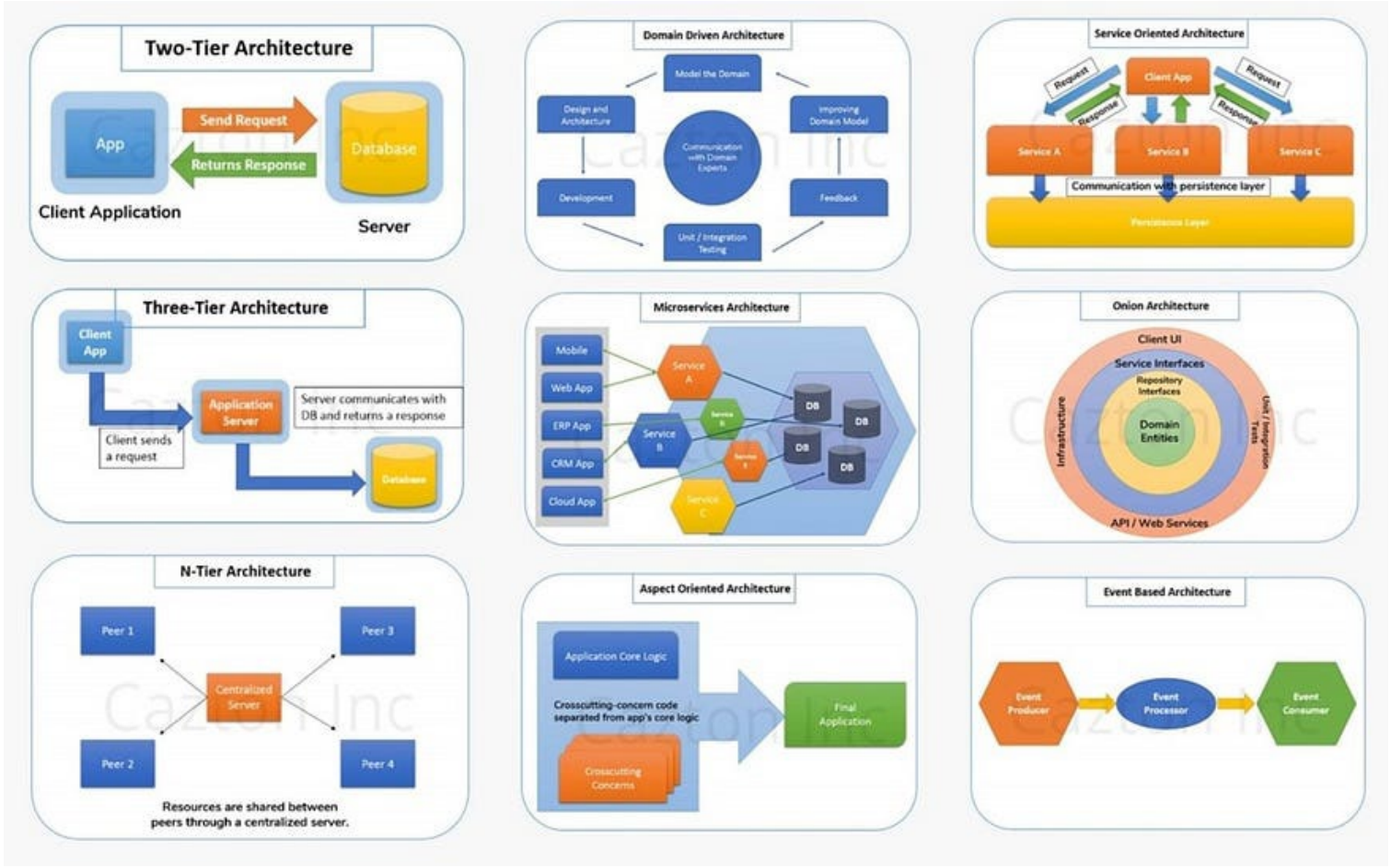
การออกแบบซอฟต์แวร์
(System Design & Architecture)

วัตถุประสงค์การเรียนรู้

- **ความเข้าใจ Software Architecture:** อธิบายรูปแบบสถาปัตยกรรมต่างๆ และเมื่อใดควรใช้
- **Design Patterns Mastery:** เลือกและนำ Design Patterns ไปใช้อย่างเหมาะสม
- **API Design:** ออกแบบ RESTful API ที่ปลอดภัย และ Scalable
- **Database Schema:** ออกแบบ Database ที่ normalized และ efficient
- **Architecture Documentation:** เขียน Architecture Document ที่ชัดเจนและครบถ้วน

Software Architecture Fundamentals



https://cazton.com/consulting/enterprise/software-architecture

What is Software Architecture?

- **นิยาม:** Architecture คือโครงสร้างที่อธิบายว่าระบบประกอบด้วยส่วนอะไร และส่วนต่างๆ ทำงานร่วมกันอย่างไร

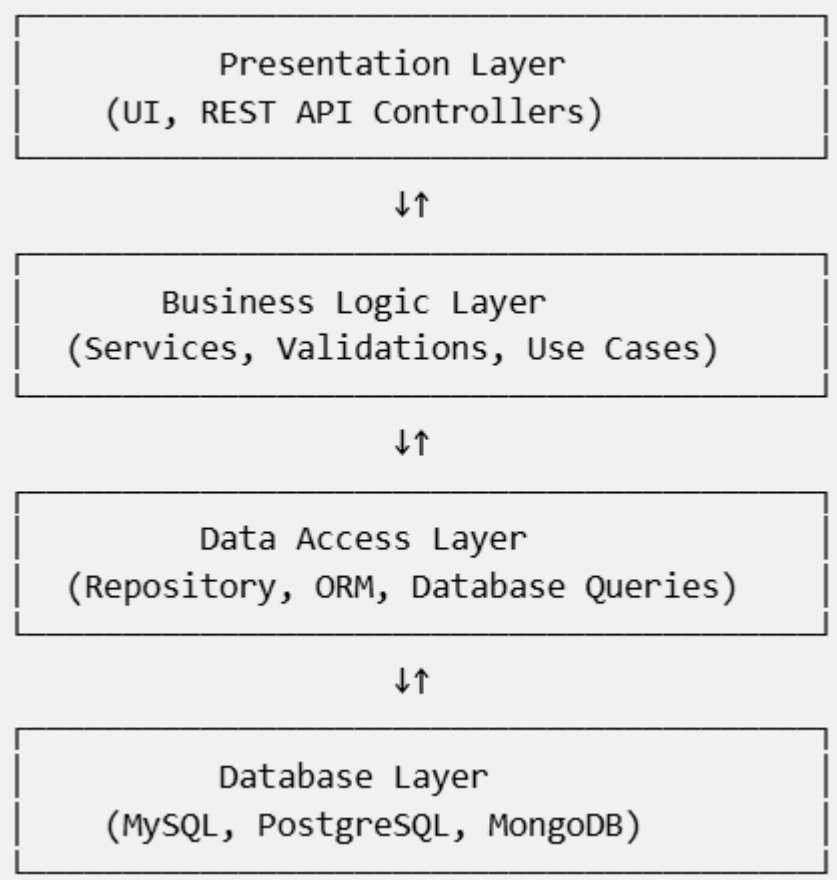
- **ทำไมสำคัญ**
  - Architecture ที่ดี = Maintainability, Scalability, Testability ดี
  - Architecture ที่ไม่ดี = Chaos, Technical Debt, ยากในการ Debug/Extend
  - 70% ของต้นทุนในการบำรุงรักษา = เลือก Architecture

- **Bad Architecture:**
  - All code in one God Class
  - No separation of concerns
  - Everything talks to database directly
  - Result: Code becomes unmaintainable

- **Good Architecture:**
  - Clear layers (UI, Business, Data)
  - Separation of concerns
  - Dependency Injection
  - Result: Easy to test, modify, extend

Architectural Patterns

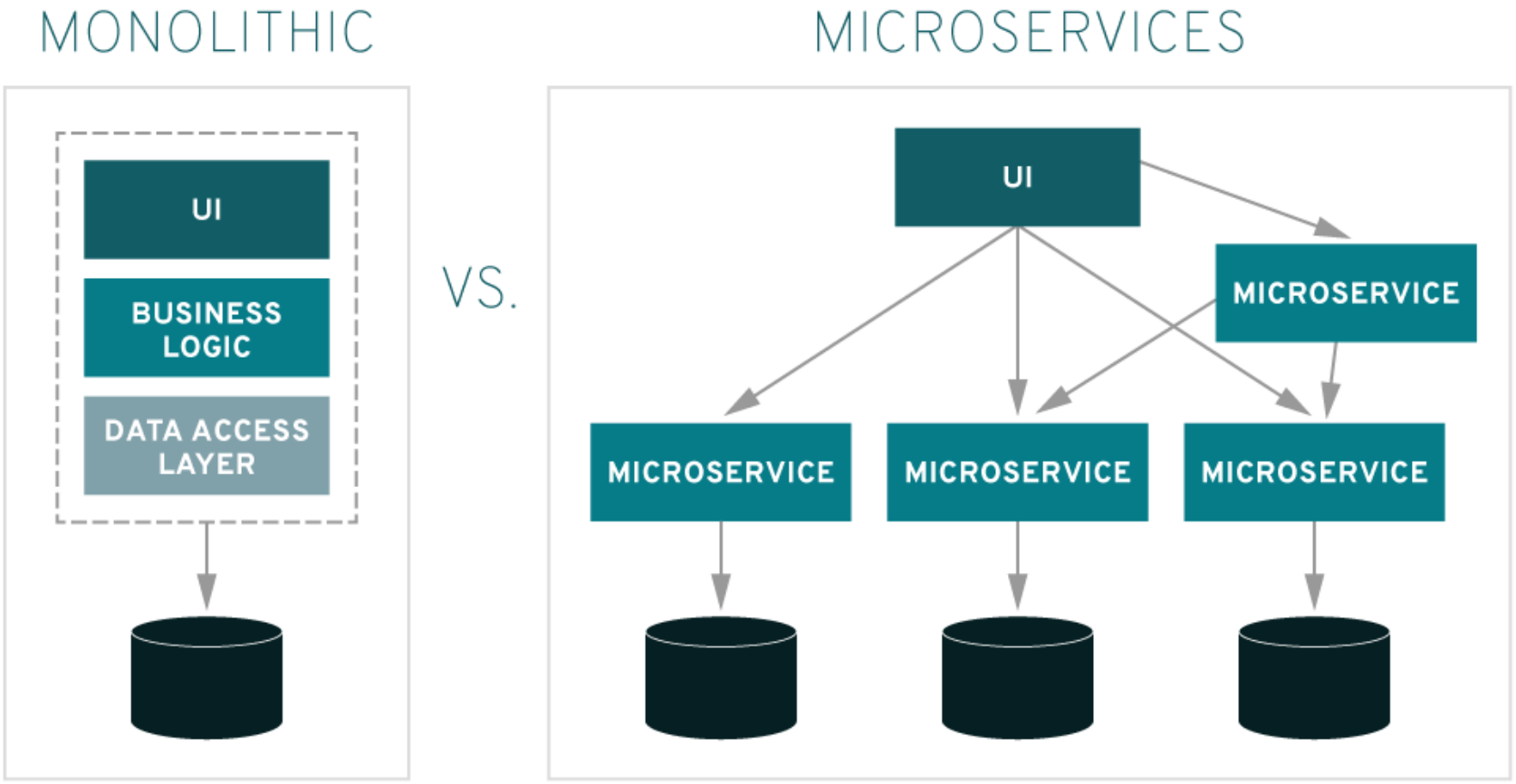
- **Pattern 1: Layered Architecture (3-Tier)**




```
Pros:
- Simple & Easy to understand
- Good for small-medium projects
- Easy to organize teams

Cons:
- Can become monolithic
- Limited scalability
- Tight coupling between layers
```

Architectural Patterns

- **Pattern 2: Microservices Architecture**

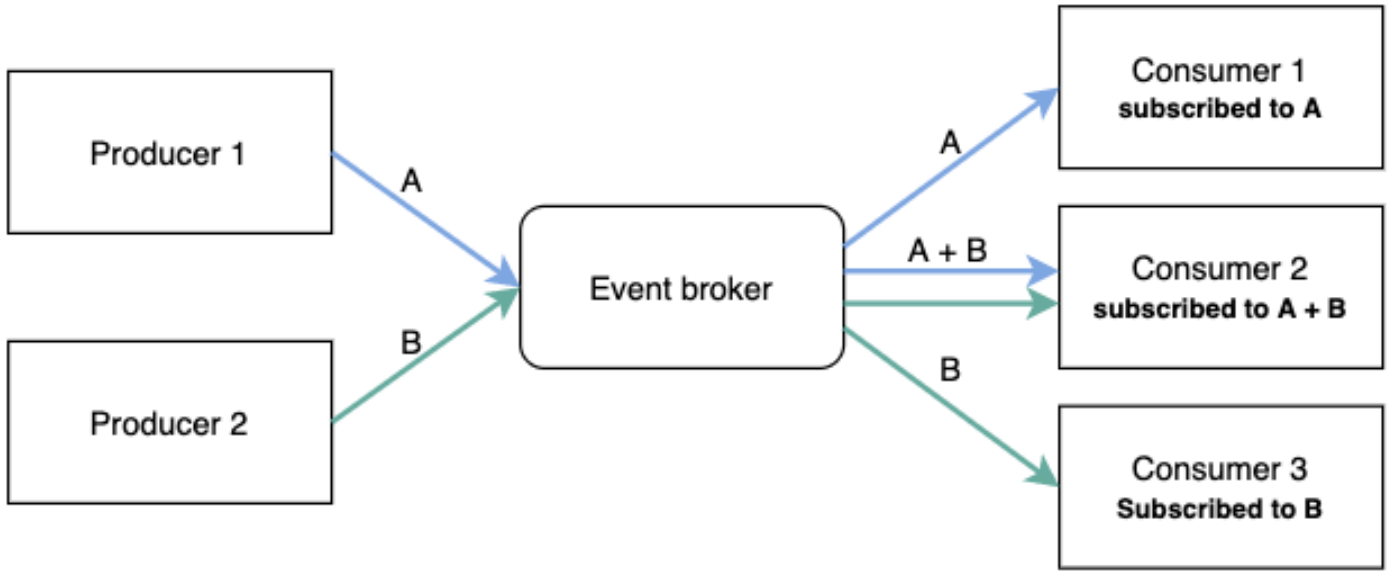



```
Pros:
- High scalability
- Independent deployment
- Technology diversity
- Fault isolation

Cons:
- Complex networking
- Distributed transactions
- Operational complexity
- Monitoring challenges
```

https://www.redhat.com/en/topics/microservices/what-are-microservices

Architectural Patterns

- **Pattern 3: Event-Driven Architecture**

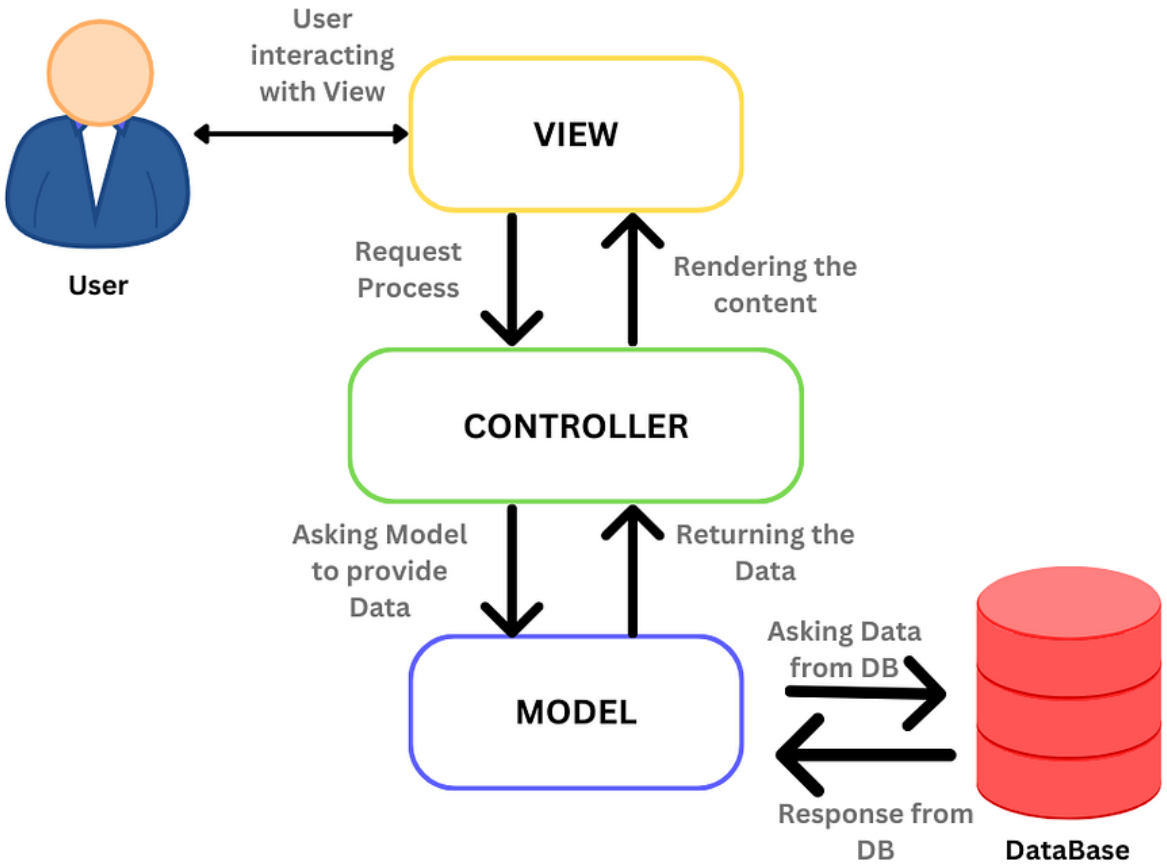



```
Pros:
- Loose coupling
- High responsiveness
- Real-time processing

Cons:
- Complex debugging
- Eventual consistency issues
- Harder to trace flows
```

https://blog.hubspot.com/marketing/event-driven-architecture

Architectural Patterns

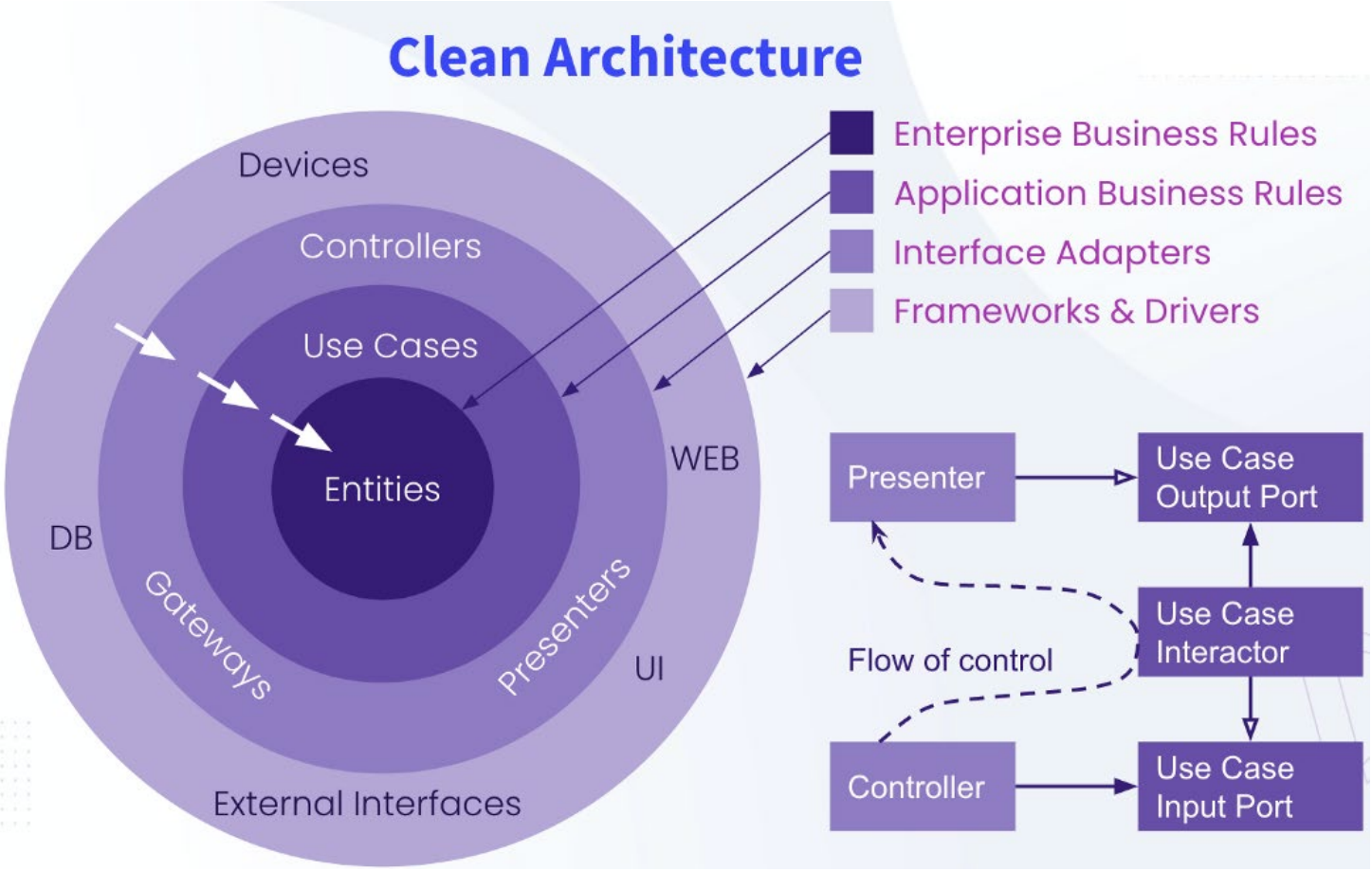
- **Pattern 4: MVC (Model–View–Controller)**



https://medium.com/@sadikarahmantanisha/the-mvc-architecture-97d47e071eb2

Architectural Patterns

- **Pattern 5: Clean Architecture**

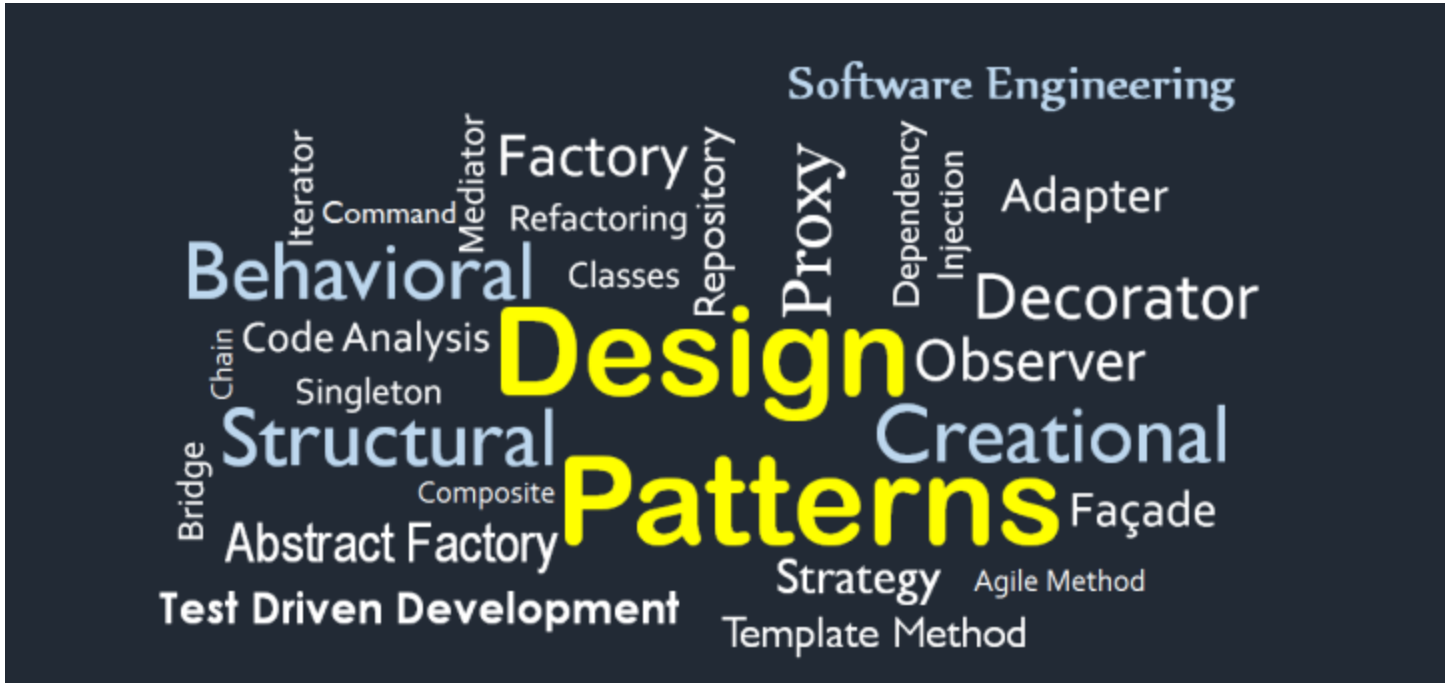


https://semihtekin.medium.com/what-is-the-clean-architecture-c80c2a2ff69a

Architecture selection criteria

- **ใช้ Layered Architecture เมื่อ:**
  - Project ขนาดเล็ก-กลาง
  - Single team ≤ 10 คน
  - Monolithic approach suitable
  - โครงการต้องออกแบบได้ รวดเร็ว
- **ใช้ Microservices เมื่อ:**
  - Project มี 100+ developers
  - Multiple independent teams
  - Different scaling needs per service
  - Technology diversity needed
  - Independent deployment crucial
- **ใช้ Event-Driven เมื่อ:**
  - Real-time processing needed
  - Multiple systems need to react to events
  - Asynchronous processing beneficial
  - Loose coupling critical

Design Patterns



What are Design Patterns?

นิยาม: Reusable solutions to common problems in software design.

1. **Creational Patterns** (How objects are created)

- **Singleton:** Ensure only one instance exists
- **Factory:** Create objects without specifying exact classes
- **Builder:** Build complex objects step by step

2. **Structural Patterns** (Object composition)

- **Adapter:** Make incompatible interfaces work together
- **Bridge:** Decouple abstraction from implementation
- **Facade:** Provide simplified interface to complex subsystem
- **Proxy:** Provide placeholder for another object

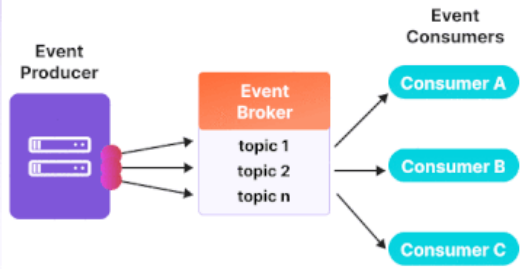
3. **Behavioral Patterns** (Object interaction)

- **Observer:** Notify multiple objects of state changes
- **Strategy:** Encapsulate algorithms
- **State:** Alter object behavior when state changes
- **Template Method:** Define algorithm skeleton in base class

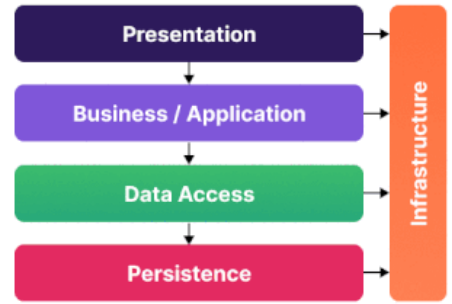
blog.amigoscode.com

SOFTWARE ARCHITECTURAL PATTERNS

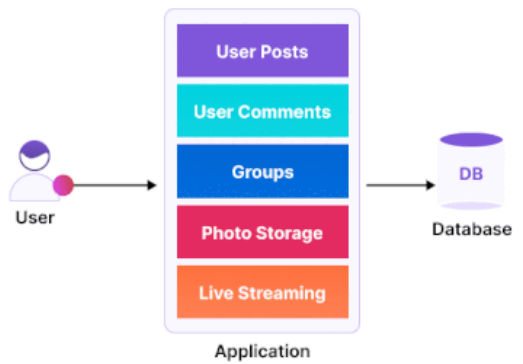
EVENT DRIVEN



LAYERED

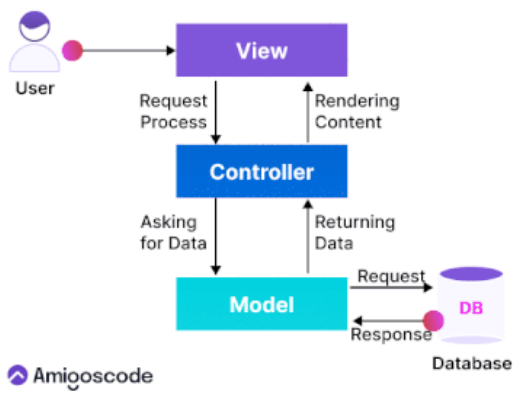


MONOLITHIC

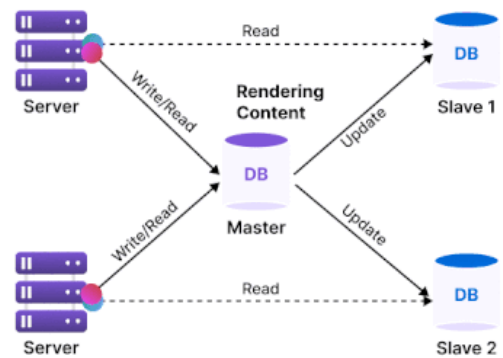


MICROSERVICE

MVC

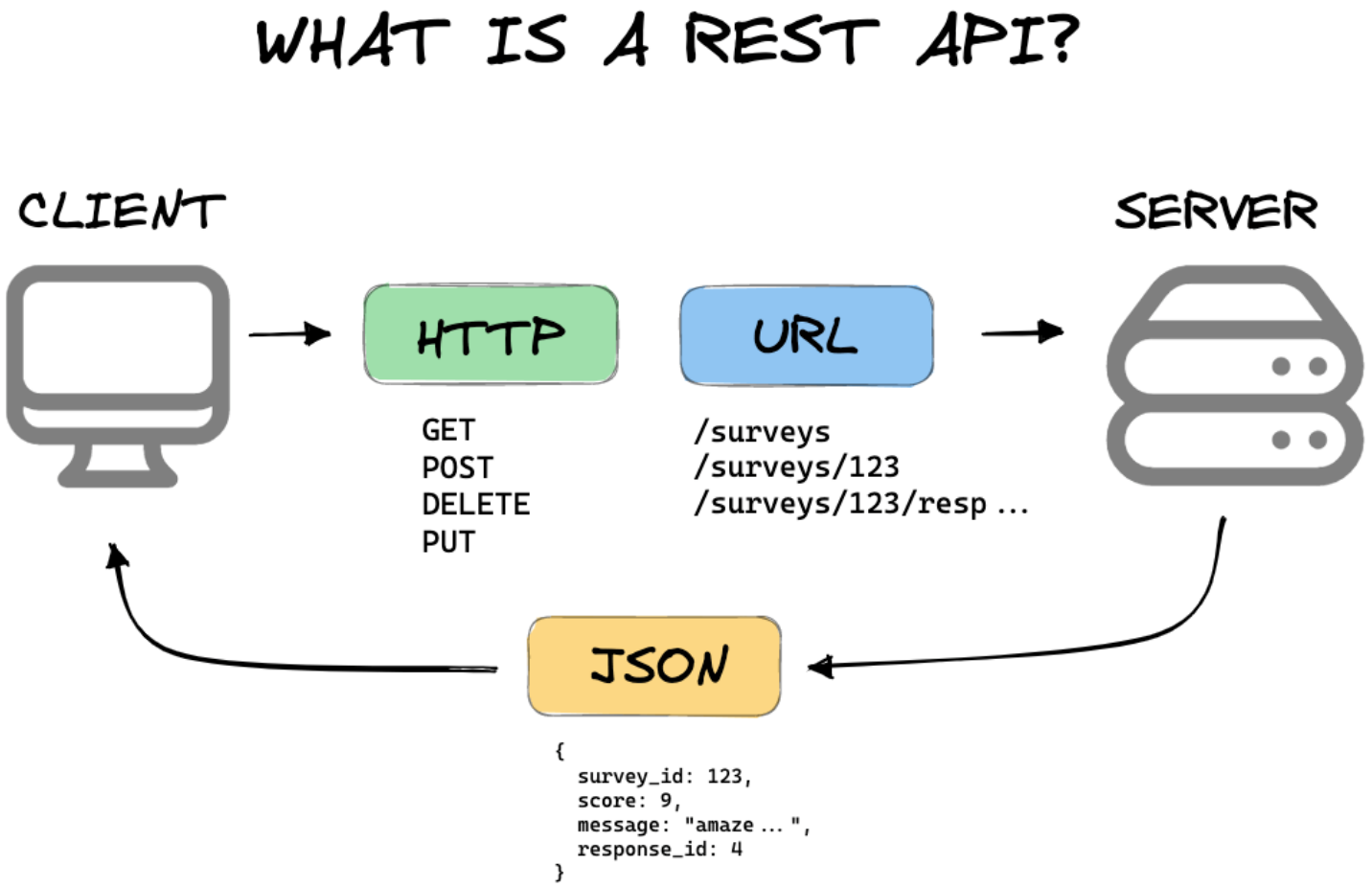


MASTER-SLAVE



Amigoscode

RESTful API Design



mannhowie.com

REST Principles

REST = Representational State Transfer

6 Constraints

- **Client–Server:** Separation of concerns
- **Statelessness:** Each request has all needed information
- **Uniform Interface:** Consistent API design
- **Cacheability:** Responses explicitly marked as cacheable
- **Layered System:** Client cannot tell if connected to end server
- **Code on Demand:** Servers can extend client functionality (optional)

RESTful API Best Practices

- **Use Proper HTTP Verbs**

```
GET /api/users          → Retrieve list of users
GET /api/users/1        → Get user with ID 1
POST /api/users         → Create new user
PUT /api/users/1        → Update user 1 completely
PATCH /api/users/1      → Partially update user 1
DELETE /api/users/1     → Delete user 1
```

RESTful API Best Practices

- **Use Proper HTTP Status Codes**

```
200 OK                    → Request successful
201 Created               → New resource created
204 No Content            → Success, no response body
400 Bad Request           → Client error in request
401 Unauthorized          → Authentication required
403 Forbidden             → Authenticated but not allowed
404 Not Found             → Resource doesn't exist
500 Server Error          → Server error
503 Service Unavailable → Server temporarily down
```

RESTful API Best Practices

- **Resource Naming**

```
GOOD:
/api/v1/users
/api/v1/users/123/orders
/api/v1/orders?status=pending

BAD:
/api/getUser            (verb in URL)
/api/user/123/data      (inconsistent naming)
/api/Users/123          (inconsistent casing)
```

RESTful API Best Practices

- **Versioning**

```
URL-based (Recommended for course):
/api/v1/users
/api/v2/users

Header-based:
Accept: application/vnd.myapp.v1+json
```

RESTful API Best Practices

- **Pagination & Filtering**

```
Pagination:
GET /api/users?page=1&limit=10
GET /api/users?offset=0&limit=10

Filtering:
GET /api/users?status=active&role=admin

Sorting:
GET /api/users?sort=name&order=asc
```

RESTful API Best Practices

- **Error Responses**

```
{
  "error": {
    "code": "INVALID_REQUEST",
    "message": "Email already exists",
    "details": "The email address test@example.com is already registered"
  }
}
```

API Documentation

- Example Endpoint:

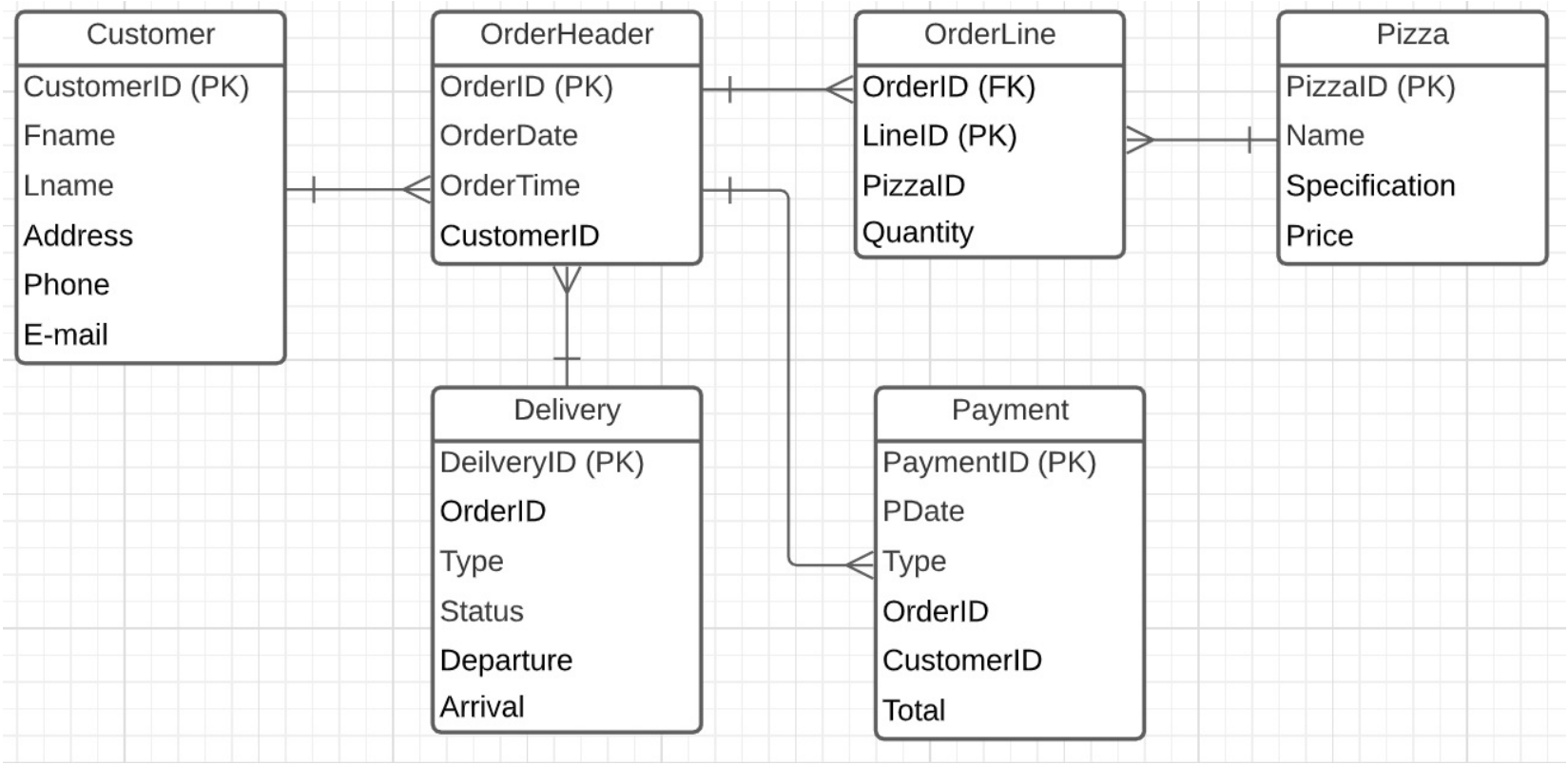
```
Endpoint: Get User Profile
Method: GET
URL: /api/v1/users/{userId}

Parameters:
- userId (path, required, integer): The user ID

Response (200):
{
  "id": 1,
  "email": "john@example.com",
  "name": "John Doe",
  "role": "admin",
  "created_at": "2025-01-01T00:00:00Z"
}

Error (404):
{
  "error": {
    "code": "USER_NOT_FOUND",
    "message": "User with ID 1 not found"
  }
}
```

Database Design

- **Database Normalization**
  - **Unnormalized:** Data scattered, redundancy
  - **1NF:** Atomic values, no repeating groups
  - **2NF:** 1NF + No partial dependencies
  - **3NF:** 2NF + No transitive dependencies
  - **BCNF:** 3NF + Every determinant is candidate key

Database Design

- **Entity-Relationship Diagram**



Database Design

- **SQL Schema Definition**

```sql
-- Create Customer table
CREATE TABLE Customer (
  id INT PRIMARY KEY AUTO_INCREMENT,
  name VARCHAR(100) NOT NULL,
  email VARCHAR(100) UNIQUE NOT NULL,
  phone VARCHAR(20),
  address TEXT,
  created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
  INDEX (email)
);

-- Create Order table
CREATE TABLE Order (
  id INT PRIMARY KEY AUTO_INCREMENT,
  customer_id INT NOT NULL,
  order_date TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
  total_price DECIMAL(10, 2),
  status ENUM('pending', 'completed', 'cancelled') DEFAULT 'pending',
  FOREIGN KEY (customer_id) REFERENCES Customer(id),
  INDEX (customer_id),
  INDEX (status)
);
```

```sql
-- Create Product table
CREATE TABLE Product (
  id INT PRIMARY KEY AUTO_INCREMENT,
  name VARCHAR(100) NOT NULL,
  description TEXT,
  price DECIMAL(10, 2) NOT NULL,
  stock INT DEFAULT 0,
  created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
  INDEX (name)
);

-- Create OrderItem table (junction table)
CREATE TABLE OrderItem (
  id INT PRIMARY KEY AUTO_INCREMENT,
  order_id INT NOT NULL,
  product_id INT NOT NULL,
  quantity INT NOT NULL,
  price DECIMAL(10, 2) NOT NULL,
  FOREIGN KEY (order_id) REFERENCES Order(id),
  FOREIGN KEY (product_id) REFERENCES Product(id),
  UNIQUE KEY (order_id, product_id)
);
```

SOLID Principles

- **SOLID คืออะไร – Acronym สำหรับ 5 principles**
  - **S** – Single Responsibility Principle (SRP)
  - **O** – Open/Closed Principle (OCP)
  - **L** – Liskov Substitution Principle (LSP)
  - **I** – Interface Segregation Principle (ISP)
  - **D** – Dependency Inversion Principle (DIP)

>>>>> Continue reading SOLID Principles

Activity 1: Design System Architecture

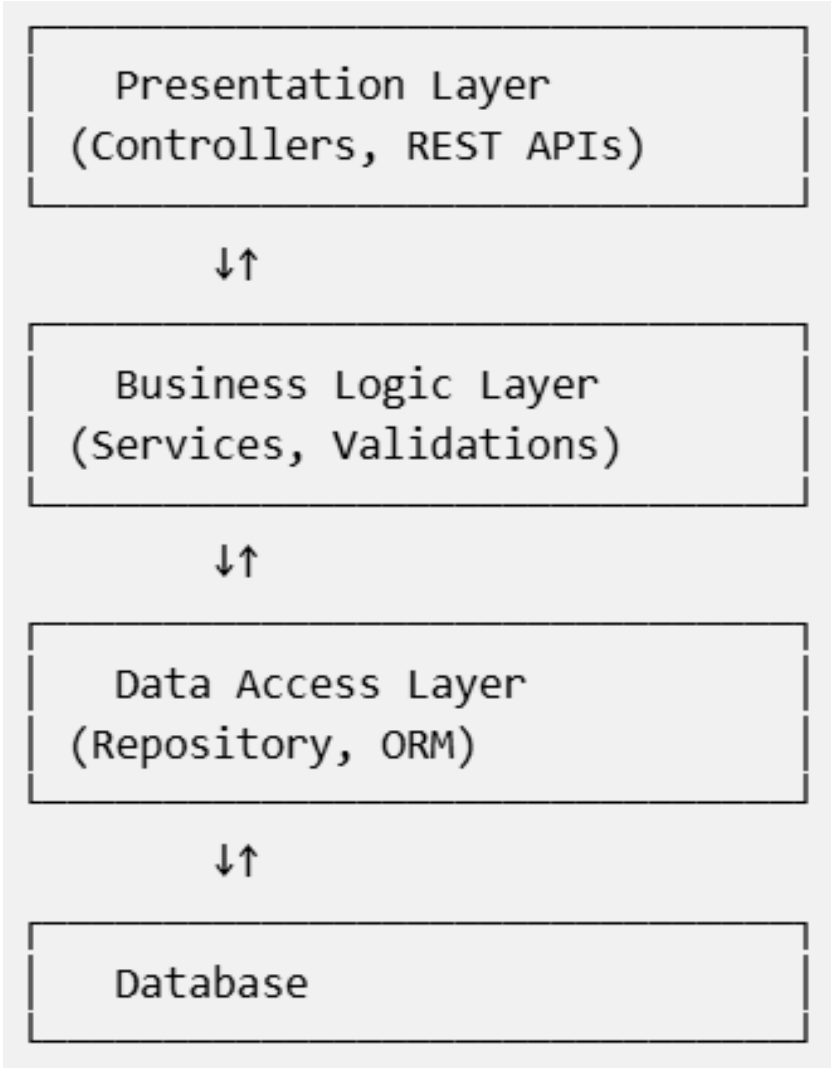
- **วัตถุประสงค์:** ทีมออกแบบ Architecture ของโครงการและสร้าง Architecture Diagram

- **Part 1: Select Architecture Pattern**
- ทีมเลือก Layered, Microservices, Event-Driven, หรือ MVC
- Justify ทำไมเลือก pattern นี้
  - Team size?
  - Scalability needs?
  - Complexity?
- บันทึก: Architecture_Pattern_Justification.md

Activity 1: Design System Architecture

- **Part 2: Create Architecture Diagram**
  - ใช้ Draw.io, Lucidchart, หรือ PlantUML
  - แสดง layers, components, relationships
  - Label ให้ชัดเจน

```
Components:
- Controllers/Routes
- Service Classes
- Repository Classes
- Models/Entities
- Database Tables
```



Activity 2: API Design & Database Schema

- **วัตถุประสงค์:** ออกแบบ API endpoints และ Database schema
- **ลำดับการทำ**
- **Part 1: List API Endpoints**

```
For E-Commerce:

User API:
- POST /api/v1/auth/register
- POST /api/v1/auth/login
- GET /api/v1/users/{id}
- PUT /api/v1/users/{id}
- DELETE /api/v1/users/{id}

Product API:
- GET /api/v1/products
- GET /api/v1/products/{id}
- POST /api/v1/products (admin)
- PUT /api/v1/products/{id} (admin)
- DELETE /api/v1/products/{id} (admin)

Order API:
- POST /api/v1/orders
- GET /api/v1/orders/{id}
- GET /api/v1/orders (user's orders)
- PUT /api/v1/orders/{id}/status (admin)
```

Activity 2: API Design & Database Schema

- **Part 2: API Specification**
  - For 2–3 main endpoints

```
POST /api/v1/orders

Request:
{
  "items": [
    {"product_id": 1, "quantity": 2},
    {"product_id": 5, "quantity": 1}
  ]
}

Response (201):
{
  "id": 123,
  "user_id": 45,
  "items": [...],
  "total": 250.00,
  "status": "pending",
  "created_at": "2025-01-01T10:00:00Z"
}

Error (400):
{
  "error": "Invalid product ID"
}
```

Activity 2: API Design & Database Schema

- **Create ER Diagram**
- Create simplified ER Diagram showing
  - Tables/Entities
  - Relationships (1:1, 1:M, M:M)
  - Primary keys
  - Foreign keys

Deliverable:

- ✅ API_Endpoints.md (list all endpoints)
- ✅ API_Specification.md (detail 5-10 main endpoints)
- ✅ ER_Diagram.* (PNG/SVG)
- ✅ GitHub: docs/ folder

การบ้าน 1 : Complete API Specification

งาน:

1. **ระบุ** API Endpoints **ทั้งหมด** (15-20 endpoints)
2. **สำหรับแต่ละ** endpoint, **เขียน:**
    - Method (GET, POST, PUT, DELETE, PATCH)
    - URL path
    - Parameters (path, query, body)
    - Request body example
    - Response examples (200, 400, 401, 404, 500)
    - HTTP status codes
    - Authentication required (yes/no)
    - Rate limiting (if applicable)
3. Group endpoints by resource
4. Include pagination, filtering, sorting specifications

Deliverable:

- ✅ Complete_API_Specification.md (5-10 pages)
- ✅ GitHub: docs/API_Specification.md

การบ้าน 2: Database Schema Design

งาน:

1. Create detailed ER Diagram (normalized to 3NF):
    - Identify all entities
    - Attributes per entity
    - Primary keys & Foreign keys
    - Relationships (1:1, 1:M, M:M)
    - Cardinality
2. Write SQL CREATE statements:
    - All table definitions
    - Data types (appropriate ones)
    - Constraints (PRIMARY KEY, FOREIGN KEY, UNIQUE, NOT NULL)
    - Indexes (on frequently searched columns)
    - Default values

Deliverable:

- ✅ ER_Diagram.* (PNG/SVG with 3NF normalized)
- ✅ Database_Schema.sql (SQL CREATE statements)
- ✅ Schema_Design_Document.md (explain decisions)
- ✅ GitHub: docs/ folder

Project Assignment:
Create Architecture & Design Document

- **Assignment 6.1 : Complete Architecture Document**

งาน:

1. **Architecture Overview**
   - Pattern selected & justified (2-3 pages)
   - Architecture diagram with explanations
   - Layers/Components & responsibilities
   - Technology stack rationale
2. **Component Design**
   - Each layer: what it does, key classes/modules
   - Interfaces & contracts between layers
   - Dependency relationships
3. **Design Patterns Used**
   - List 3-5 design patterns (Singleton, Factory, Observer, etc.)
   - For each: where used, why chosen, diagram/code snippet
   - Benefits & trade-offs

Deliverable:

- ✅ Architecture_Design_Document.md (8-12 pages)
- ✅ Architecture diagrams (2-3)
- ✅ Component descriptions
- ✅ GitHub: docs/Architecture/ folder

Project Assignment:
Create Architecture & Design Document

- **Assignment 6.2: API Specification & Database Schema**

งาน:

Consolidate homework + activities:

1. **API Specification**
   - 15-20 endpoints documented completely
   - Consistent naming & structure
   - Error handling examples
   - Authentication strategy (JWT, API Key, etc.)
   - Rate limiting (if applicable)
2. **Database Schema**
   - Normalized ER diagram (3NF)
   - SQL CREATE statements (all tables)
   - Indexes identified & justified
   - Sample data inserts
3. **Integration**
   - How do APIs map to database?
   - Data flow examples
   - Transaction handling strategy

Deliverable:

```
GitHub structure:
docs/
├── API_Specification.md
├── Database_Schema.sql
├── Database_ER_Diagram.* (PNG/SVG)
└── Integration_Document.md
```

Architecture Patterns Comparison

Pattern	Complexity	Scalability	Team Size	Best For
Layered	Low	Medium	3-8	Small-medium projects
Microservices	High	Very High	20+	Large distributed systems
Event-Driven	High	High	8-15	Real-time processing
MVC	Low	Medium	2-5	Web applications
CQRS	Very High	Very High	15+	Complex domain logic

Design Patterns Summary

Pattern	Purpose	Difficulty
Singleton	One instance only	Easy
Factory	Create objects	Easy
Observer	Event notification	Medium
Strategy	Algorithm selection	Medium
Dependency Injection	Loose coupling	Medium
Builder	Complex objects	Medium-Hard
State	State management	Hard
Chain of Responsibility	Request handling	Hard

สรุป

Software Architecture:

- ✅ 5 common patterns: Layered, Microservices, Event-Driven, MVC, Clean
- ✅ Selection based on: team size, scalability, complexity
- ✅ Good architecture = maintainability, testability, scalability
- ✅ For this course: Layered architecture recommended

Design Patterns:

- ✅ 3 types: Creational, Structural, Behavioral
- ✅ Common patterns: Singleton, Factory, Observer, Strategy, DI
- ✅ Use patterns to solve recurring problems
- ✅ But avoid over-engineering with unnecessary patterns

RESTful APIs:

- ✅ 6 REST constraints guide API design
- ✅ Proper HTTP verbs & status codes
- ✅ Consistent resource naming & versioning
- ✅ Comprehensive error handling
- ✅ Clear documentation crucial

Database Design:

- ✅ Normalize to 3NF (or BCNF)
- ✅ Create proper ER diagrams
- ✅ Use relationships (1:1, 1:M, M:M)
- ✅ Add indexes for performance
- ✅ SQL CREATE statements complete

SOLID Principles:

- ✅ Write maintainable, extensible code
- ✅ Single Responsibility, Open/Closed, Liskov, Interface Segregation, DI
- ✅ Apply SOLID to improve code quality

D2 Deliverable Readiness:

- ✅ Architecture document complete (8-12 pages)
- ✅ API specification detailed (15-20 endpoints)
- ✅ Database schema normalized (3NF)
- ✅ Code skeleton started
- ✅ Coding standards documented
- ✅ Git workflow ready