішніх механізмів повідомлення, таких як системи моніторингу журналів або середовища сторонніх виробників, які виявляють у даних журналу умови виникнення помилки і формують відповідні повідомлення.

Такий підхід рекомендується, якщо потрібно відокремити систему моніторингу та повідомлення від коду додатків, залишивши в програмах код протоколювання. Але якщо необхідно забезпечити негайне повідомлення без використання зовнішніх систем моніторингу, можна ввести в додаток спеціалізовані механізми повідомлення.

Крок 8 – ухвалення рішення про оброблення необроблюваних винятків. Якщо виняток залишається необробленим до останньої точки або межі, немає можливості відновлення після винятку перед поверненням керування користувачеві, програма має обробити це як необроблений виняток.

Для необроблюваних винятків потрібно зібрати необхідні відомості, записати їх у файл журналу або аудиту, розіслати всі необхідні для цього винятку повідомлення, виконати все необхідне очищення і, нарешті, передати інформацію про помилку користувачеві.

Не слід розкривати всіх деталей оброблення винятку. Треба надати користувачеві зрозуміле універсальне повідомлення про помилку. Для клієнтів без користувацького інтерфейсу, таких як вебсервіси, замість детального винятку, можна сформувати універсальний виняток. Це допоможе запобігти розголошенню даних системи кінцевому користувачеві.

10.3 Проєктування стратегії валідації введення і даних

Проєктування ефективного механізму валідації має велике значення з погляду забезпечення зручності та простоти використання і надійності додатка, а інакше воно може залишитися відкритим для неузгодженості даних і порушень бізнес-правил, а також забезпечувати незадовільний рівень взаємодії з користувачем.

Додаток може виявитися вразливим до таких загроз безпеки, як міжсайтові атаки впровадження сценаріїв, атаки на зразок упровадження SQL-коду, переповнення буфера й інші атаки за допомогою вхідних даних. Чіткого й вичерпного визначення дійсного або зловмисного введення немає.

Можливі ризики, що зумовлені зловмисним введенням, залежать також від того, як додаток використовує введення.

Наведені нижче рекомендації допоможуть правильно виробити стратегію валідації для додатка. Під час проєктування валідації введення передусім необхідно позначити межі довіри і ключові сценарії того, коли треба проводити валідацію даних. Далі визначаються дані, що підлягають валідації, і місце, де ця валідація має виконуватися.

Необхідно також знайти, як забезпечити можливість повторного використання стратегії валідації під час її реалізації. І, нарешті, визначити відповідну стратегію валідації для додатка.

Крок 1 – визначення меж довіри. Межі довіри поділяють дані з довіреного джерела і дані, до яких немає довіри. Дані, що містяться з одного боку межі довіри, є системами, яким довіряють, і з іншого боку межі розміщуються ті дані, що не мають довіри. Щоб зрозуміти, що доведеться перевіряти, необхідно найперше виявити дані, які перетинають межі довіри.

Використовуйте валідацію введення даних на кожній межі довіри, щоб скоротити загрози безпеки, такі як міжсайтові атаки впровадженням сценаріїв і впровадженням коду.

Прикладами меж довіри можуть бути міжмережевий екран, межа між вебсервером і сервером бази даних, межа між додатком і сервісом стороннього виробника.

Визначте, які системи і підсистеми будуть взаємодіяти з додатком, і зовнішні межі системи, що перетинаються під час запису даних у файли на сервері, виклику сервера бази даних або виклику вебсервісу. Позначте точки входу на межах довіри і точки виходу, у яких виконується запис даних із клієнтського введення, де спільно використовуються бази даних.

Крок 2 – визначення ключових сценаріїв. Після того, як межі довіри в додатку позначені, необхідно визначитися з ключовими сценаріями, у яких потрібна валідація даних. Усі дані, що вводяться користувачем доти, доки не пройдуть валідацію, мають розглядатися як зловмисні. Наприклад, у шарі подання вебдодатка підлягають перевірці поля форм, рядки запитів і приховані поля, параметри запитів GET і POST, завантажені дані (зловмисні користувачі можуть перехоплювати HTTP-запити і змінювати вміст) і cookies (які містяться на клієнтському комп'ютері і можуть бути змінені).

У бізнес-шарі обмеження на дані накладають бізнес-правила. Будь-яке порушення цих правил розглядається як помилка валідації, і бізнес-шар повинен сформувати виняток для подання цього порушення. Під час використання підсистеми керування правилами або робочим процесом необхідно забезпечити їм перевірку результатів кожного правила на підставі даних, яких потребує це правило, і висновків, отриманих після перевірки попередніх правил.

Крок 3 – вибір місця валідації. На цьому етапі треба визначитися з місцем проведення валідації: на клієнті або і на сервері, і на клієнті. Не слід покладатися лише на валідацію на стороні клієнта. Слід використовувати її для забезпечення більш інтерактивного UI, але завжди реалізовувати також і валідацію на стороні сервера, щоб безпечно перевірити дані в межах довіри.

Валідація даних і бізнес-правил на клієнті може зменшити кількість циклів запит-відповідь до сервера і поліпшити взаємодію з користувачем. Для вебдодатка браузер клієнта має підтримувати DHTML і JavaScript, в ідеальному варіанті реалізований в окремому файлі .js для забезпечення можливості повторного використання і кешування браузером. Найпростіший підхід у вебдодатку – застосування елементів керування валідацією ASP.NET. Це набір серверних елементів керування. У вебдодатку перевірку даних і бізнес-правил на стороні сервера може бути реалізовано з використанням елементів керування валідації ASP.NET. Як альтернативний варіант для веб- та інших типів додатків також можна використовувати Validation Application Block (блок валідації) від групи patterns & practices. Validation Application Block допоможе створити логіку валідації, придатну для повторного використання в різних шарах. Validation Application Block може застосовуватися в додатках Windows Forms, ASP.NET і WPF.

Крок 4 – вироблення стратегій валідації. Розглянемо загальні стратегії валідації даних:

1 Приймання свідомо допустимого (список дозволеного введення або позитивна перевірка). Приймаються тільки дані, що задовольняють заданим критеріям, усі інші дані відхиляються. Якщо можливо, слід дотримуватися цієї стратегії, оскільки вона забезпечує найвищу безпеку.

2 Відхилення свідомо неприпустимого (список забороненого введення або негативна перевірка). Приймаються дані, що не відповідають заданому критерію (наприклад, не містять відомий набір символів). Слід застосовувати цю стратегію обачно і тільки як другу лінію захисту, оскільки дуже складно створити вичерпний список критеріїв для всіх відомих недійсних вводів.

3 Очищення. Відомі некоректні символи або значення усуваються або перетворюються з метою зробити введення безпечним. Як і підхід зі списком забороненого введення (негативної перевірки), слід застосовувати цю стратегію обачно і тільки як другу лінію захисту, оскільки дуже складно створити вичерпний список критеріїв для всіх відомих недійсних введень.

10.4 Протоколювання й інструментування

Проєктування ефективної стратегії протоколювання й інструментування має велике значення з погляду забезпечення безпеки і

надійності додатка. Також файли журналів можуть стати в пригоді для доказу правопорушень у разі судового розгляду. Аудит і протоколювання дій у всіх шарах додатка необхідні, тому що можуть допомогти виявити підозрілі дії і забезпечити раннє виявлення серйозних атак. Аудит вважається найбільш достовірним, якщо дані аудиту формуються безпосередньо в момент доступу до ресурсу і саме тими процедурами, які виконують доступ до ресурсу. Інструментування можна реалізувати, використовуючи лічильники продуктивності та події, що забезпечують адміністраторів відомостями про стан продуктивності та працездатності програми. Під час проєктування стратегії протоколювання й інструментування слід дотримуватися таких рекомендацій:

1 Проєктувати централізований механізм протоколювання й інструментування, що забезпечує перехоплення критично важливих для системи і бізнесу подій. Уникати занадто детального протоколювання й інструментування, але передбачати додаткові функції протоколювання й інструментування, які настроюються під час виконання для отримання додаткових даних і для допомоги під час настроювання.

2 Створювати політику безпечного керування файлами журналу. Не зберігати конфіденційні дані у файлах журналу і захищати їх від несанкціонованого доступу. Продумувати, як забезпечити безпечний доступ і передавання даних аудиту й протоколювання між шарами додатка, а також стримування і правильне оброблення збоїв протоколювання.

3 Настроювати свої приймачі журналу або аналізатори трасування, щоб забезпечити можливість їх зміни під час виконання відповідно до вимог інфраструктури розгортання.

11 АУТЕНТИФІКАЦІЯ ТА АВТОРИЗАЦІЯ В МІКРОСЕРВІСНИХ ДОДАТКАХ

11.1 Складові поділу прав доступу

На процесах аутентифікації та авторизації основано поділ прав доступу.

Ідентифікація – процес визначення, що це за людина. *Аутентифікація* – процес підтвердження, що ця людина саме та, за кого себе видає. *Авторизація* – процес ухвалення рішення про те, що саме цій аутентифікованій персоні дозволяється це робити. Інакше кажучи, це три різні, послідовні та взаємно незамінні поняття. Ідентифікацію часто мають на увазі в складі аутентифікації.

Найголовніше – чітко розрізняти аутентифікацію та авторизацію.

У процесі аутентифікації можна переконатися, що людина має докази, які підтверджують її особу.

11.2 Сувора аутентифікація

З 2017 року різко зріс відсоток використання суворої аутентифікації. Зі збільшенням кількості вразливостей, які стосуються традиційних рішень для аутентифікації, організації посилюють свої можливості аутентифікації за допомогою суворої аутентифікації.

Під час використання суворої аутентифікації для перевірки автентичності користувача використовується кілька методів або факторів:

– *фактор знання*: загальний секрет між користувачем і суб'єктом перевірки автентичності користувача (наприклад, паролі, відповіді на секретні запитання тощо);

– *фактор володіння*: пристрій, яким володіє тільки користувач (наприклад, мобільний пристрій, криптографічний ключ тощо);

– *фактор невід'ємності*: фізичні (часто біометричні) характеристики користувача (наприклад, відбиток пальця, малюнок райдужної оболонки ока, голос, поведінка тощо).

Майже будь-який спосіб аутентифікації має недоліки. Однак необхідність зламати кілька чинників значно збільшує ймовірність невдачі для зловмисників, оскільки обхід або обман різних факторів потребує використання декількох типів тактик злому, для кожного фактора окремо (таблиця 11.1).

Таблиця 11.1 – Можливі варіанти аутентифікації

Аутентифікація	Фактор	Опис	Ключові вразливості
Пароль або PIN	Знання	Фіксоване значення, яке може містити букви, цифри й низку інших знаків	Може бути перехоплено, украдено, дібрано або зламано
Аутентифікація, основана на знаннях	Знання	Запитання, відповіді на які може знати тільки легальний користувач	Може бути перехоплено, дібрано, отримано за допомогою методів соціальної інженерії

Продовження таблиці 11.1

Аутентифікація	Фактор	Опис	Ключові вразливості
Апаратні OTP	Володіння	Спеціальний пристрій, що генерує одноразові паролі	Код може бути перехоплено і повторено, або пристрій може бути вкрадено
Програмні OTP	Володіння	Додаток (мобільний, доступний через браузер, або відправляє коди ПЗ e-mail), який генерує одноразові паролі	Код може бути перехоплено і повторено, або пристрій може бути вкрадено
SMS OTP	Володіння	Одноразовий пароль, що надходить за допомогою текстового SMS	Код може бути перехоплено і повторено, смартфон (або SIM-карту) може бути вкрадено, SIM-карту може бути дубльовано
Смарткарти	Володіння	Карта, яка містить криптографічний чип і захищену пам'ять із ключами, що використовує для аутентифікації інфраструктуру відкритих ключів	Може бути фізично вкрадено (але зловмисник не зможе скористатися пристроєм без знання PIN-коду)
Ключі безпеки – токени	Володіння	Пристрій з інтерфейсом USB, яке містить криптографічний чип і захищену пам'ять із ключами, що використовує для аутентифікації інфраструктуру відкритих ключів	Може бути фізично вкрадено (але зловмисник не зможе скористатися пристроєм без знання PIN-коду)

Кінець таблиці 11.1

Аутентифікація	Фактор	Опис	Ключові вразливості
Під’єднання до пристрою	Володіння	Процес, який створює профіль, часто з використанням javascript, або за допомогою маркерів, таких як cookies та Flash Shared Objects, щоб гарантувати, що використовується конкретний пристрій	Маркери може бути вкрадено (скопійовано), також характеристики легального пристрою може бути імітовано зловмисником на своєму пристрої
Поведінка	Невід'ємність	Аналізується, як користувач взаємодіє з пристроєм або програмою	Поведінку може бути імітовано
Відбитки пальців	Невід'ємність	Збережені відбитки пальців порівнюються зі зчитаними зразками, отриманими оптичним або електронним способом	Зображення може бути вкрадено і використано для аутентифікації
Сканування очей	Невід'ємність	Порівнюються характеристики очей, такі як малюнок райдужної оболонки ока, з новими сканами, отриманими оптичним способом	Зображення може бути вкрадено і використано для аутентифікації
Розпізнавання обличчя	Невід'ємність	Порівнюються характеристики особи з новими сканами, отриманими оптичним способом	Зображення може бути вкрадено і використано для аутентифікації
Розпізнавання голосу	Невід'ємність	Порівнюються характеристики записаного зразка голосу з новими зразками	Запис може бути вкрадено і використано для аутентифікації або імітовано будь-яким способом

11.3 Протоколи аутентифікації

На сьогодні застосовуються такі протоколи аутентифікації:

– OpenID 1.0 (2006), OpenID 2.0 (2007) дозволяють додаткам (app) запитувати в довіреного сервера (authority) перевірку користувача (user); відмінності між версіями для користувача несуттєві;

– OpenID Attribute Exchange 1.0 (2007) розширює OpenID 2.0, дозволяючи отримувати й зберігати профіль користувача;

– OAuth 1.0 (2010) дає змогу користувачеві дозволити додатку отримувати обмежений доступ на третіх серверах (third-party server), які довіряють засвідчувальному центру;

– OAuth 2.0 (2012) робить те саме, що і OAuth 1.0, але протокол істотно змінився і став простішим;

– OpenID Connect (2014 року) об'єднує можливості OpenID 2.0, OpenID Attribute Exchange 1.0 і OAuth 2.0 в один загальний протокол, дозволяє програмам використовувати засвідчувальний центр, щоб перевіряти облікові дані користувача й отримувати профіль користувача (або його частини).

Важливо розуміти, що OpenID Connect не дає доступ до зовнішніх ресурсів, використовуючи OAuth 2.0 для того, щоб подати параметри профілю.

11.4 Протокол OAuth 2.0

Найпоширеніший стандарт авторизації – фреймворк авторизації OAuth 2.0. Стандарт було прийнято 2012 року, і згодом протокол змінювали і доповнювали. RFC стало настільки багато, що автори оригінального протоколу вирішили написати OAuth 2.1, який об'єднає всі поточні зміни ПЗ OAuth 2.0 в одному документі. Поки протокол на стадії чернетки через велику кількість неузгодженостей. Актуальну версію OAuth описано в RFC 6749.

Особливості протоколу OAuth 2.0:

– поділ сутності користувача і додатка, що запитує доступ, завдяки цьому поділу можна керувати правами додатка окремо від прав користувача;

– замість звичних логіна і пароля, які мають певний набір прав і час дії, отримується доступ до ресурсів за допомогою випадково згенерованих рядків – токенів;

– можна видавати права максимально точково з огляду на власні побажання, а не на заздалегідь визначений набір прав.

У OAuth 2.0 визначено чотири ролі**:**

1 Resource owner – сутність, яка має права доступу на захищений ресурс. Сутність може бути кінцевим користувачем або будь-якою

системою. Захищений ресурс – це HTTP endpoint, яким може бути будь-що: API endpoint, файл на CDN, вебсервіс.

2 Resource server – сервер, на якому зберігається захищений ресурс, до якого має доступ resource owner.

3 Client – це програма, яка робить запит про доступ до захищеного ресурсу від імені resource owner і з його дозволу – з авторизацією.

4 Authorization server – сервер, який видає клієнтові токен для доступу до захищеного ресурсу після успішної авторизації resource owner.

Кожен учасник взаємодії може поєднувати в собі кілька ролей. Наприклад, клієнт може бути одночасно resource owner і запитувати доступ до своїх же ресурсів. Важливим є те, що клієнт має бути заздалегідь зареєстрований у сервісі. Схему взаємодії буде розглянуто далі.

Реєстрація клієнта. Спосіб реєстрації клієнта, наприклад ручний або service discovery, вибирається самостійно, залежно від фантазії конкретної реалізації. Але при будь-якому способі під час реєстрації, крім ID клієнта, мають бути обов'язково вказані два параметри – redirection URI і client type.

Redirection URI – це адреса, на яку відправиться власник ресурсу після успішної авторизації. Крім авторизації, адреса використовується для підтвердження, що сервіс, який звернувся за авторизацією, той, за кого себе видає.

Client type – тип клієнта, від якого залежить спосіб взаємодії з ним. Тип клієнта визначається його можливістю безпечно зберігати свої облікові дані для авторизації – токен. Тому існує всього два типи клієнтів:

– Confidential – клієнт, який може безпечно зберігати свої облікові дані, наприклад, до такого типу клієнтів належать вебдодатки, які мають backend;

– Public – клієнт не може безпечно зберігати свої облікові дані; цей клієнт працює на пристрої власника ресурсу, наприклад, це браузерні або мобільні додатки.

Токени. Токен в OAuth 2.0 – це рядок, непрозорий для клієнта. Зазвичай рядок має вигляд випадково згенерованої послідовності символів – його формат не має значення для клієнта. Токен – це ключ доступу до чого-небудь, наприклад до захищеного ресурсу (access token) або до нового токена (refresh Token).

У кожного токена свій час дії, але в refresh token токен має бути більшим, тому що він використовується для отримання access token. Наприклад, якщо термін дії access token близько години, то refresh token можна залишити діяти на цілий тиждень.

Refresh token опціональний і доступний тільки для конфіденційних (confedential) клієнтів. У деяких реалізаціях час дії access token зроблено дуже тривалим, а refresh token взагалі не використовується, щоб не ускладнювати процес оновлення, але це небезпечно. Якщо access token було скомпрометовано, його можна обнулити, а сервіс отримає новий

access token за допомогою refresh token. Якщо refresh token немає, то потрібно проходити процес авторизації знову.

За access token закріплено певний набір прав доступу, який видається клієнту під час авторизації. Розглянемо, який вигляд мають права доступу в OAuth 2.0.

Права доступу. Права доступу видаються клієнту у вигляді scope. Scope – це параметр, який складається з розділених пробілами рядків – scope-token.

Кожен із scope-token надає певні права, видатні клієнту. Наприклад, scope-token «doc_read» може надавати доступ на читання до якогось документа на resource server, а «employee» – доступ до функціонала програми лише для працівників фірми. Підсумковий scope може мати такий вигляд: «email doc_read employee».

У OAuth 2.0 самостійно створюється scope-token, що настроюється відповідно до власних потреб. Імена scope-token обмежуються тільки фантазією і двома символами таблиці ASCII – "\.

На етапі реєстрації клієнта в настройках сервісу авторизації клієнту видається стандартний scope за стандартною настройкою. Але клієнт може запросити в сервера авторизації scope, відмінний від стандартного. Залежно від політик на сервері авторизації і вибору власника ресурсу, підсумковий набір scope може мати зовсім інший вигляд. Після авторизації клієнта власник ресурсів може відібрати частину прав без повторної авторизації сервісу, але, щоб видати додаткові дозволи, потрібно повторна авторизація клієнта.

11.5 Сценарій авторизації за протоколом OAuth 2.0

Існує кілька сценаріїв застосування протоколу OAuth 2.0. Найпоширенішим із них є **абстрактний OAuth 2.0 Flow із застосуванням Access token.** Розглянемо надання доступу до сервісу Flow.

Нижче зображено абстрактну схему (або Flow) (рисунок 11.1) взаємодії між учасниками. Усі кроки на цій схемі виконуються чітко зверху вниз.

Клієнт (Client) відправляє запит на доступ до необхідного ресурсу *Власнику ресурсу* (resource owner).

Власник ресурсу передає назад *Клієнту Надання дозволу* (authorization grant), що посвідчує особу *Власника ресурсу* і його права на ресурс, доступ до якого запрошує *Клієнт*. Залежно від Flow це може бути токен або облікові дані.

Клієнт відправляє *Надання дозволу*, отримане на попередньому кроці від *Сервера авторизації* (authorization server), чекаючи від нього Access token для доступу до захищеного ресурсу.

Сервер авторизації переконується у валідності *Надання доступу*, після чого відсилає Access token клієнтові у відповідь.

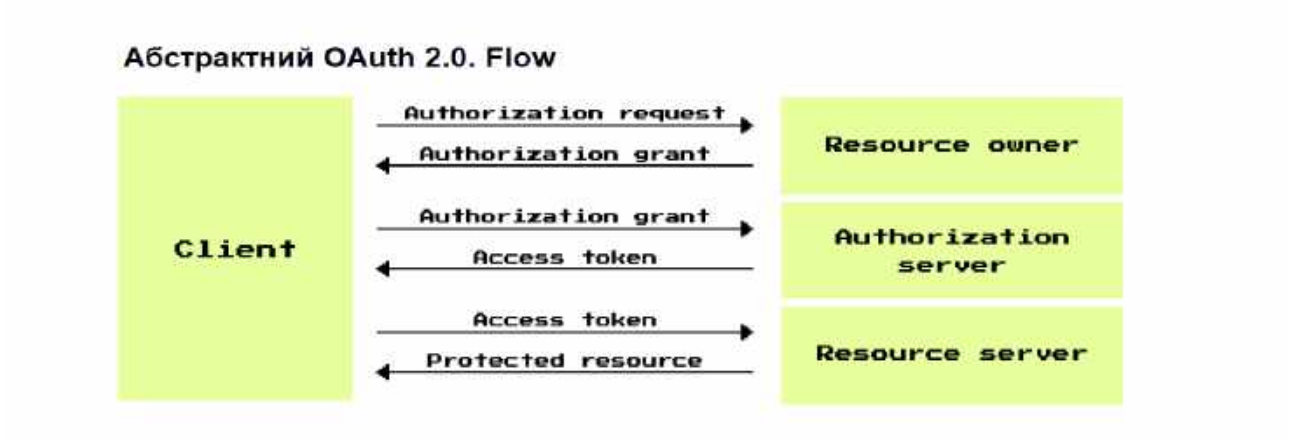

Рисунок 11.1 – Абстрактна схема OAuth 2.0. Flow із застосуванням Access token

Отримавши Access token, клієнт запитує захищений ресурс у *Ресурсу сервера*.

Ресурс сервера переконується в коректності Access token, після чого надає доступ до захищеному ресурсу.

Клієнт отримує схвалення від *Власника ресурсу*, на основі якого йому видається доступ до ресурсу.

Таким чином, протокол OAuth 2.0 регламентує процеси авторизації в розподіленій системі. Дотримання вимог цього протоколу забезпечує високий рівень безпеки та захищеності інформаційних систем.

БІБЛІОГРАФІЧНИЙ СПИСОК

Авраменко, В. С. Проектування інформаційних систем [Електронний ресурс] : навч. посіб. / В. С. Авраменко, А. С. Авраменко. – Черкаси : Черкаський національний університет ім. Б. Хмельницького, 2017. – 434 с. – Режим доступу: http://eprints.cdu.edu.ua/1481/1/pro.pdf.

Гасімов, О. Мікросервісна архітектура для початківців. Частина I [Електронний ресурс] / О. Гасімов // Globallogic. – Режим доступу: https://www.globallogic.com/ua/insights/blogs/microservices-architecture-for-beginners-part-one/.

Гасімов, О. Мікросервісна архітектура для початківців. Частина II. [Електронний ресурс] / О. Гасімов // Globallogic. – Режим доступу: https://www.globallogic.com/ua/insights/blogs/microservices-architecture-for-beginners-part-two/.

Зелінська, О. В. Інформаційні системи та технології в галузі [Електронний ресурс] : навч. посіб. / О. В. Зелінська, Н. А. Потапова, Л. О. Волонтир. – Вінниця : ВНАУ, 2020. – 263 с. – Режим доступу: https://r.donnu.edu.ua/bitstream/123456789/1817/1/%D0%9F%D0%BE%D1%81%D1%96%D0%B1%D0%BD%D0%B8%D0%BA_%D0%86%D0%A1%D1%96%D0%A2%D0%B2%D0%93_2020_%D0%97%D0%B5%D0%BB%D1%96%D0%BD%D1%81%D1%8C%D0%BA%D0%B0_%D0%9F%D0%BE%D1%82%D0%B0%D0%BF%D0%BE%D0%B2%D0%B0_%D0%92%D0%BE%D0%BB%D0%BE%D0%BD%D1%82%D0%B8%D1%80.pdf.

Ознайомлення з патерном MVC (Model-View-Controller) [Електронний ресурс] // Javarush. – Режим доступу: https://javarush.com/ua/groups/posts/uk.2536.chastina-7-oznayomlennja-z-paternom-mvc-model-view-controller.

Проектування інформаційних систем: Загальні питання теорії проектування ІС. Конспект лекцій [Електронний ресурс] : навч. посіб. для студ. спеціальності 122 «Комп’ютерні науки» / КПІ ім. Ігоря Сікорського; уклад.: О. С. Коваленко, Л. М. Добровська. – Електронні текстові дані (1 файл: 2,02 Мбайт). – Київ : КПІ ім. Ігоря Сікорського, 2020. – 192 с. – Режим доступу: https://ela.kpi.ua/bitstream/123456789/33651/1/PIS_KL.pdf.

Шаховська, Н. Б. Проектування інформаційних систем : навч. посіб. : гриф МОН України / Н. Б. Шаховська, В. В. Литвин ; М-во освіти і науки України ; за наук. ред. В. В. Пасічника. – Львів : Магнолія-2006, 2011. – 380 с.

Albano, J. Best Practices for REST API Error Handling [Electronic resource] / J. Aibano // Baeldung. – URL: https://www.baeldung.com/rest-api-error-handling-best-practices.

Ayooluwa, I. Logging Best Practices: 12 Dos and Don'ts [Electronic resource] / I. Ayooiuwa // BetterStack. – URL: https://betterstack.com/community/guides/logging/logging-best-practices/.

Bass, L. Software Architecture in Practice [Electronic resource] / L. Bass, P. Clements, R. Kazman. – 3rd Edition. – Addison-Wesley Professional,

2012. – 624 p. – URL: https://edisciplinas.usp.br/pluginfile.php/5922722/mod_resource/content/1/2013%20-%20Book%20-%20Bass%20%20Kazman-Software%20Architecture%20in%20Practice%20%281%29.pdf.

Bourque, P. Guide to the Software Engineering Body of Knowledge (SWEBOL Guide V 3.0) [Electronic resource] / P. Bourque, R. E. Fairley – Piscataway, NJ : IEEE and IEEE Computer Society, 2014. – 335 p. – URL: https://www.computer.org/education/bodies-of-knowledge/software-engineering.

Hernandez, R. The Model View Controller Pattern – MVC Architecture and Frameworks Explained [Electronic resource] / R. Hernandez // Free Code Camp. – URL: https://www.freecodecamp.org/news/the-model-view-controller-pattern-mvc-architecture-and-frameworks-explained/.

Rainer, R. Kelly. Introduction to information systems [Electronic resource] / R. K. Rainer, B. Price. – Hoboken, NJ : Wiley, 2021. – 608 p. – URL: https://books.google.com.ua/books?id=byY_EAAAQBAJ&printsec=frontcover&hl=uk#v=onepage&q&f=false.

Recker, J. Scientific research in information systems: A beginner's guide / J. Recker. – Cham : Springer Nature, 2021. – 221 p. – DOI: 10.1007/978-3-030-85436-2.

Richards, M. Software Architecture Patterns [Electronic resource] / M. Richards. – 2nd edition. – Sebastopol : O’Reilly Media, 2022. – 73 p. – URL: https://storage.pardot.com/1009792/1685141235Z8bRQIn3/Software_Architecture_Patterns.pdf.

Software Architecture: the Hard Parts [Electronic resource] / N. Ford, M. Richards, P. Sadalage, Z. Dehghani. – 1st edition. – Sebastopol : O’Reilly Media, 2021. – 459 p. – URL: https://dl.ebooksworld.ir/books/Software.Architecture.The.Hard.Parts.Neal.Ford.OReilly.9781492086895.EBooksWorld.ir.pdf.

Zosym, M. SWEBOK v3 [Electronic resource] / M. Zosym // MAX ZOSYM. – URL: https://www.maxzosim.com/swebok-v3/.

Zmerzlyi, I. Мікросервісна архітектура [Electronic resource] / I. Zmerziy // Medium. – URL: https://medium.com/@IvanZmerzlyi/microservices-architecture-461687045b3d.

Understanding Service-Oriented Architecture (SOA): A systematic literature review and directions for further investigation, Information Systems / N. Naghmeh, I. Waidah, G. Imran, N. Behzad, B. Mahadi, A. R. B. C. Hussin // Information Systems / Elsevier. – 2020. – Volume 91. – DOI: 10.1016/j.is.2020.101491.

Waswani, N. Microservices Architecture, The Hard Parts : Resilient Patterns [Electronic resource] / N. Waswani // Medium. – URL: https://medium.com/@waswani/microservices-architecture-the-hard-parts-resilient-patterns-c4bcfe568458.

Навчальне видання

Яшина Олена Сергіївна
Пісклова Тетяна Сергіївна

ПРОЄКТУВАННЯ ІНФОРМАЦІЙНИХ СИСТЕМ

Редактор А. Г. Литвин

Зв. план, 2024
Підписано до видання 04.07.2024
Ум. друк. арк. 3,8. Обл.-вид. арк. 4,25. Електронний ресурс

Видавець і виготовлювач
Національний аерокосмічний університет ім. М. Є. Жуковського
«Харківський авіаційний інститут»
61070, Харків-70, вул. Чкалова, 17
http://www.khai.edu
Видавничий центр «ХАІ»
61070, Харків-70, вул. Чкалова, 17
izdat@khai.edu

Свідоцтво про внесення суб'єкта видавничої справи
до Державного реєстру видавців виготовлювачів і розповсюджувачів
видавничої продукції сер. ДК № 391 від 30.03.2001