УДК 004.4'2 DOI https://doi.org/10.35546/kntu2078-4481.2026.2.50

Т. В. ФІЛІМОНЧУК
кандидат технічних наук, доцент,
доцент кафедри електронних обчислювальних машин
Харківський національний університет радіоелектроніки
ORCID: 0000-0002-4380-504X

Є. І. ЗОЛОТАЙКО
магістрант кафедри електронних обчислювальних машин
Харківський національний університет радіоелектроніки
ORCID: 0009-0009-4688-9857

О. М. СЕВОСТЬЯНОВА
старший викладач кафедри електронних обчислювальних машин
Харківський національний університет радіоелектроніки
ORCID: 0009-0008-2595-5133

МОДЕЛЬ АРХІТЕКТУРИ ВЕБЗАСТОСУНКУ НА ОСНОВІ СУЧАСНИХ БАГАТОШАРОВИХ АРХІТЕКТУР

У роботі досліджено актуальну задачу оновлення класичних багатошарових архітектур вебзастосунків з огляду на зростання вимог до темпів розробки, гнучкості та масштабованості сучасних програмних рішень. Автори здійснили ґрунтовний аналіз традиційних підходів (N-layered, Onion, Clean Architecture) та визначили їхні ключові обмеження в умовах динамічного розвитку проєктів. Особливу увагу приділено проблемі «вертикальної залежності», за якої зміни в одній бізнес-функції потребують коригування коду на всіх горизонтальних рівнях, що спричиняє надмірну зв'язність та ускладнення логіки системи.

Запропоновано удосконалену модель архітектури вебзастосунку, яка охоплює презентаційний, прикладний, доменний та інфраструктурний рівні, поперечні компоненти, спеціалізовані механізми валідації та інтеграцію сучасних архітектурних патернів (CA, VSA, CQRS). У роботі детально описано наповнення кожного рівня: від застосування SSR та CSR у презентаційному шарі до використання брокерів повідомлень та кешування в інфраструктурі. Центральне місце в запропонованій моделі займає впровадження принципів Domain-Driven Design (DDD) та підходу Rich Domain Model, що забезпечує інкапсуляцію бізнес-правил безпосередньо в доменних сутностях та запобігає виникненню некоректних станів системи. Також розглянуто концепцію вертикальних зрізів, у межах якої кожна бізнес-функція реалізується як автономний модуль, що знижує взаємозалежність між компонентами.

Практична ефективність запропонованої моделі підтверджена низкою прикладів. Зокрема, застосування гібридного підходу дало змогу скоротити час розгортання інфраструктури з трьох тижнів до одного. Розробка нових функцій прискорюється до 60%, а внесення змін у наявний код відбувається на 70% швидше порівняно з традиційними архітектурними моделями. Окрім цього, завдяки використанню принципу CQRS вдалося підвищити продуктивність високонавантажених операцій читання до 50%, зменшивши час відповіді із 400 мс до 150-200 мс. Отже, запропонована модель зберігає переваги класичного багатошарового підходу: структурованість коду, зручність модульного тестування та простоту масштабування, водночас забезпечуючи більшу адаптивність та гнучкість архітектури в умовах сучасних вимог.

Ключові слова: *вебзастосунок, Clean Architecture, Vertical Slice Architecture, CQRS, Onion Architecture, Multi-Tier Architecture, Domain-Driven Design, Rich Domain, масштабованість, модульне тестування, структурованість коду.*

T. V. FILIMONCHUK
PhD in Engineering, Associate Professor,
Associate Professor at the Department of Electronic Computing Machines
Kharkiv National University of Radioelectronics
ORCID: 0000-0002-4380-504X

Y. I. ZOLOTAIKO
Graduate Student at the Department of Electronic Computing Machines
Kharkiv National University of Radioelectronics
ORCID: 0009-0009-4688-9857

ISSN 2078-4481

O. M. SEVOSTIANOVA
Senior Lecturer at the Department of Electronic Computers
Kharkiv National University of Radioelectronics
ORCID: 0009-0008-2595-5133.

MODEL OF WEBAPPLICATION ARCHITECTURE BASED ON MODERN MULTILAYERED ARCHITECTURES

The paper explores the relevant task of updating classic multi-layered web application architectures in view of the increasing requirements for development speed, flexibility and scalability of modern software solutions. The authors have conducted a thorough analysis of traditional approaches (N-layered, Onion, Clean Architecture) and identified their key limitations in the context of dynamic project development. Attention is paid to the problem of "vertical dependency", in which changes in one business function require code adjustments at all horizontal levels, which causes excessive coherence and complexity of the system logic.

An improved web application architecture model is proposed, which covers the presentation, application, domain and infrastructure levels, cross-components, specialized validation mechanisms and integration of modern architectural patterns (CA, VSA, CQRS). The paper describes in detail the content of each level: from the use of SSR and CSR in the presentation layer to the use of message brokers and caching in the infrastructure. The central place in the proposed model is occupied by the implementation of the principles of Domain-Driven Design (DDD) and the Rich Domain Model approach, which ensures the encapsulation of business rules directly in domain entities and prevents the occurrence of incorrect system states. The concept of vertical slices is also considered, within which each business function is implemented as an autonomous module, which reduces the interdependence between components.

The practical effectiveness of the proposed model is confirmed by several examples. In particular, the use of a hybrid approach made it possible to reduce the infrastructure deployment time from three weeks to one. The development of new functions is accelerated by up to 60%, and changes to existing code are made 70% faster compared to traditional architectural models. In addition, thanks to the use of the CQRS principle, it was possible to increase the performance of high load read operations by up to 50%, reducing the response time from 400 ms to 150-200 ms. Therefore, the proposed model retains the advantages of the classic multi-layer approach: structured code, ease of unit testing, and ease of scaling, while providing greater adaptability and flexibility of the architecture in the face of modern requirements.

Key words: *web application, Clean Architecture, Vertical Slice Architecture, CQRS, Onion Architecture, Multi-Tier Architecture, Domain-Driven Design, Rich Domain, scalability, unit testing, code structure.*

Постановка проблеми

Розвиток вебзастосунків вирізняється зростанням вимог до швидкості розробки, масштабованості та легкості в підтримці. На ці параметри впливає архітектура, що використовується при розробці продукту. Протягом останніх років велику популярність мали багатошарові архітектури, до яких відносяться N-layered Architecture, Onion Architecture, Clean Architecture та інші. Такий підхід має на меті добитися чіткого розділення відповідальності, він ефективно слугував галузі, особливо на початкових етапах розвитку монолітних систем.

Незважаючи на очевидні переваги такого підходу до розробки вебзастосунків, класична багатошарова архітектура демонструє значні недоліки в умовах сучасних проєктів, що відрізняються динамічністю:

– проблему вертикальної залежності та зв'язаності: при внесенні змін, що стосуються певної бізнес-функції, розробник часто змушений модифікувати код у кожному шарі (контролер, сервіс, репозиторій, модель даних), що призводить до високої зв’язності та громіздкого шару бізнес-логіки;

– складність масштабування та розгортання: у монолітних застосунках, що побудовані на багатошаровій моделі, масштабування відбувається на рівні всього застосунку, навіть якщо лише невелика частина функціоналу потребує більшої обчислювальної потужності, що призводить до неефективного використання ресурсів;

– низька зрозумілість бізнес-функцій: код, який стосується однієї бізнес-функції, є розділеним по горизонтальних шарах, що ускладнює новим членам команди швидке розуміння конкретного функціоналу та його незалежну підтримку.

Одним з рішень цих проблем є використання елементів Vertical Slice Architecture (VSA), що пропонує альтернативний підхід до організації кодової бази, звертаючи увагу в першу чергу на вертикальну сегментацію коду за окремими бізнес-функціями (Features/Use Cases), а не за технічними шарами. Кожен вертикальний зріз є автономною одиницею, що містить усі необхідні компоненти для реалізації однієї функції. Таким чином, кожну частину функціоналу можна розвивати незалежно одна від одної, що дозволяє використовувати техніки та технології, які найкраще підходять для конкретного функціоналу.

Грамотне злиття ідей VSA з багатошаровою архітектурою дозволить отримати і вище зазначені переваги VSA, і переваги багатошарових архітектур, дозволяючи зберегти код в рамках однієї бізнес-функції структурованим та малозв’язним.

Аналіз останніх досліджень і публікацій

У межах дослідження сучасних архітектур вебзастосунків треба звернути увагу на наукові роботи та статті, що вивчають актуальні багатошарові архітектури, та ті, що висвітлюють особливості Vertical Slice Architecture. Аналіз таких багатошарових архітектур, як N-Layered Architecture, Onion Architecture та Clean Architecture дозволить обрати ту з них, що найкраще підходить до поєднання з VSA.

Робота [1] обґрунтовує необхідність переходу від монолітних структур до багаторівневих (multi-tier) для вирішення проблем масштабованості та складності управління кодом у сучасних вебсистемах. Автор виділяє ключові рівні: інтерфейс користувача, бізнес-логіку та доступ до даних, підкреслюючи, що їхня незалежність дозволяє вносити зміни в одну частину системи без ризику для інших. Робота детально описує етапи проєктування, від аналізу вимог до розгортання, і наводить приклади таких компаній як Netflix та Uber, які використовують ці принципи для підтримки високої відмовостійкості та гнучкості.

Стаття [2] представляє концепцію Onion Architecture, де центральним елементом є доменна логіка, що захищена концентричними шарами інфраструктури та представлення. Головним принципом тут є правило залежностей: зовнішні шари можуть залежати лише від внутрішніх, що робить ядро системи незалежним від конкретних баз даних чи фреймворків. Автор наголошує на перевагах високої тестованості та простоти супроводу коду, хоча й зауважує, що для малих проєктів така структура може бути надмірно складною.

Робота [3] зосереджена на фундаментальних принципах багаторівневого дизайну (MVC, клієнт-сервер, мікросервіси) та їхній здатності підвищувати якість розробки програмного забезпечення. Автор детально описує функції презентаційного шару, шару бізнес-логіки та доступу до даних, пояснюючи, як поділ відповідальності зменшує складність систем. Стаття стверджує, що правильна ієрархія шарів дозволяє розробникам незалежно тестувати та вдосконалювати компоненти, що забезпечує довгострокову стійкість та масштабованість програмного продукту.

Стаття [4] присвячена практичному застосуванню принципів Роберта Мартіна для розробки сучасних застосунків на платформі.NET. Автори акцентують на важливості інверсії залежностей та незалежності бізнес-логіки від зовнішніх фреймворків та баз даних. Робота демонструє структуру рішення, де ядром є домен і використання Clean Architecture для чіткого поділу шарів, що забезпечує високу масштабованість та легкість тестування системи.

Робота [5] базується на опитуванні розробників щодо їхньої обізнаності з багаторівневими структурами (презентація, контролер, репозиторій, модель) та інструментами тестування. Автор підтверджує, що розуміння архітектурних шарів, зокрема шаблону MVC, є критичним для створення якісних та візуально привабливих вебсторінок. Стаття також аналізує популярність сучасних інструментів розробки, таких як Visual Studio Code, JUnit та Apache JMeter, роблячи висновок про важливість системного підходу до програмування та тестування у вебсередовищі.

Дослідження [6] присвячено архітектурі модульного моноліту як «золотій середині», що поєднує швидкість розробки моноліту із масштабованістю мікросервісів. Автори на основі огляду літератури визначають його як систему з чітко розмежованими доменами та слабко пов’язаними модулями, які за необхідності можуть бути легко розгорнуті як окремі мікросервіси. У статті розглядаються такі інструменти як Google Service Weaver та Spring Modulith, а також успішні кейси використання (наприклад, Shopify), де модульний моноліт став ефективною альтернативою складним мікросервісним екосистемам.

Робота [7] присвячена розв'язанню проблем гнучкості та відповідності програмного забезпечення бізнес-вимогам, що виникають при традиційному підході, де ядром системи є база даних. Автори обґрунтовують переваги патерну Domain-Driven Design (DDD), який фокусується на об'єктивній реальності предметної області та забезпечує узгодженість між аналізом, дизайном та реалізацією. Така структура дозволяє чітко розділити відповідальність, забезпечити низьку зв’язність модулів та ефективно масштабувати систему відповідно до змінних потреб бізнесу, використовуючи такі ключові елементи, як сутності, об’єкти-значення та агрегати.

Робота [8] надає порівняльний аналіз двох провідних архітектурних підходів у екосистемі.NET: Clean Architecture та VSA. Автор розглядає Clean Architecture як еволюцію багатошарового дизайну, де головний акцент робиться на інверсії залежностей та ізоляції ядра домену від інфраструктури, забезпечуючи високу стабільність та зручність в тестуванні системи в довгостроковій перспективі. Натомість VSA орієнтована на функціональну декомпозицію, де кожна бізнес-вимога реалізується як автономний наскрізний потік (зріз), що значно прискорює доставку нових функцій та спрощує паралельну роботу команд. Ключовим висновком дослідження є те, що ці парадигми не є взаємовиключними, а скоріше доповнюють одна одну на різних етапах життєвого циклу продукту: вертикальні зрізи домінують на ранніх стадіях для швидкої адаптації, тоді як роль «чистої архітектури» зростає разом із складністю продукту для підтримки цілісності рішень. Найбільш ефективною стратегією для великих систем автор вважає гібридну конфігурацію, де Clean Architecture визначає системне ядро та стандарти взаємодії, а Vertical Slice Architecture використовується для розробки периферійних модулів, що дозволяє збалансувати швидкість змін та стійкість архітектури.

На основі аналізу робіт по темі можна вивести узагальнену модель типової багатошарової архітектури:

$$MLA = \{ PL, BL, DA, INF \}, \quad (1)$$

де PL – шар презентації (інтерфейс користувача або шар ендпоінтів); BL – бізнес-логіка (сервіси для обробки даних, валідатори та ін.); DA – шар доступу до даних (зазвичай класи для отримання за запису даних в базу даних); INF – зовнішня інфраструктура (підключення до бази даних, зовнішні сервіси).

Модель традиційної багатошарової архітектури має великі переваги, такі як чітка структура коду та розділення кодової бази на вузьконаправлені частини, що в загальному випадку допомагає із підтримкою програмного продукту, а також з тестуванням окремих частин його функціоналу. Але така модель вимагає абсолютно однакової структури від кожної частини кодової бази, що може викликати проблеми, якщо різні частини Use Cases вимагають різної кількості шарів, бо тобі у ліпшому випадку, розробник буде вимушений писати зайвий код для створення шару, що непотрібен для даного Use Case, а у гіршому – кількості шарів буде недостатньо, і доведеться додавати до існуючих шарів код, що логічно до них не відноситься. Такі випадки можуть призводити до підвищення хаосу та неоднородностей в кодовій базі, що вже навпаки шкодить подальшій підтримці та ясності коду.

Приймаючи до уваги можливі проблеми, що йдуть із використанням класичних багатошарових архітектур, актуальним стає створення модифікованої моделі архітектури, яка буде відрізнятися гнучкістю та динамічністю без шкоди масштабованості.

Формулювання мети дослідження

Метою дослідження є модернізація класичної багатошарової архітектури шляхом інтеграції сучасних архітектурних патернів, таких як Clean Architecture (CA), Vertical Slice Architecture (VSA) та CQRS. Основна задача полягає у створенні моделі, яка зберігає ключові переваги традиційного багатошарового підходу: чітку організацію коду, можливість ефективного впровадження модульного тестування та легкість масштабування застосунку, водночас підвищуючи гнучкість та динамічність архітектури.

Викладення основного матеріалу дослідження

На основі аналізу сучасних багатошарових архітектур можна дійти висновку, що багато описаних вище проблем виникають через фокус цих архітектур тільки на розділенні кодової бази на горизонтальні шари, у кожному з яких код має бути однообразним для усіх частин функціоналу. Такий підхід є ефективним, якщо різні бізнес-функції мають схожі або однакові потреби до своєї структури, але викликає проблеми, якщо новий Use Case має інші потреби. В такому випадку внесення потрібних змін до структури коду цього Use Case створить неоднорідність в структурі шару. Такі неоднорідності часто включають наявність в шарі коду, який логічно до нього не відноситься, що шкодить якості коду та йде в розріз із ідеями багатошарових архітектур. Внесення змін у вже сталу структуру також є проблематичним, бо вони потребують великих додаткових витрат часу від розробників, через що дуже невеликі зміни в потребах Use Case вимагатимуть великих затрат часу.

Враховуючи недоліки багатошарових архітектур, запропоновано наступну модель гібридної архітектури вебзастосунку:

$$M = \{ PL, AL, DL, IL, VAL, CC, CSA \}, \quad (2)$$

де PL (Presentation Layer) – презентаційний рівень; AL (Application Layer) – прикладний рівень; DL (Domain Layer) – доменний рівень; IL (Infrastructure Layer) – інфраструктурний рівень; VAL (Validation) – механізми валідації; CC (Cross-Cutting components) – поперечні компоненти; CSA (Clean Slices Architecture) – гібридна архітектура.

Презентаційний шар (PL) – це інтерфейсний рівень, який відповідає за взаємодію із зовнішнім світом. Він складається з наступних компонентів:

$$PL = \{WebUI, Web API, Endpoints \{REST, GraphQL\}\}, \quad (3)$$

де Web UI (Web User Interface) – це інтерфейс вебзастосунку; Web API (Web Application Programming Interface) – зв'язок між сервером та браузером; Endpoints – основна частина API; REST (Representational State Transfer) –архітектурний стиль побудови API; GraphQL – мова запитів.

Web UI – це інтерфейс сайту або вебзастосунку, який користувач бачить у браузері. На даний час існує два популярні способи його побудови: SSR (Server-Side Rendering) та CSR (Client Side Rendering), який використовується в підході SPA (Single Page Application).

Web API – це інтерфейс між вебсервером та веббраузером, який складається з одного або більше публічних Endpoints, що створюють чітко визначену систему типу «запит – відповідь».

Endpoints – це основна частина будь-якого API (REST API, GraphQL тощо). Endpoint виступає цифровою локацією, на яку API приймає запити від клієнта (у випадку даної моделі Web UI) на отримання ресурсів.

REST – це архітектурний стиль побудови API. Його принципи включають чітке розділення клієнту та серверу, відсутність стану (кожен запит повинен мати всю інформацію для його виконання), кешованість, шарову структуру (між клієнтом та сервером може бути один або декілька проміжних шарів, про які вони обидва не мають знати) та уніфікований інтерфейс для різних платформ.

GraphQL – мова запитів із відкритим вихідним кодом, як більш гнучка альтернатива REST API. Цей підхід має на меті вирішити такі проблеми статичної структури REST API, як Over-fetching (вибірка непотрібних даних разом із потрібними) та Under-fetching (недостатність даних з одного Endpoint, що змушує робити по декілька запитів до API).

Прикладний рівень (AL) реалізує сценарії використання (Use Cases) та має наступну структуру:

$$AL = \{ C, Q, M, H \}, \quad (4)$$

де C (Command) – команда, зміна стану, запит на дію; Q (Query) – запит на отримання даних; M (Mapping) – перетворення між об'єктами; H (Handler) – обробник Command або Query.

Команда (C) є будь-якою дією, що змінює стан системи і вона може мати в собі складну бізнес-логіку та зазвичай не повертає даних, за виключенням статусу виконання операції або ідентифікатора новоствореного запису.

Запит (Q) є операцією читання, який не змінює стан системи, натомість може повертати велику кількість даних.

Компонент перетворення (M) відповідає за перетворення даних між різними рівнями. Він дозволяє у зручний спосіб приводити дані до виду, що відповідають потребам конкретного рівня.

Обробник (H) – це компонент, що займається обробкою команди або запиту, він збирає воєдино всі елементи прикладного рівня. Обробник за допомогою компоненту (M) перетворює дані у модель, що відповідає потребам доменного рівня (DL), виконує команду або запит, в залежності від конкретного Use Case, після чого перетворює отримані дані або результат дії у вигляд, що відповідає потребам презентаційного рівня (PL).

DL (Domain Layer) – ядро системи, найважливіший шар, що не має залежати від жодного іншого шару. У якості структури для даного рівня було обрано принципи DDD та Rich Domain, які зміщують фокус з технологій на бізнес-логіку. Основа підходу DDD полягає в ясному описі кодом бізнес-процесів, щоб підвищити якість та ясність коду, що описує складну логіку та взаємодію багатьох сутностей. Такий результат досягається за допомогою використання Rich Domain Model.

Суть Rich Domain Model полягає в тому, що бізнес-правила та логіка описуються в самих сутностях домену замість їх винесення в зовнішні сервіси, як це відбувається в класичних багатошарових архітектурах. Збереження бізнес-правил в сутностях означає, що сутність не може бути переведена в некоректний стан у зовнішніх компонентах, бо опис усіх обмежень та коректних станів відбувається в самій сутності. Структуру домену з використанням Rich Domain Model можна представити наступним чином:

$$DL = \{ EN, VO, AG, DS, DE, BR, RA \}, \quad (5)$$

де EN (Entities) – сутності домену; VO (Value Objects) – об'єкти-значення; AG (Aggregates) – агрегати; DS (Domain Services) – сервіси домену; DE (Domain Events) – події домену; BR (Business Rules) – бізнес-правила; RA (Repository Abstractions) – абстракції та контракти для роботи з даними.

Сутності (EN) є основою Rich Domain Model, що інкапсулюють в собі бізнес-правила. Кожна сутність є об'єктом, яка співвідноситься із однією концепцією з домену (користувач, замовлення) та має унікальний ідентифікатор, що використовується для перевірки двох сутностей на рівність.

Об'єкти-значення (VO) не мають ідентифікатора і описуються своїми полями. Вони описують невеликі концепти (наприклад, об'єкт «гроші», що має в собі поля для суми та валюти) і вони незмінні, бо імітують поведінку примітивних типів. Призначення VO полягає в зручному описі невеликих частин сутностей, що не можна описати одним полем, але вони мають працювати як одне ціле.

Агрегати (AG) є кластерами об'єктів (сутностей та об'єктів-значень), що розглядаються як єдине ціле в рамках бізнес-функції. В агрегаті один об'єкт (зазвичай сутність) завжди є його коренем (Aggregate Root). Будь-яка взаємодія з агрегатом відбувається тільки через його корінь. Прикладом агрегату є об'єкт замовлення в магазині, що має в собі колекцію об'єктів товарів, логіку, пов'язану з оплатою, знижками та ін. Агрегати розраховані на атомарні операції, в одній бізнес-функції має бути не більше пари агрегатів.

Сервіси домену (DS) описують бізнес-логіку, яка належить предметній області, але не може бути природно розміщена всередині однієї конкретної сутності чи агрегату. У доменно-орієнтованому проєктуванні кожна сутність відповідає за власний стан і поведінку, проте інколи виникають сценарії, що охоплюють кілька об'єктів одночасно. У таких випадках логіка не «належить» жодному з них окремо, тому її виділяють у сервіс домену.

Наприклад, переказ коштів між двома рахунками не є поведінкою лише одного рахунку, адже операція змінює стан одразу двох агрегатів та містить правила, які стосуються їх взаємодії: перевірку достатності балансу, обмеження за лімітами, можливі комісії або додаткові умови. Поміщати всю цю логіку в один із рахунків було б концептуально неправильно, оскільки жоден із них не «контролює» інший. Саме тому створюється окремий сервіс домену, який координує цю взаємодію та інкапсулює відповідні бізнес-правила. Такий сервіс залишається частиною доменної моделі, він описує саме бізнес-поведінку, а не технічні аспекти. Він не зберігає власний стан і не працює з базою даних чи мережними викликами. Його задача – виразити дію мовою предметної області та

забезпечити коректне застосування правил. Водночас він відрізняється від прикладного сервісу, який більше відповідає за організацію сценарію виконання, керування транзакціями або взаємодію з інфраструктурою.

Отже, сервіс домену з'являється тоді, коли потрібно формалізувати важливу бізнес-операцію, що відбувається між кількома об'єктами й не може бути логічно закріплена за одним із них. Він допомагає зберегти чистоту моделі та зробити правила предметної області явними й зрозумілими.

Події домену (DE) є механізмом, який дозволяє повідомити про суттєві зміни в одній частині домену іншим частинам системи. Вони фіксують факт того, що в предметній області вже щось відбулося, і роблять цю подію явною для інших компонентів моделі. Подія не описує намір чи команду, а саме завершений факт, який має значення з точки зору бізнесу.

Коли в системі відбувається важлива зміна стану, наприклад створення замовлення, підтвердження оплати або скасування бронювання, це може породжувати подію домену. Така подія відображає бізнесову реальність та формулюється мовою предметної області. Вона стає способом повідомити інші частини системи, що певний факт вже відбувся, і що вони можуть на нього відреагувати відповідно до своїх правил.

Події домену допомагають зменшити зв'язність між різними частинами системи. Замість того щоб один компонент безпосередньо викликав інший і знав про його внутрішню логіку, він просто публікує подію. Інші частини домену можуть підписатися на неї та виконати необхідні дії, не створюючи жорсткої залежності між собою, що дозволяє системі залишатися гнучкою та розширюваною. Крім того, події домену підкреслюють важливість часу в моделі, вони завжди описують щось, що сталося в минулому, і часто містять дані, пов'язані з цим фактом. Завдяки цьому модель стає більш виразною, а бізнес-процеси – прозорішими. Таким чином, події домену не лише забезпечують механізм повідомлення про зміни, а й допомагають краще структурувати та осмислити саму предметну область.

Бізнес-правила (BL) є чіткими директивами, що визначають обмеження, стандарти та логіку прийняття рішень у межах предметної області. Вони формують основу поведінки системи, адже саме через них відображаються вимоги бізнесу до того, як повинні оброблятися дані, які дії дозволені або заборонені, за яких умов можливі певні операції та як мають виконуватися обчислення. Фактично бізнес-правила перетворюють абстрактні вимоги на конкретну формалізовану логіку.

Зазначені правила визначають допустимі межі значень, встановлюють залежності між параметрами, описують умови переходу об'єктів із одного стану в інший або задають алгоритми розрахунків. Вони забезпечують цілісність і послідовність моделі, запобігаючи виникненню некоректних станів і гарантують, що система поводиться відповідно до очікувань бізнесу. Без чітко визначених правил доменна модель втратила б зміст, оскільки дані існували б без контролю за тим, як і коли їх можна змінювати.

Носіями бізнес-правил є сутності домену (EN), адже саме вони відповідають за власний стан та його коректність. Сутність не лише зберігає дані, а й інкапсулює логіку, яка визначає допустимість змін і поведінку в різних ситуаціях. Таким чином, бізнес-правила не існують окремо від моделі, а вбудовані безпосередньо в її структуру, що дозволяє уникнути розпорошення логіки по різних частинах системи та забезпечує узгодженість між даними й правилами їх використання.

У результаті бізнес-правила стають центральним елементом доменної моделі, оскільки саме вони відображають реальні процеси та обмеження предметної області. Вони роблять систему передбачуваною, зрозумілою й відповідною вимогам бізнесу, перетворюючи програмну реалізацію на точне відображення реальних процесів та рішень.

Компонент абстракцій репозиторіїв (RA) представляє собою набір контрактів для отримання та збереження даних, які реалізуються на інфраструктурному рівні (IL). Тут мова йдеться про абстракції, що визначають спосіб взаємодії доменної моделі з механізмами зберігання, не розкриваючи деталей конкретної технології. Іншими словами, RA формує інтерфейс доступу до даних, який описує, що саме можна отримати або зберегти, але не пояснює, яким чином це відбувається технічно.

Такі контракти дозволяють доменному шару залишатися незалежним від баз даних, файлових систем чи зовнішніх сервісів. Домен працює з абстрактними сховищами, сприймаючи їх як частину моделі, тоді як фактична реалізація доступу до даних відбувається на інфраструктурному рівні. Це означає, що зміна технології зберігання або способу інтеграції з зовнішніми системами не впливає на бізнес-логіку, оскільки вона взаємодіє лише з визначеними контрактами.

Таким чином, складова RA моделі виконує роль межі між доменом та технічною реалізацією. Вона забезпечує чітке розділення відповідальностей: домен визначає, які дані йому потрібні та в якому вигляді, а інфраструктурний рівень відповідає за те, як ці дані фізично зчитуються або зберігаються. Такий підхід сприяє чистоті архітектури, покращує тестованість системи та підтримує принципи інверсії залежностей, оскільки домен залежить лише від абстракцій, а не від конкретних реалізацій.

IL є інфраструктурним рівнем застосунку, його призначення – технічна реалізація залежностей. Даний шар складається з наступних компонентів:

$$IL = \{ DB, ORM, R, OAPI, FS, CS, MR \}, \quad (6)$$

де DB – база даних; ORM (Object-Relation Mapping) – компонент перетворення відношень бази даних до типів ООП; R (Repositories) – реалізації репозиторіїв згідно з контрактами з рівня DL; OAPI – зовнішні API; FS (File Storage) – система зберігання файлів; CS (Cache Server) – система зберігання кешу; MR (Message Broker) – компонент передачі повідомлень між частинами розподіленої системи.

Компонент DB є головним механізмом постійного зберігання даних у вебзастосунках. На даний час бази даних (БД) поділяються на:

– реляційні або SQL бази даних (SQL Server, PostreSQL, MySQL та ін.), які зберігають дані в таблицях, що створює чітку структуру: кожен запис є рядком таблиці, а колонки таблиці є його атрибутами. Зазначені БД є найпопулярнішим видом бази даних для вебзастосунків;

– noSQL – бази даних, які підходять, коли потрібно зберігати велику кількість неструктурованих даних. Такі БД використовують інші способи зберігання даних, наприклад в документах (MongoDB), парах «ключ-значення» (DynamoDB), графах (Cassandra) або таблицях з упором на колонки замість рядків (Neo4j);

– спеціалізовані бази даних, які зазвичай оптимізовані для вузьконаправленої задачі. До них входять, наприклад, векторні бази даних, що часто використовуються для роботи зі штучним інтелектом, бо дозволяють «пошук за схожістю».

До компоненту DB входять такі складові, як система керування базами даних (СКБД), конфігурації для роботи з базою даних та механізми її міграції. Цей компонент охоплює всю технічну частину, пов'язану із зберіганням, організацією та підтримкою даних у системі. Він відповідає не лише за фізичне розміщення інформації, а й за забезпечення її цілісності, доступності та коректної структури відповідно до вимог застосунку.

СКБД є основою цього компоненту, адже саме вона забезпечує зберігання, обробку запитів, транзакційність та контроль доступу до даних. Конфігурації для роботи з базою визначають параметри підключення, налаштування пулів з'єднань, особливості взаємодії з конкретною СКБД та інші технічні аспекти, необхідні для стабільної та ефективної роботи. Вони дозволяють адаптувати систему до різних середовищ, таких як розробка, тестування або продакшен.

Міграції бази даних є важливою частиною цього компоненту, оскільки забезпечують еволюцію структури сховища разом із розвитком системи. Вони дозволяють контрольовано змінювати схему бази, додавати або змінювати таблиці, індекси чи обмеження, зберігаючи узгодженість між версіями програмного забезпечення та структурою даних. Завдяки цьому підтримується передбачуваність змін та можливість відтворення стану бази в різних середовищах.

У результаті компонент DB виступає технічною основою для збереження інформації в системі, забезпечує стабільність, масштабованість та керованість процесів роботи з даними.

ORM є «мостом» між БД та програмним кодом, оскільки забезпечує узгоджену взаємодію між об'єктною моделлю застосунку та реляційною структурою зберігання даних. У програмному коді розробник оперує класами, об'єктами, властивостями та зв'язками між ними, тоді як у базі даних інформація представлена у вигляді таблиць. ORM бере на себе відповідальність за перетворення цих двох різних підходів до представлення даних, дозволяючи розробнику працювати з об'єктами, не занурюючись у деталі SQL-запитів та специфіку конкретної системи керування базами даних. Цей компонент виконує автоматичне відображення класів на таблиці, властивостей на стовпці, а також керує зв'язками між сутностями, такими як один-до-одного або один-до-багатьох. Коли застосунок створює або змінює об'єкт, ORM перетворює ці зміни у відповідні запити до бази даних. Аналогічно, під час отримання даних із бази ORM формує об'єкти доменної моделі на основі отриманих записів. Таким чином забезпечується прозорий механізм синхронізації стану між пам'яттю програми та постійним сховищем.

Окрім простого відображення, ORM часто надає додаткові можливості, такі як керування транзакціями, відстеження змін об'єктів, кешування та оптимізація запитів, що спрощує розробку, підвищує продуктивність команди та зменшує ймовірність помилок, пов'язаних із ручним написанням запитів або обробкою результатів.

В екосистемі.NET прикладом ORM є ADO.NET та Entity Framework Core. ADO.NET надає базові механізми роботи з даними та підключеннями, тоді як Entity Framework Core реалізує більш високорівневий підхід до об'єктно-реляційного відображення, дозволяючи працювати з базою даних через доменні класи та запити мовою LINQ. Завдяки таким інструментам розробник може зосередитися на бізнес-логіці, залишаючи технічні аспекти взаємодії з базою даних на рівні інфраструктурного компонента.

Репозиторії (R) є компонентом, що реалізує контракти, визначені на доменному рівні, і забезпечує фактичну взаємодію з механізмами зберігання даних. У домені зазвичай описується лише абстракція доступу до агрегатів або сутностей, тобто визначається, які операції можливі з точки зору бізнес-логіки. Конкретна ж реалізація цих операцій покладається на репозиторії, які знаходяться на інфраструктурному рівні (IL) та працюють із базами даних, ORM або іншими джерелами даних.

Репозиторій виступає посередником між чистою доменною моделлю та технічною реалізацією зберігання. З одного боку, він приймає та повертає доменні об'єкти, дозволяючи бізнес-логіці працювати з ними так, ніби

вони знаходяться в пам'яті. З іншого боку, він виконує всі необхідні дії для збереження або отримання цих об'єктів із зовнішнього сховища. Таким чином, домен не знає і не повинен знати, яким саме способом відбувається доступ до даних.

Ключовою особливістю є те, що репозиторії не створюють прямої залежності домену від інфраструктури. Це досягається завдяки тому, що домен залежить лише від абстрактних контрактів, а конкретні реалізації підключаються ззовні. Такий підхід відповідає принципу інверсії залежностей та дозволяє змінювати технологію зберігання без впливу на бізнес-логіку. У результаті репозиторії допомагають зберегти чистоту архітектури, підтримати ізольованість доменної моделі та зробити систему більш гнучкою та тестованою.

Зовнішні API (OAPI) поєднують застосунок із функціоналом зовнішніх програм та сервісів, забезпечуючи інтеграцію з іншими інформаційними системами. Вони дозволяють розширити можливості власного застосунку за рахунок використання сторонніх ресурсів, не реалізовуючи всю необхідну функціональність самостійно. Через зовнішні API система може обмінюватися даними, ініціювати дії або отримувати результати обробки, що виконуються поза її межами.

Часто такі інтеграції використовуються для надсилання повідомлень, зокрема через електронну пошту або SMS, коли застосунок делегує доставку повідомлень спеціалізованим сервісам, що дозволяє скористатися їхньою надійністю, масштабованістю та готовою інфраструктурою. Окрім цього, зовнішні API можуть застосовуватися для роботи з платіжними системами, сервісами автентифікації, картографічними платформами чи аналітичними інструментами. Таким чином, застосунок стає частиною ширшої екосистеми цифрових сервісів.

Важливо, що взаємодія із зовнішніми API зазвичай відбувається через чітко визначені контракти, які описують формат запитів та відповідей, правила автентифікації та обмеження використання. Оскільки ці інтеграції залежать від сторонніх систем, їх реалізація зазвичай розміщується на інфраструктурному рівні (IL), щоб ізолювати доменну логіку від технічних деталей мережної взаємодії. Такий підхід дозволяє мінімізувати вплив змін у зовнішніх сервісах на внутрішню структуру застосунку та підтримувати стабільність та передбачуваність його роботи.

Файлова система (FS) використовується для збереження файлів, що застосунок створює, обробляє або використовує у своїй роботі: зображення, документи, звіти, тимчасові файли, імпортовані дані чи будь-який інший вміст, який не зберігається безпосередньо в базі даних. На відміну від структурованих записів у БД, файли зазвичай потребують іншого підходу до організації, зберігання та доступу.

Існує багато способів організувати збереження файлів. У найпростішому випадку це може бути локальна директорія на сервері із продуманою структурою підпапок для впорядкування даних. У більш складних або масштабованих системах використовуються спеціалізовані хмарні сервіси зберігання, такі як Amazon S3, що забезпечують високу доступність, надійність та можливість горизонтального масштабування. Такі рішення дозволяють розподіляти навантаження, зберігати великі обсяги даних та забезпечувати доступ до файлів із різних середовищ.

Оскільки конкретний спосіб збереження файлів може змінюватися залежно від вимог до масштабованості, безпеки чи інфраструктури, доцільно виділяти файлову систему в окремий компонент архітектури, що дозволяє ізолювати доменну та прикладну логіку від деталей фізичного зберігання файлів. Застосунок взаємодіє із абстрактним інтерфейсом збереження, тоді як конкретна реалізація може бути локальною, мережною або хмарною. Такий підхід підвищує модульність системи, спрощує тестування та дає можливість змінювати спосіб збереження без впливу на бізнес-логіку.

Система кешування (CS) використовується для зберігання даних, до яких застосунку потрібен частий доступ на читання, щоб підвищити швидкодію та зменшити навантаження на основне сховище даних. Вона дозволяє зберігати тимчасові копії результатів запитів або обчислень, які часто повторюються, щоб при наступному зверненні до тих самих даних не виконувати дорогі операції із базою або складними обчисленнями. Таким чином система кешування прискорює роботу застосунку та забезпечує більш стабільну продуктивність при високих навантаженнях.

Однією із найпопулярніших систем для кешування є Redis – надшвидке сховище даних, яке одночасно може виконувати функції NoSQL бази даних. Завдяки збереженню всіх даних безпосередньо в пам'яті, Redis забезпечує неймовірно швидкий доступ до інформації, час затримки при читанні даних зазвичай менше однієї мілісекунди. Крім того, Redis підтримує різні структури даних, такі як рядки, списки, множини та хеші, що дозволяє ефективно організовувати кешовану інформацію залежно від потреб застосунку.

Використання Redis як компоненту кешування дозволяє системі залишатися гнучкою та масштабованою. Доменна або прикладна логіка працює із абстракцією кешу, не прив'язуючись до конкретної реалізації, а сам Redis виконує зберігання, оновлення та очищення кешу, що дає змогу швидко реагувати на зміни навантаження, підтримувати високу продуктивність та уникати повторюваних витратних операцій, зберігаючи при цьому стабільність та передбачуваність роботи всього застосунку.

Брокер повідомлень (MR) використовується для організації асинхронного спілкування між модулями або сервісами в розподілених системах, забезпечуючи обмін даними без необхідності прямого підключення одних

компонентів до інших. Він виступає проміжним елементом, який приймає повідомлення від одного сервісу і передає їх відповідним отримувачам, таким чином знижуючи залежність між компонентами та підвищуючи гнучкість та масштабованість системи. Завдяки брокеру повідомлень сервіси можуть працювати незалежно один від одного, не очікуючи миттєвої відповіді, що особливо важливо при обробці великої кількості даних або в умовах нерівномірного навантаження.

До сфери відповідальності брокера повідомлень належить не лише передача повідомлень, а й їх маршрутизація. Він визначає, який користувач повинен отримати конкретне повідомлення, може забезпечувати черги, повторні спроби доставки, пріоритети обробки та гарантовану доставку, що дозволяє ефективно координувати роботу розподілених сервісів, спрощує інтеграцію та зменшує ризик втрати даних під час високих навантажень.

Популярними рішеннями для реалізації брокера повідомлень є RabbitMQ, Kafka, Amazon SQS та інші. Кожне з цих рішень має свої особливості та переваги: RabbitMQ забезпечує надійну чергову передачу повідомлень та підтримку різних протоколів, Kafka орієнтований на обробку великих потоків даних та збереження історії повідомлень, а Amazon SQS дозволяє масштабувати черги у хмарі без необхідності керувати інфраструктурою. Використання таких інструментів дозволяє будувати надійні, масштабовані та гнучкі розподілені системи, де компоненти взаємодіють через стандартизований та керований обмін повідомленнями.

Механізми валідації даних (VAL) забезпечують коректність інформації, що надходить у систему, а також правильну роботу самого застосунку, запобігаючи помилкам та неконсистентності даних. Валідація перевіряє, чи відповідають дані встановленим правилам та вимогам бізнесу, а також технічним обмеженням, перш ніж вони будуть використані для обчислень, збережені в базі або передані іншим компонентам. Це критично важливо для збереження цілісності системи та забезпечення надійності її функціонування.

Конкретні механізми валідації розподілені по різних шарах архітектури. На доменному рівні (DL) перевіряються бізнес-правила та логіка, що стосуються сутностей та агрегатів, щоб гарантувати, що стан об’єктів завжди залишатиметься коректним. На прикладному рівні (AL) можуть перевірятися сценарії використання, формати запитів та правильність параметрів, що надходять від користувачів або зовнішніх систем. На інфраструктурному рівні (IL) здійснюється перевірка технічних обмежень, таких як типи даних у базі, довжина рядків, наявність обов’язкових полів та інші параметри збереження. Незважаючи на розподіл по шарах, всі механізми валідації поділяють спільну мету – забезпечити цілісність, надійність та передбачуваність роботи застосунку, гарантувати, що дані залишаються коректними, а логіка системи виконується відповідно до очікувань.

Компонент VAL включає наступні частини:

$$VAL = \{ DV, AV, IV, PV \}, \tag{7}$$

де DV (Domain Validation) – доменна валідація; AV (Application Validation) – валідація прикладного рівня; IV (Infrastructure Validation) – валідація інфраструктури; PV (Presentation Validation) – презентаційна валідація.

Доменна валідація (DV) дозволяє забезпечити дотримання бізнес-правил та уникнути ситуацій, коли стан системи стає некоректним або суперечить вимогам предметної області. Вона гарантує, що всі дані та дії в межах домену відповідають встановленим правилам та обмеженням. Більша частина такої валідації відбувається безпосередньо всередині сутностей та об’єктів-значень, де перевіряються значення властивостей та забезпечується цілісність агрегатів. Сутності самостійно контролюють свої стани та не дозволяють створювати або змінювати об’єкти таким чином, щоб порушувати бізнес-логіку. Окрім того, доменні сервіси теж можуть містити валідацію, яка перевіряє коректність операцій, що охоплюють кілька сутностей одночасно або вимагають виконання складних правил. Такі сервіси забезпечують, щоб взаємодія між об’єктами відбувалася відповідно до бізнес-логіки і не створювала суперечностей у системі.

Прикладом доменної валідації може бути тип E-mail, де перевірка того, чи є рядок правильною електронною адресою, виконується безпосередньо в конструкторі цього типу. Завдяки цьому неможливо створити об’єкт E-mail із некоректним значенням, і таким чином система гарантує, що всі електронні адреси, що використовуються в домені, завжди відповідають очікуваному формату. Такий підхід робить доменну модель надійною, самодостатньою та стійкою до помилок, адже правила валідації вбудовані безпосередньо в логіку об’єктів та операцій.

Валідація прикладного рівня (AV) перевіряє правильність виконання конкретних бізнес-функцій, забезпечуючи, щоб операції, ініційовані користувачем або зовнішніми системами, могли відбутися коректно. Вона відповідає за перевірку даних на рівні сценаріїв використання і контролює, чи виконуються всі необхідні умови для виконання конкретної команди. Основна увага приділяється вхідним даним, що надходять у Command, адже саме через них користувач або інший сервіс передає параметри операції. Валідація на цьому рівні гарантує, що ці параметри відповідають очікуваному формату, обмеженням і правилам бізнес-логіки, перш ніж вони будуть передані в домен.

Крім перевірки формату даних, прикладна валідація контролює наявність необхідних сутностей у системі, без яких виконання дії неможливе або неприпустиме. Наприклад, перед створенням нового замовлення перевіряється, чи існує користувач, до якого прив’язане замовлення, або чи доступні об’єкти, які беруть участь у транзакції. Завдяки цьому забезпечується цілісність процесу та виключається ситуація, коли бізнес-функція запускається

з неповними або некоректними даними. Таким чином, валідація прикладного рівня (AV) служить проміжним контролем між зовнішнім світом та доменною логікою, захищаючи систему від помилок на етапі підготовки до виконання операцій і забезпечуючи передбачувану та надійну роботу застосунку.

Валідація інфраструктури (IV) відповідає за перевірку зовнішніх обмежень системи та технічних аспектів роботи застосунку, забезпечуючи, щоб усі операції виконувалися відповідно до правил, встановлених інфраструктурою. Вона контролює, чи дані та запити відповідають обмеженням бази даних, таким як типи полів, довжина рядків, унікальність значень або наявність обов'язкових стовпців, а також перевіряє дотримання технічних лімітів зовнішніх сервісів та API. Така перевірка дозволяє запобігти помилкам, що можуть виникнути через невідповідність даних або перевищення ресурсних обмежень зовнішніх систем, і забезпечує стабільну роботу застосунку під час взаємодії з базою даних та сторонніми сервісами.

Валідація інфраструктури також важлива для підтримки цілісності та надійності системи, адже навіть коректні з точки зору домену або прикладної логіки дані можуть викликати помилки, якщо порушують технічні обмеження. Вона ізолює домен та бізнес-логіку від технічних деталей реалізації, дозволяючи системі працювати передбачувано та безпечно, незалежно від конкретних характеристик бази даних, мережних сервісів або сторонніх API. У такий спосіб складова (IV) служить останнім рівнем контролю перед фактичним виконанням операцій, гарантує сумісність та правильність взаємодії компонентів застосунку із інфраструктурою.

Валідація презентації (PV) відповідає за перевірку API-запитів на коректність та виконує початкову обробку вхідних даних ще до того, як вони потрапляють на прикладний (Al) або доменний рівень (DL). Вона служить першим бар'єром, який забезпечує, що дані, які надходять від користувача або зовнішньої системи, відповідають базовим вимогам формату та структури. Основна увага приділяється перевіркам на null або присутності порожніх рядків, а також іншим елементарним правилам коректності, які дозволяють запобігти помилкам ще на рівні взаємодії з інтерфейсом або API.

Такий підхід дозволяє зменшити навантаження на внутрішні шари системи та захистити її від некоректних або неповних даних. Складова (PV) не оцінює бізнес-логіку або технічні обмеження, її мета полягає у формальній перевірці даних, забезпеченні їхньої базової цілісності та готовності до подальшої обробки. Завдяки цьому користувач або клієнт API отримує швидкий і передбачуваний зворотний зв'язок у разі очевидних помилок у запиті, що підвищує стабільність та надійність роботи застосунку, а також робить систему більш стійкою до некоректного використання або випадкових помилок на стороні клієнта.

Поперечні компоненти (CC) – це технічні аспекти системи, які застосовуються одночасно до всіх рівнів архітектури та винесені окремо від бізнес-логіки, оскільки їхня функція не пов'язана безпосередньо з предметною областю, а підтримує загальні механізми роботи системи. Вони забезпечують повторювані або загальні задачі, які потрібні у багатьох частинах застосунку, і дозволяють централізовано керувати цими функціями, не розпорошуючи їх по різних шарах або компонентах.

Поперечні компоненти можуть включати такі механізми, як логування, моніторинг, автентифікація та авторизація, обробка винятків, маршрутизація запитів, кешування, аудит подій або безпека даних. Завдяки виділенню цих функцій у окремі компоненти, бізнес-логіка залишається чистою і зрозумілою, а технічні аспекти можна змінювати, масштабувати або оптимізувати незалежно від доменних правил. Вони забезпечують узгодженість і повторне використання технічних рішень у різних частинах системи, підвищують її надійність, полегшують тестування та підтримку, а також дозволяють швидко впроваджувати нові крос-рівневі функції без втручання у бізнес-логіку.

Структуру складової (CC) можна представити наступним чином:

$$CC = \{ LOG, EH, AUTH, Cache, MON, Middleware \}, \quad (8)$$

де LOG – логування; EH (Error Handling) – механізм обробки помилок; AUTH – аутентифікація; Cache – кешування; MON – моніторинг; Middleware – проміжні функції.

Модуль логування (LOG) відповідає за документацію усіх дій, що відбуваються в застосунку. Структуроване логування спрощує відладку роботи застосунку та пошук багів як в середовищі розробки, так і в робочому середовищі (production).

Механізм обробки помилок (EH) відповідає за забезпечення стабільності роботи системи, гарантуючи, що жодна помилка не призведе до збоїв або «падіння» застосунку. Помилки повинні фіксуватися в логах та коректно відображатися кінцевому користувачу, водночас дотримуючись вимог безпеки та не розкриваючи внутрішніх деталей системи. У запропонованій моделі реалізації механізму обробки помилок передбачено два ключові підходи: використання патерну Result, який дозволяє безпечну передачу результатів операцій з інформацією про помилки та фільтр виключень (Exception Filter), що централізовано перехоплює та обробляє винятки, забезпечуючи контрольовану реакцію системи на непередбачені ситуації.

Патерн Result працює таким чином, що будь-яка функція, в якій може виникнути помилка, повертає не просто значення, а спеціальний об'єкт-результат. Цей об'єкт містить або коректні дані функції у разі успішного виконання, або інформацію про помилку, якщо сталася проблема. Такий підхід дозволяє обробляти помилки

безпосередньо під час виконання програми і дає змогу створювати власні типи для представлення специфічних помилок, адаптованих під конкретні сценарії.

Фільтр виключень забезпечує централізовану обробку непередбачених ситуацій у Web API. Він виступає як обгортка над хендлерами запитів та відстежує виникнення критичних помилок або виключень (Exceptions). Використання фільтра дозволяє сконцентрувати обробку таких помилок в одному місці, документувати їх за допомогою модуля логування (LOG) та повертати користувачу коректну відповідь (наприклад, HTTP-код 500 Internal Server Error). Завдяки цьому обробники запитів та сервіси залишаються чистими та не перевантажуються логікою обробки винятків, що підвищує структурування та передбачуваність поведінки системи при критичних ситуаціях.

Патерн Result та фільтр виключень можна комбінувати наступним чином: Result використовується для обробки очікуваних помилок (помилки валідації вхідних даних, не знайдено жодного результату для Query), в той час як фільтр виключень відповідає за критичні помилки (наприклад, помилки в конфігурації інфраструктурі). Разом ці підходи перекривають усі можливі помилкові ситуації, при цьому майже не шкодячи чистоті коду.

Модуль автентифікації та авторизації (AUTH) відповідає за контроль доступу користувачів до системи та її ресурсів. Він виконує дві взаємопов'язані функції: автентифікацію, яка підтверджує особу користувача, та авторизацію, що визначає, які операції дозволені для конкретного користувача в рамках системи. Завдяки цьому модуль (AUTH) підвищує безпеку застосунку та гарантує, що конфіденційні дані або функції не стануть доступними для неповноважених осіб.

Популярним підходом для реалізації автентифікації є використання JSON Web Token (JWT). Цей метод дозволяє створювати цифровий токен, який містить закодовану інформацію про користувача, наприклад, його ідентифікатор, роль або права доступу. Токен додається до заголовків HTTP-запиту та передається разом із запитом до сервера. Сервер, отримавши запит, розшифровує токен та перевіряє його дійсність, що дозволяє підтвердити особу користувача та визначити його права. Такий підхід робить систему більш масштабованою та незалежною від стану сесій, оскільки токен містить усю необхідну інформацію для перевірки доступу без потреби звертатися до центрального сховища стану сесій. Крім того, JWT дозволяє легко інтегруватися із різними сервісами та клієнтськими застосунками, забезпечуючи безпечну та ефективну аутентифікацію та авторизацію у сучасних веб- та мобільних системах.

Механізм кешування (Caching) зберігає результати частих або повторюваних запитів, щоб наступні звернення до тих самих даних відбувалися значно швидше, що дозволяє уникнути повторного виконання дорогих операцій, таких як обчислення, обробка великих обсягів даних або звернення до бази даних. Ці дії підвищують продуктивність та скорочують час відповіді системи. Завдяки кешуванню підвищується пропускна здатність серверного або вебзастосунку, оскільки зменшується навантаження на основні компоненти та зберігаються ресурси для обробки нових запитів.

Компонент кешування (Caching) використовує сервер кешування (CS), який входить до складу інфраструктурного рівня (IL) для фактичного зберігання тимчасових даних. Цей сервер працює як високошвидкісне сховище, здатне обробляти запити за мілісекунди, і забезпечує надійне зберігання кешованих результатів. Завдяки такій організації кешування стає прозорим для доменної та прикладної логіки: компоненти системи взаємодіють із кешем через абстракції механізму, не турбуючись про технічні деталі збереження або оновлення даних, що дозволяє системі залишатися гнучкою, масштабованою та здатною підтримувати високу продуктивність навіть при збільшенні навантаження або обсягу інформації, що оброблюється.

Впровадження складової моніторингу та аналітики (MON) полягає у безперервному спостереженні за роботою застосунку та зборі даних, які допомагають оцінювати його ефективність та приймати обґрунтовані рішення щодо подальшого розвитку системи. Використання таких методів, як логування, замір метрик та трасування шляху запитів, дозволяє виявляти вузькі місця, затримки або аномалії в роботі компонентів, аналізувати причини збоїв та оптимізувати продуктивність. Завдяки цьому можна точно розуміти, як поведінка системи відповідає очікуванням, і визначати пріоритети для покращень або масштабування.

До функцій цього компонента також входять перевірки працездатності, які постійно контролюються стан ключових сервісів та ресурсів системи. У разі виявлення проблем модуль моніторингу здатний оперативно сповіщати розробників або адміністраторів, забезпечуючи швидку реакцію на неполадки та зменшуючи ризик тривалих збоїв. Крім того, зібрані дані використовуються для аналітики продуктивності, планування ресурсів та прогнозування навантажень, що дозволяє системі розвиватися більш передбачувано та ефективно. Таким чином, компонент моніторингу та аналітики є важливим інструментом для підтримки стабільності, надійності та масштабованості застосунку.

Middleware – це функція або набір функцій, які виконуються на проміжному етапі між отриманням HTTP-запиту та відправкою відповіді клієнту. Вони виступають як «посередник», який перехоплює запит і може обробляти його, змінювати або додавати необхідну інформацію ще до того, як управління передається основному обробнику запиту, тобто Endpoint. Це дозволяє централізовано впроваджувати різні крос-рівневі функції системи, не засмічуючи логіку окремих сервісів або контролерів.

Механізм Middleware можна застосовувати для реалізації багатьох поперечних компонентів, таких як автентифікація, авторизація, логування, моніторинг чи обробка помилок. Наприклад, в ASP.NET Core вбудовані функції

автентифікації та авторизації реалізовані саме як Middleware, які перехоплюють запит та перевіряють права доступу користувача перед тим, як запит потрапляє до відповідного Endpoint. Аналогічно, фільтри виключень у Web API фактично є обгортками над обробником Endpoint, що теж по суті є Middleware. Завдяки такому підходу Middleware дозволяє централізовано впроваджувати повторювану логіку для всіх запитів, роблячи систему більш структурованою, гнучкою та передбачуваною.

Складова, що відповідає за гібридну архітектуру (CSA) включає в себе декілька архітектурних патернів, що забезпечують чітку, але при цьому гнучку та динамічну структуру застосунку:

$$CSA = \{ CA, VSA, CQRS \}, \quad (9)$$

де CA (Clean Architecture) – «чиста» архітектура; VSA (Vertical Slice Architecture) – архітектура вертикальних зрізів; CQRS (Command and Query Respondibility Segregation) – принцип розбиття операцій застосунку на команди та запити.

VSA розбиває програмну систему не на горизонтальні шари, а на вертикальні зрізи, кожен з яких повністю охоплює один Use Case. Кожен зріз є окремим модулем, що має бути максимально незалежним від інших зрізів. Ключові принципи архітектури включають в себе:

– високу зв’язність (High Cohesion): код, що змінюється разом, знаходиться поруч;

– низьке зчеплення: зрізи максимально незалежні один від одного.

Ще одна перевага такої архітектури полягає в тому, що вона відповідає підходу «Модульного моноліту», який допомагає розробляти програму як систему із чітко розмежованими модулями, що за потреби можуть бути розгорнуті як окремі сервіси, що дозволяє почати розробку програмного продукту як моноліту на етапі, коли використання мікросервісів ще не має сенсу, та зробити безболісний перехід на мікросервісну архітектуру, коли вона стане доцільною.

В класичній VSA структура зрізів взагалі ніяк не узгоджена. Такий підхід робить її максимально гнучкою, але відсутність будь-яких загальних правил може привести до того, що код у зрізі взагалі не буде структурованим, що призведе до проблем з тестованістю коду та майбутньою його підтримкою. У зв'язку з цим пропонується поєднати VSA із однією з багатошарових архітектур, а саме із чистою архітектурою (CA).

Основні принципи CA передбачають поділ коду застосунку на 4 шари (презентаційний, інфраструктурний, прикладний та доменний) з однонаправленими залежностями між ними, де напрямок руху залежностей йде від Presentation до Infrastructure, далі до Application і нарешті до Domain. Така організація дозволяє бізнес-функціям і правилам залишатися повністю ізольованими від зовнішніх систем та технологій, зокрема від бази даних, забезпечуючи чистоту доменної логіки та незалежність бізнес-правил від технічних деталей реалізації.

Незалежність вертикальних зрізів не тільки підвищує гнучкість архітектури, але й добре поєднується з принципом Command Query Responsibility Segregation (CQRS). CQRS – це архітектурний шаблон, що полягає в розділенні операцій читання від операцій запису.

Основні компоненти CQRS включають в себе:

– запити: використовуються для отримання даних з метою їх відображення користувачу; мають максимально просту логіку (оптимізована вибірка та приведення результату до вигляду, якого потребує інтерфейс); ніколи не змінюють стан системи;

– команди: змінюють стан системи; містять бізнес-правила та валідацію, не повертають дані (виключенням може бути ідентифікатор новоствореного об’єкту або статус виконання операції).

CQRS допомагає у вирішенні наступних проблем:

– перевантажені моделі: в традиційних архітектурах одна і та сама модель часто є перевантаженою через те, що вона розрахована і на читання, і на запис;

– різниця в потребах: для запитів головну роль грає продуктивність отримання даних, в той час як команди можуть потребувати складної логіки та валідації, що призводить до різних потреб у структурі реалізації цих двох видів операцій;

– масштабованість: читання зазвичай відбувається значно частіше за запис. Використання підходу CQRS дозволяє масштабувати їх окремо;

– продуктивність: буває складно оптимізувати одну базу даних і для швидкого читання, і для складних транзакцій.

Незалежність вертикальних зрізів VSA добре поєднується з принципом Command Query Responsibility Segregation (CQRS). Їхня сумісність полягає в тому, що обидва підходи мають в своїй основі вертикальний поділ, що дозволяє дуже органічно їх поєднувати.

Результати та їх обговорення

Розглянемо декілька прикладів, які демонструють переваги використання гібридної серверної архітектури.

Приклад 1: розробка нового вебзастосунку з нуля. Створення застосунку з використанням Clean Architecture потребує приблизно 3 тижні на підготовку базової інфраструктури, включно з налаштуванням абстракцій,

організацією проєктів для шарів та іншими технічними аспектами, перш ніж можна буде реалізовувати бізнес-функції, що приносять реальну користь. Такі витрати часу на підготовчі етапи, які безпосередньо не додають цінності користувачу, збільшують Time to Market проєкту та уповільнюють отримання перших результатів. Використання модифікованої гібридної моделі архітектури дозволяє зменшити час, потрібний на створення інфраструктури, до одного тижня. При цьому створена інфраструктура відзначатиметься гнучкістю та мінімальною кількістю необхідних абстракцій.

Приклад 2: додавання нових бізнес-функцій. Розробка нового функціоналу є типовою задачею при створенні та підтримці серверних та вебзастосунків. При використанні класичної багатошарової архітектури реалізація такого функціоналу може займати близько 20 годин, залежно від складності конкретної задачі. Використання гібридної архітектури під час розробки застосунку дозволяє зекономити до 60% затраченого часу, скорочуючи його до 8 годин. В залежності від масштабу проєкту такий виграш у часі може означати меншу кількість розробників, що потребуються для розробки застосунку.

Однією з причин швидкої розробки нового функціоналу є мінімізація необхідної кількості абстракцій, що була згадана у попередньому прикладі. Вона досягається за допомогою використання узагальнених абстракцій CQRSI, які дозволяють не створювати нові інтерфейси для шарів бізнес-функції у переважній кількості випадків. Також, завдяки використанню DDD та Rich Domain в домені бізнес-функцій зі складною бізнес-логікою облегшує написання тестів для коду, що теж робить свій внесок у скорочення часу розробки.

Приклад 3: модифікація існуючої бізнес-функції. В класичних багатошарових архітектурах внесення змін у бізнес-функцію може займати до 10 годин, навіть якщо потрібна лише незначна корекція в її структурі. Крім того, подібні зміни зазвичай призводять до погіршення якості коду та ускладнення шару, де здійснюються правки, що спричиняє поступове розростання та ускладнення підтримки бізнес-логіки. Внесення змін у бізнес-функції, створені за принципами гібридної архітектури, проходить до 70% швидше, займаючи приблизно 3 години. Також зміни, внесені в один вертикальний зріз, ніколи не шкодять структурі інших бізнес-функцій завдяки тому, що кожна бізнес-функція є окремим модулем. Таким чином гібридна архітектура дозволяє зберігати високу чистоту коду навіть в динамічному середовищі, де потреби бізнесу регулярно змінюються.

Приклад 4: масштабування від моноліту до мікросервісів. Уявімо, що розробка e-commerce вебзастосунку почалася як монолітний проєкт, оскільки на початкових етапах навантаження було невелике, і використання мікросервісів не виправдовувало додаткові складнощі. Через 3 місяці зростання кількості бізнес-функцій та користувачів призвело до підвищеного навантаження на модуль обробки платежів, що поставило питання про перехід до мікросервісної архітектури.

Класичні багатошарові архітектури розраховані на великі монолітні застосунки, тому перехід такої системи на мікросервіси часто вимагає суттєвих змін у коді. Наприклад, необхідно «розплутати» залежності модуля, який виділяється в окремий сервіс, від інших частин моноліту, що залишаються в основному застосунку. Ці складнощі уповільнюють процес створення нового мікросервісу, який може займати півтора-два тижні, а зміни в структурі коду часто супроводжуються появою багів та проблем.

Якщо ж застосунок спочатку будувався за принципами гібридної архітектури, перехід від моноліту до мікросервісів проходить набагато швидше та простіше. Завдяки розбиттю доменної кодової бази на незалежні вертикальні зрізи (vertical slices) бізнес-функції можна виділяти в окремі сервіси без суттєвих змін у структурі їхнього коду. У більшості випадків це дозволяє реалізувати перехід всього за 2-3 дні, забезпечуючи швидку масштабованість та мінімізуючи ризик виникнення багів під час трансформації архітектури.

Приклад 5: продуктивність високонавантажених операціях. Розглянемо соціальні мережі, де кількість операцій читання значно перевищує кількість операцій запису. Наприклад, перегляди стрічки новин можуть відбуватися у сотні разів частіше, ніж створення нових постів. У класичній багатошаровій архітектурі для обох типів операцій використовується одна й та сама модель, що ускладнює структуру коду та може створювати «пляшкове горло» (bottleneck) при високому навантаженні на читання.

Використання гібридної архітектури дозволяє оптимізувати операції читання окремо від операцій запису, що значно зменшує затримки: час відповіді для запитів читання можна скоротити з 400 мс до 150-200 мс, забезпечуючи приріст продуктивності до 50%. Такий результат досягається завдяки принципу CQRS, який розділяє операції на команди для змін стану та запити для отримання даних, що дозволяє оптимізувати запити незалежно від команд, прибираючи непотрібні абстракції та перевірки, які залишаються лише у командах з більш складною бізнес-логікою, і тим самим підвищує ефективність системи під високим навантаженням.

Висновки

В ході проведення дослідження на основі попереднього аналізу було модифіковано модель архітектури вебзастосунку за рахунок додавання складової (CSA), яка включає в себе декілька архітектурних патернів (CA, VSA, CQRS), що забезпечують чітку, але при цьому гнучку та динамічну структуру застосунку.

Використання Clean Architecture забезпечує чіткий поділ на шари, де бізнес-логіка і правила залишаються ізольованими від зовнішніх залежностей, таких як база даних або UI, що робить систему більш стабільною та

передбачуваною. Вертикальні зрізи (VSA) дозволяють організовувати код у повністю незалежні функціональні блоки, де всі шари від UI до домену реалізовані для конкретної бізнес-операції, що значно полегшує розробку нових функцій та їх тестування без ризику впливу на інші частини застосунку.

Додавання архітектурного шаблону CQRS забезпечує чітке розділення операцій на команди, які змінюють стан системи, і запити, які лише читають дані, що дозволяє оптимізувати роботу з даними, підвищити продуктивність, зменшити ймовірність помилок та спрощує масштабування окремих частин системи незалежно одна від одної.

Результатом такого підходу стає архітектура, яка поєднує структурованість та передбачуваність класичних багатошарових моделей із високою адаптивністю та динамічністю сучасних архітектурних патернів, що дозволяє будувати масштабовані, стійкі до змін та зручні для підтримки застосунки, які легко адаптуються до зростання навантаження, розвитку бізнес-функцій і переходу до мікросервісного підходу при потребі.

Список використаної літератури

1. Sereda, D. Creating a Multi-tier Architecture for Web Applications: Design and Implementation. *American Scientific Research Journal for Engineering, Technology, and Sciences*. Vol. 102 № 1. 2025. P. 134-139. https://asrjetsjournal.org/American_Scientific_Journal/article/view/11686

2. Khan, S. M. (2023) Onion Architecture Used in Software Development. https://www.researchgate.net/publication/371006360_Onion_Architecture_Used_in_Software_Development

3. Tu, Z. Research on the Application of Layered Architecture in Computer Software Development. *Journal of Computing and Electronic Information Management*. № 11(3). 2023. С. 34-38. https://doi.org/10.54097/jceim.v11i3.08

4. Dulko, D., Kolianova, T. Using Clean Architecture to Build Scalable Solutions in.NET. Матеріали конференцій МЦНД. 2025. С. 174-176. https://doi.org/10.62731/mcnd-07.03.2025.002

5. Varga, M. Web Programming and Multi-Tier Architecture of Web Applications. *TEM Journal*. Volume 13, Issue 4, Pages 3286-3294. https://doi.org/10.18421/TEM134-63

6. Su, R., Li, X. Modular Monolith: Is This the Trend in Software Architecture? 2024. *arXiv preprint arXiv:2401.11867v1*. https://doi.org/10.48550/ arXiv.2401.11867

7. Jayaraman, K. D., Sharma, P. Exploring CQRS Patterns for Improved Data Handling in Web Applications. *International Journal of Research in all Subjects in Multi* Languages. Vol. 13, Issue: 01. 2025. PP. 92-109. https://doi.org/10.5281/zenodo.15069282

8. Yakhin, S. (2025) Comparative Review of Clean Architecture and Vertical Slice Architecture Approaches for Enterprise.NET Applications. *International Journal of Advanced Artificial Intelligence Research*, Volume 2, Issue 12, PP. 1-12. https://doi.org/10.55640/ijaair-v02i12-01

References

1. Sereda, D. (2025). Creating a Multi-tier Architecture for Web Applications: Design and Implementation. *American Scientific Research Journal for Engineering, Technology, and Sciences*. Vol. 102 № 1. P. 134-139. https://asrjetsjournal.org/American_Scientific_Journal/article/view/11686

2. Khan, S. M. (2023) Onion Architecture Used in Software Development. https://www.researchgate.net/publication/371006360_Onion_Architecture_Used_in_Software_Development

3. Tu, Z. (2023). Research on the Application of Layered Architecture in Computer Software Development. *Journal of Computing and Electronic Information Management*. № 11(3). pp. 34-38. https://doi.org/10.54097/jceim.v11i3.08

4. Dulko, D., Kolianova, T. (2025). Using Clean Architecture to Build Scalable Solutions in.NET. Materialy konferentsii MTsND. pp. 174-176. https://doi.org/10.62731/mcnd-07.03.2025.002

5. Varga, M. Web Programming and Multi-Tier Architecture of Web Applications. *TEM Journal*. Volume 13, Issue 4, Pages 3286-3294. https://doi.org/10.18421/TEM134-63

6. Su, R., Li, X. (2024). Modular Monolith: Is This the Trend in Software Architecture? *arXiv preprint arXiv:2401.11867v1*. https://doi.org/10.48550/ arXiv.2401.11867

7. Jayaraman, K. D., Sharma, P. (2025). Exploring CQRS Patterns for Improved Data Handling in Web Applications. *International Journal of Research in all Subjects in Multi* Languages. Vol. 13, Issue: 01. pp. 92-109. https://doi.org/10.5281/zenodo.15069282

8. Yakhin, S. (2025). Comparative Review of Clean Architecture and Vertical Slice Architecture Approaches for Enterprise.NET Applications. *International Journal of Advanced Artificial Intelligence Research*, Volume 2, Issue 12, pp. 1-12. https://doi.org/10.55640/ijaair-v02i12-01

Дата першого надходження статті до видання: 19.02.2026
Дата прийняття статті до друку після рецензування: 26.03.2026
Дата публікації (оприлюднення) статті: 07.05.2026