**Architecture Decision Records**

# **Заголовок**

Перехід на модульной моноліт

## **Контекст**

Поточна архітектура проєкту базується на трирівневій (three-layered) структурі, яка з часом ускладнює масштабування, підтримку та внесення змін. Зі зростанням функціональності проєкту виникла потреба у більш гнучкому, масштабованому та підтримуваному рішенні. При цьому бажано зберегти простоту розгортання та уникнути повноцінного переходу на мікросервіси, які потребують додаткових інфраструктурних витрат.

## **Рішення**

Було прийнято рішення перейти на **модульний моноліт** — архітектурний підхід, що поєднує переваги моноліту (простота розгортання, цілісність коду) та мікросервісів (ізоляція функціональності, розділення відповідальності).

Модульний моноліт передбачає розділення системи на окремі модулі, кожен з яких:

* має чітко визначені межі відповідальності;
* інкапсулює бізнес-логіку;
* взаємодіє з іншими модулями через **публічні інтерфейси** або **асинхронні повідомлення (event-driven design)**.

Таке розділення дозволяє командам працювати над модулями незалежно один від одного, зменшуючи ризик неузгоджених змін та полегшуючи тестування і розширення системи.

## **Статус**

Прийнято

## **Наслідки**

Моноліт було розбито на дві частини: **Students** та **Works**, кожна з яких ізольовано одна від одної і обмінюється даними за допомогою брокера повідомлень RabbitMq. Стало простіше працювати над частинами, внутрішня реалізація модуля може змінюватися без впливу на інші модулі, за умови збереження контракту, систему стало простіше розширювати а процес розгортання майже не змінився. Також, як невеличкий бонус, модульна архітектура полегшує потенційний поділ на мікросервіси в майбутньому.

# **Заголовок**

Використання чистої архітектури в модулях

## **Контекст**

Із переходом до модульного моноліту кожен функціональний блок (модуль) став самостійною одиницею, яка відповідає за окрему частину бізнес-логіки. Для підвищення якості коду, покращення тестованості, спрощення підтримки та підготовки до потенційної незалежної міграції модулів (наприклад, до мікросервісів) виникла потреба у впровадженні архітектурного шаблону, який забезпечить чітке розділення відповідальностей.

Було розглянуто кілька підходів, і найбільш відповідним виявилася **чиста архітектура (Clean Architecture)**.

## **Рішення**

Було прийнято впровадити **чисту архітектуру** всередині кожного модуля. Цей підхід забезпечує:

* **Інверсію залежностей** — бізнес-логіка не залежить від інфраструктури.
* **Чітке розділення шарів**:
  + **Domain** — сутності, value-об'єкти, сервіси домену.
  + **Application** — use cases, інтерфейси для зовнішньої взаємодії.
  + **Infrastructure** — реалізації зовнішніх сервісів (БД, брокери повідомлень, API).
  + **Presentation** — контролери, обробники повідомлень тощо.
* **Тестованість** — доменна логіка та use cases легко покриваються unit-тестами без залежності від БД чи інших зовнішніх систем.
* **Гнучкість** — заміна реалізацій (наприклад, іншої бази даних або способу доставки повідомлень) не впливає на core-логіку.

## **Статус**

Прийнято

## **Наслідки**

Організація модулів з дотриманням чистої архітектури внесла ряд покращень у бік гнучкості та розширюваності. Стало простіше інтегрувати шаблони CQRS, Mediator, Event-Driven Design у відповідних шарах. Зросла кількість файлів/структурних елементів у модулях, однак це компенсується кращою підтримуваністю та тестованістю. Кожен модуль тепер має чітку ієрархію шарів, залежності між якими спрямовані лише “всередину”.