

ỦY BAN NHÂN DÂN TP HỒ CHÍ MINH

**TRƯỜNG ĐẠI HỌC SÀI GÒN**

**KHOA CÔNG NGHỆ THÔNG TIN**

**~~~~~~\*~~~~~~**

**BÁO CÁO CÁ NHÂN**

**KIẾN TRÚC CLEAN CỦA DJANGO**

**Sinh viên thực hiện:**

Trần Đồng Gia Hân 3121411066

**Lớp: DCT121C3**

Giảng viên hướng dẫn: TS. Đỗ Như Tài

*TP. Hồ Chí Minh, tháng 03 năm 2025*

# **MỤC LỤC**

[**MỤC LỤC** 1](#_Toc194889412)

[**1.** **Giới thiệu về kiến trúc Clean Architecture** 2](#_Toc194889413)

[**1.1.** **Khái niệm** 2](#_Toc194889414)

[**1.4.** **Sơ đồ minh họa** 2](#_Toc194889415)

[**2.** **Ưu điểm và nhược điểm của kiến trúc phân lớp nguyên khối** 3](#_Toc194889416)

[**2.1. Ưu điểm** 3](#_Toc194889417)

[**2.2. Nhược điểm** 4](#_Toc194889418)

[**3.** **So sánh với kiến trúc Microservices** 4](#_Toc194889419)

[**4.** **Ứng dụng thực tế** 5](#_Toc194889420)

[**5. Kết luận** 5](#_Toc194889421)

## **Giới thiệu về kiến trúc Clean Architecture**

* 1. **Khái niệm**

Clean Architecture là một mẫu kiến trúc phần mềm được đề xuất bởi Robert C. Martin (Uncle Bob), nhằm tổ chức mã nguồn một cách rõ ràng, linh hoạt và dễ bảo trì. Kiến trúc này giúp phân tách hệ thống thành các tầng riêng biệt, mỗi tầng có vai trò rõ ràng và giảm sự phụ thuộc lẫn nhau giữa các phần trong hệ thống. View (V): Xử lý yêu cầu từ người dùng và kết nối Model và Template.

* 1. **Nguyên lý cốt lõi**

Tầng trong không phụ thuộc tầng ngoài: logic nghiệp vụ (domain) không bị ảnh hưởng nếu thay đổi UI, database, hay framework.

Mọi phụ thuộc đều hướng vào trong (từ ngoài → trong).

Business logic (logic nghiệp vụ) là trung tâm của hệ thống và phải độc lập với công nghệ.

* 1. **Mục tiêu**

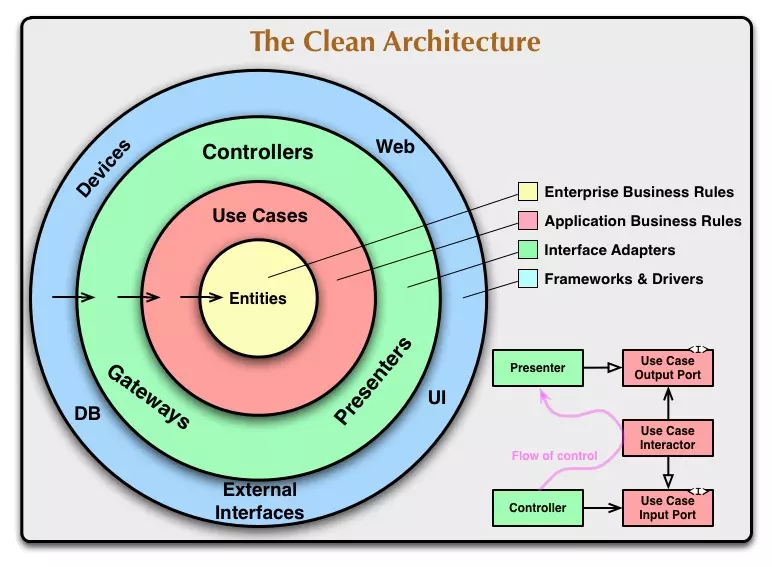
Dễ mở rộng, bảo trì và kiểm thử.

Tái sử dụng logic ứng dụng cho nhiều nền tảng (Web, Mobile, API...).

Giảm rủi ro khi thay đổi framework, công nghệ hoặc thư viện.

* 1. **Sơ đồ minh họa**

Dưới đây là sơ đồ minh họa các thành phần cơ bản của một ứng dụng Django:



1. **Ưu điểm và nhược điểm của kiến trúc phân lớp nguyên khối**

**2.1. Ưu điểm**

* Tách biệt rõ ràng các thành phần: Kiến trúc này giúp tách biệt rõ ràng giữa các thành phần như Domain, Use Cases, Infrastructure, và UI. Điều này giúp mã nguồn trở nên dễ dàng bảo trì và mở rộng.
* Dễ dàng thay đổi và mở rộng: Do tách biệt các lớp, bạn có thể thay đổi hoặc thay thế một phần của hệ thống mà không làm ảnh hưởng đến các phần khác. Ví dụ, bạn có thể thay đổi cách lưu trữ dữ liệu (chuyển từ SQL Server sang MongoDB) mà không cần thay đổi các lớp liên quan đến logic nghiệp vụ (Use Cases, Domain).
* Dễ dàng kiểm thử (Testable): Các thành phần như Use Cases và Domain được tách biệt hoàn toàn khỏi các phần phụ thuộc bên ngoài (như cơ sở dữ liệu, API). Điều này giúp việc kiểm thử dễ dàng hơn vì bạn có thể mock các phụ thuộc này mà không cần phải chạy toàn bộ hệ thống.
* Khả năng mở rộng tốt: Clean Architecture cho phép bạn mở rộng hệ thống một cách dễ dàng. Nếu bạn muốn thêm một tính năng mới (chẳng hạn như thêm một loại cơ sở dữ liệu mới hoặc giao diện người dùng mới), bạn chỉ cần thêm vào các thành phần tương ứng mà không ảnh hưởng đến các phần còn lại của hệ thống.
* Chạy độc lập với công nghệ: Kiến trúc này giúp hệ thống không bị phụ thuộc vào công nghệ cụ thể. Ví dụ, phần Infrastructure có thể thay đổi từ ORM này sang ORM khác mà không làm ảnh hưởng đến logic nghiệp vụ.

**2.2. Nhược điểm**

* Phức tạp và tốn thời gian khi mới bắt đầu: Clean Architecture yêu cầu một sự tổ chức cẩn thận và chi tiết ngay từ đầu. Điều này có thể tạo cảm giác phức tạp khi mới bắt đầu, đặc biệt là trong các dự án nhỏ hoặc với nhóm ít người.
* Yêu cầu về thiết kế sơ bộ: Vì kiến trúc này đòi hỏi tách biệt rõ ràng các phần của ứng dụng, nên việc thiết kế và xác định các lớp từ đầu là rất quan trọng. Nếu thiết kế không tốt ngay từ đầu, bạn sẽ gặp khó khăn khi áp dụng Clean Architecture vào dự án.
* Đôi khi tạo ra sự phức tạp không cần thiết: Đối với những dự án nhỏ hoặc đơn giản, Clean Architecture có thể gây ra sự phức tạp không cần thiết. Việc chia tách quá mức các thành phần có thể dẫn đến việc mã nguồn trở nên rườm rà, gây khó khăn cho việc phát triển nhanh chóng.
* Tăng độ trễ và khó quản lý khi quy mô nhỏ: Khi xây dựng một dự án có quy mô nhỏ, các lớp riêng biệt có thể tạo ra nhiều lớp gọi và dữ liệu trung gian, làm giảm hiệu suất và tăng độ phức tạp. Điều này không lý tưởng khi cần ra mắt một sản phẩm nhanh chóng.
* Chi phí học hỏi và đào tạo: Nếu đội ngũ phát triển không quen thuộc với Clean Architecture, sẽ mất thời gian để làm quen và đào tạo. Các lập trình viên sẽ cần hiểu rõ về các khái niệm và cách tổ chức mã nguồn.

1. **So sánh với kiến trúc Microservices**

|  |  |  |
| --- | --- | --- |
| **Tiêu chí** | **Clean Architecture** | **Microservices Architecture** |
| **Mục đích** | Tổ chức mã nguồn trong một ứng dụng đơn lẻ. | Phân tách ứng dụng thành các dịch vụ nhỏ, độc lập. |
| **Cấu trúc** | Các lớp như Entities, Use Cases, Interfaces, Infrastructure. | Các dịch vụ độc lập, mỗi dịch vụ làm một chức năng riêng. |
| **Phạm vi** | Áp dụng trong một ứng dụng đơn lẻ. | Áp dụng cho hệ thống phân tán, nhiều dịch vụ. |
| **Quản lý** | Dễ dàng bảo trì, mở rộng trong phạm vi ứng dụng đơn lẻ. | Dễ dàng phát triển, triển khai độc lập cho từng dịch vụ. |
| **Mở rộng** | Mở rộng dễ dàng nhưng trong phạm vi ứng dụng đơn lẻ. | Dễ dàng mở rộng hệ thống bằng cách thêm dịch vụ mới. |
| **Triển khai** | Thường chỉ triển khai một ứng dụng. | Triển khai nhiều dịch vụ, mỗi dịch vụ có thể chạy độc lập. |
| **Kiểm thử** | Kiểm thử dễ dàng nhờ vào việc tách biệt các lớp. | Kiểm thử yêu cầu kiểm tra cả hệ thống dịch vụ phân tán. |
| **Phụ thuộc vào công nghệ** | Không phụ thuộc vào công nghệ bên ngoài (Cơ sở dữ liệu, UI...). | Mỗi dịch vụ có thể sử dụng công nghệ khác nhau (Cơ sở dữ liệu, giao thức...). |
| **Ứng dụng** | Phù hợp với các ứng dụng nhỏ và vừa, một hệ thống đơn lẻ. | Phù hợp với các hệ thống lớn, phức tạp với nhiều dịch vụ. |

1. **Ứng dụng thực tế**

**Ứng dụng Thương mại điện tử**:

* **Entities**: Order, Product, Customer.
* **Use Cases**: **Tạo đơn hàng**, **Xử lý thanh toán**.
* **Infrastructure**: Cơ sở dữ liệu, Gateway thanh toán.

**Ứng dụng Quản lý Nhân sự**:

* **Entities**: Employee, Salary, Attendance.
* **Use Cases**: **Tính lương**, **Cập nhật thông tin nhân viên**.
* **Infrastructure**: Cơ sở dữ liệu nhân viên.

**Ứng dụng Y tế**:

* **Entities**: Patient, Doctor, Appointment.
* **Use Cases**: **Đặt lịch khám**, **Cập nhật bệnh án**.
* **Infrastructure**: Cơ sở dữ liệu bệnh án.

**Ứng dụng Tài chính**:

* **Entities**: Transaction, Account, Invoice.
* **Use Cases**: **Gửi tiền**, **Chuyển khoản**.
* **Infrastructure**: API thanh toán, hệ thống cơ sở dữ liệu ngân hàng.

**5. Kết luận**

Clean Architecture rất phù hợp với các dự án lớn, dài hạn, có yêu cầu mở rộng và bảo trì trong tương lai. Tuy nhiên, đối với các dự án nhỏ, đơn giản, việc áp dụng Clean Architecture có thể khiến hệ thống trở nên phức tạp hơn mức cần thiết và không mang lại lợi ích lớn.

Do đó, quyết định áp dụng Clean Architecture hay không phụ thuộc vào quy mô và yêu cầu của dự án.