Introduzione

Cos'è un Architecture Pattern

Un **Architecture Pattern** definisce la **struttura fondamentale** di un sistema software a livello macroscopico. A differenza dei design pattern (che operano a livello di classi/oggetti), gli architecture pattern operano a livello di **componenti e sottosistemi**.

Differenza: Design Pattern vs Architecture Pattern

Aspetto	Design Pattern	Architecture Pattern
Scope	Classe/Modulo	Sistema completo
Granularità	Fine (micro)	Grossa (macro)
Impatto	Locale	Globale
Esempi	Singleton, Factory	Layered, Microservices
Quando	Durante implementazione	Durante progettazione sistema

Perché Servono

Problemi senza pattern architetturali:

- Crescita incontrollata della complessità
- Difficoltà di manutenzione e scalabilità
- Deployment complicato
- Testing difficile
- Accoppiamento elevato tra componenti

Benefici:

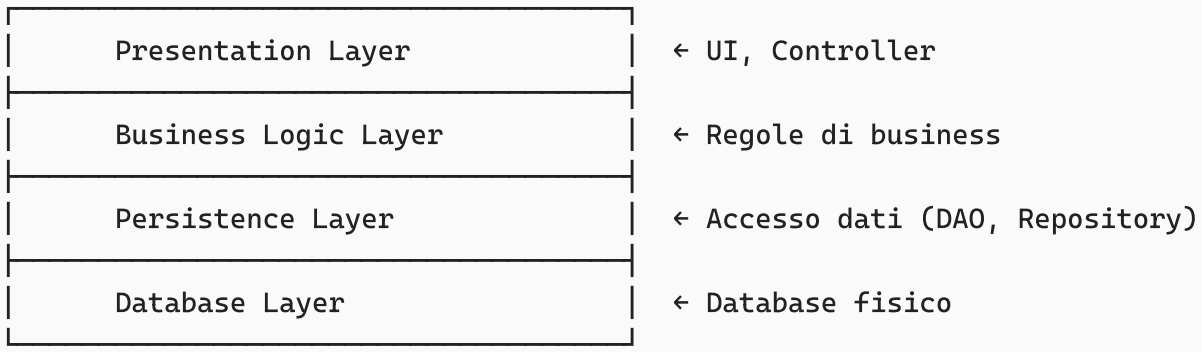
- Struttura chiara e riconoscibile
- Decisioni architetturali documentate
- Facilitano comunicazione nel team
- Guidano l'implementazione

Layered Architecture

Definizione

Il pattern **Layered** (a livelli) organizza il sistema in **strati orizzontali**, ciascuno con responsabilità specifiche. Ogni livello può comunicare **solo con il livello immediatamente sottostante**.

Struttura



Flusso di comunicazione:

```
Presentation → Business Logic → Persistence → Database
```

Componenti Tipici

1. Presentation Layer

Responsabilità: interfaccia utente e gestione richieste.

Contenuto:

- Controller (MVC)
- View templates
- Validazione input
- Serializzazione risposte (JSON, XML)

Esempio (Spring MVC):

```java
@RestController
@RequestMapping("/api/studenti")
public class StudenteController {
    @Autowired
    private StudenteService service;

    @GetMapping("/{id}")
    public ResponseEntity<StudenteDTO> getStudente(@PathVariable Long id) {
        return ResponseEntity.ok(service.getStudente(id));
    }
}
```

2. Business Logic Layer

Responsabilità: logica applicativa e regole di business.

Contenuto:

- Service classes
- Domain logic
- Transazioni
- Validazioni business
- Orchestrazione

Esempio:

```
@Service
@Transactional
public class StudenteService {
    @Autowired
    private StudenteRepository repository;

    public StudenteDTO getStudente(Long id) {
        Studente entity = repository.findById(id)
            .orElseThrow(() -> new NotFoundException());

        // Business logic
        if (!entity.isAttivo()) {
            throw new InactiveStudentException();
        }

        return toDTO(entity);
    }

    public void iscriviACorso(Long studenteId, Long corsoId) {
        // Orchestrazione multi-entity
        Studente studente = repository.findById(studenteId).orElseThrow();
        Corso corso = corsoRepository.findById(corsoId).orElseThrow();

        // Business rule
        if (studente.getCorsi().size() >= 6) {
            throw new MaxCorsiException();
        }

        studente.addCorso(corso);
        repository.save(studente);
    }
}
```

3. Persistence Layer

Responsabilità: accesso e gestione dati persistenti.

Contenuto:

- Repository / DAO
- Query SQL/ORM
- Mapping entity ↔ database

Esempio:

```
@Repository
public interface StudenteRepository extends JpaRepository<Studente, Long> {
    List<Studente> findByCorsoId(Long corsoId);

    @Query("SELECT s FROM Studente s WHERE s.età > :age")
    List<Studente> findMaggiorenni(@Param("age") int age);
}
```

4. Database Layer

Responsabilità: storage fisico.

Contenuto:

- RDBMS (PostgreSQL, MySQL)
- NoSQL (MongoDB, Cassandra)
- File system

Varianti

Closed Layers (Strict Layering)

Regola: ogni layer può comunicare **solo** con quello immediatamente sotto.

```
A → B → C → D   ✓ OK
A → C           ✗ VIETATO
```

Vantaggi:

- Disaccoppiamento massimo
- Facile sostituzione layer

Svantaggi:

- Performance: attraversare molti layer

Open Layers (Relaxed Layering)

Regola: un layer può **saltare** layer intermedi se necessario.

```
A → B → C → D   ✓ OK
A → C           ✓ OK (per performance)
```

Vantaggi:

- Performance migliori
- Flessibilità

Svantaggi:

- Accoppiamento aumenta
- Difficile manutenzione

Esempio Completo: Java EE

```
┌─────────────────────────────────────────┐
│  Web Layer (JSP/JSF/REST)               │
│  - StudenteController.java              │
├─────────────────────────────────────────┤
│  Service Layer                          │
│  - StudenteService.java                 │
│  - StudenteServiceImpl.java             │
├─────────────────────────────────────────┤
│  DAO/Repository Layer                   │
│  - StudenteDAO.java                     │
│  - StudenteDAOImpl.java                 │
├─────────────────────────────────────────┤
│  Domain Model                           │
│  - Studente.java (Entity)               │
├─────────────────────────────────────────┤
│  Database                               │
│  - PostgreSQL                           │
└─────────────────────────────────────────┘
```

Flusso richiesta:

1. **HTTP Request** → `StudenteController`
2. `StudenteController` → `StudenteService`
3. `StudenteService` → `StudenteDAO`
4. `StudenteDAO` → **Database** (query)

5. **Database** → `StudenteDAO` (result set)
6. `StudenteDAO` → `StudenteService` (entity)
7. `StudenteService` → `StudenteController` (DTO)
8. `StudenteController` → **HTTP Response** (JSON)

Analisi del Pattern

Vantaggi ✅

- **Separazione responsabilità** chiara
- **Testabilità** elevata (mock dei layer)
- **Manutenibilità**: modifiche locali
- **Riusabilità**: business logic indipendente da UI
- Pattern **ben conosciuto**: facile onboarding
- **Allineamento organizzativo**: team per layer

Svantaggi ❌

- **Monolite**: deployment unico e completo
- **Scalabilità verticale**: scala tutto o niente
- **Performance**: overhead di attraversamento layer
- **Granularità grossa**: modifiche piccole richiedono re-deploy completo
- **Accoppiamento nascosto**: dipendenze trasversali tra layer

Rating

Criterio	Rating	Note
Agility	⭐ ⭐	Modifiche piccole difficili
Deployment	⭐	Monolite, difficile continuous deployment
Testability	⭐ ⭐ ⭐ ⭐	Facile mock dei layer
Performance	⭐ ⭐	Overhead dei layer
Scalability	⭐	Solo verticale
Ease of Development	⭐ ⭐ ⭐ ⭐ ⭐	Pattern noto, semplice

Quando Usare Layered

✅ **Usa Layered quando:**

- Applicazioni di **piccola/media dimensione**
- Budget limitato
- Team piccolo (< 10 persone)
- Requisiti di scalabilità modesti
- Dominio applicativo semplice
- Preferenza per semplicità

❌ **Evita Layered quando:**

- Microservices necessari
- Scalabilità orizzontale critica
- Deployment continuo richiesto
- Team distribuiti geograficamente
- Dominio complesso e in evoluzione

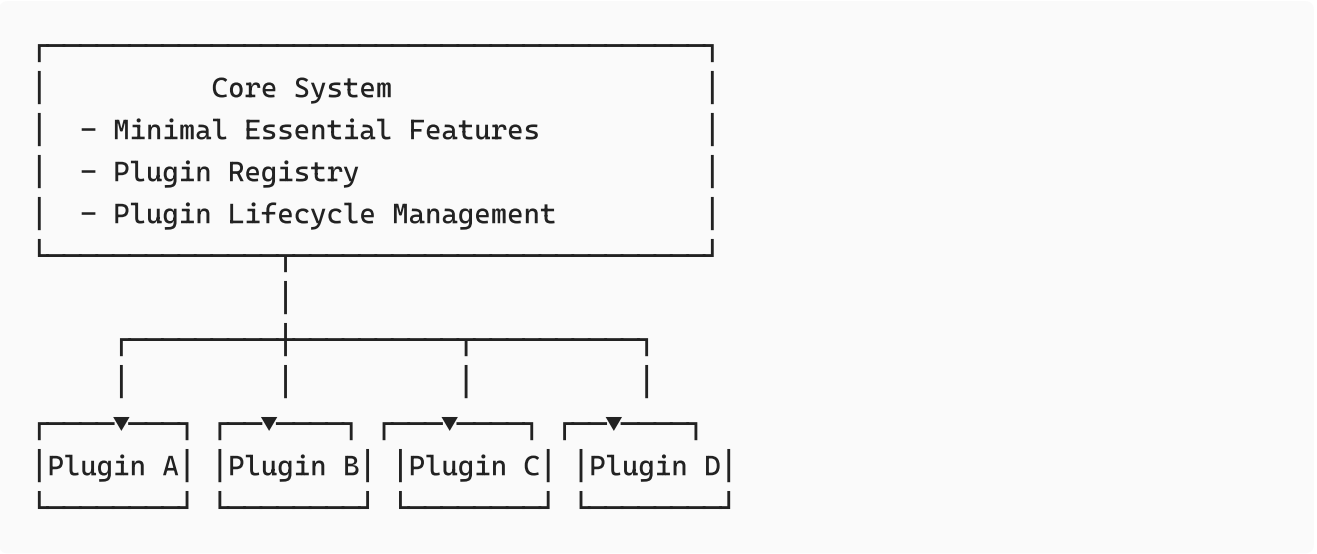
Microkernel Architecture

Definizione

Il pattern **Microkernel** (o **Plugin Architecture**) separa il sistema in:

- **Core System**: funzionalità minime essenziali
- **Plugin Modules**: estensioni opzionali e intercambiabili

Struttura



Caratteristiche:

- Core minimo e stabile
- Plugin aggiunti/rimossi dinamicamente

- Isolamento tra plugin

Componenti

1. Core System

Responsabilità: funzionalità minime indispensabili.

Contenuto:

- **Plugin registry**: registro plugin disponibili
- **Lifecycle management**: caricamento/scaricamento plugin
- **API**: interfaccia per plugin
- **Routing**: instradare richieste al plugin corretto

Esempio (Eclipse IDE):

```
public class PluginManager {
    private Map<String, Plugin> plugins = new HashMap<>();

    public void registerPlugin(String name, Plugin plugin) {
        plugins.put(name, plugin);
        plugin.onLoad();
    }

    public void unregisterPlugin(String name) {
        Plugin plugin = plugins.remove(name);
        if (plugin != null) {
            plugin.onUnload();
        }
    }

    public void executePlugin(String name, Context ctx) {
        Plugin plugin = plugins.get(name);
        if (plugin != null) {
            plugin.execute(ctx);
        }
    }
}
```

2. Plugin Modules

Responsabilità: funzionalità specifiche estese.

Caratteristiche:

- Implementano interfaccia comune
- Indipendenti tra loro
- Possono essere sviluppati da terze parti

Esempio:

```
public interface Plugin {
    void onLoad();
    void onUnload();
    void execute(Context ctx);
    String getName();
    String getVersion();
}

public class PDFExportPlugin implements Plugin {
    @Override
    public void execute(Context ctx) {
        // Esporta documento come PDF
        Document doc = ctx.getDocument();
        PDFGenerator.generate(doc, ctx.getOutputStream());
    }

    @Override
    public String getName() {
        return "PDF Exporter";
    }

    @Override
    public String getVersion() {
        return "1.2.0";
    }
}
```

Comunicazione Plugin-Core

Point-to-Point (contratti espliciti)

```
Core ←————→ Plugin A
Core ←————→ Plugin B
```

Pro: semplice, diretto
Contro: core conosce ogni plugin

Adapter Pattern

```
Core ⟷ Adapter ⟷ Plugin
```

Pro: core disaccoppiato dai plugin
Contro: complessità aggiuntiva

Esempi Reali

1. Eclipse IDE

- **Core**: piattaforma base (workspace, editor)
- **Plugin**: Java Development Tools, C++ Tools, Git integration

2. Visual Studio Code

- **Core**: editor di testo
- **Plugin**: estensioni per linguaggi (Python, Java), debugger, themes

3. Jenkins

- **Core**: CI/CD pipeline executor
- **Plugin**: Git, Docker, Kubernetes integrations

4. Web Browser

- **Core**: rendering engine
- **Plugin**: AdBlocker, Password Manager, Developer Tools

Analisi del Pattern

Vantaggi ✅

- **Estensibilità**: aggiungere features senza modificare core
- **Flessibilità**: attivare/disattivare funzionalità
- **Isolamento**: errori in un plugin non crashano il sistema
- **Testing**: testare plugin indipendentemente
- **Customization**: client personalizzano con plugin
- **Sviluppo parallelo**: team diversi su plugin diversi

Svantaggi ❌

- **Complessità architetturale**: gestione lifecycle complessa
- **Performance**: overhead di plugin loading

- **Versioning**: compatibilità plugin-core
- **Documentazione**: API core deve essere stabile e ben documentata

Rating

Criterio	Rating	Note
Agility	⭐ ⭐ ⭐ ⭐	Modifiche tramite plugin
Deployment	⭐ ⭐ ⭐ ⭐	Plugin indipendenti
Testability	⭐ ⭐ ⭐ ⭐	Plugin testabili isolatamente
Performance	⭐ ⭐ ⭐	Overhead loading
Scalability	⭐ ⭐	Limitata
Ease of Development	⭐ ⭐ ⭐	Richiede API stabile

Quando Usare Microkernel

✅ **Usa Microkernel quando:**

- Applicazione con **funzionalità opzionali**
- Personalizzazione per client diversi
- Terze parti devono estendere il sistema
- Feature evolvono indipendentemente
- Isolamento errori critico

❌ **Evita Microkernel quando:**

- Funzionalità fortemente interconnesse
- Core system instabile
- Overhead di plugin non accettabile
- Dominio troppo semplice

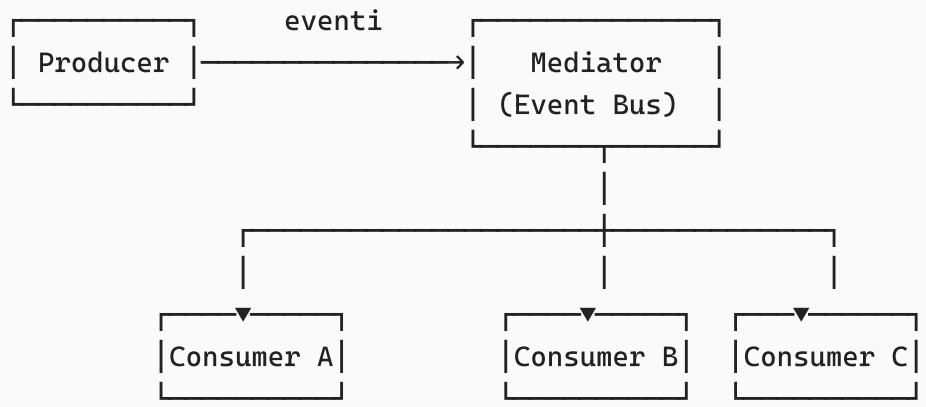
Event-Driven Architecture

Definizione

Event-Driven Architecture (EDA) organizza il sistema intorno a **eventi** e **reazioni asincrone** a tali eventi. I componenti comunicano tramite messaggi/eventi invece di chiamate dirette.

Struttura

Topologia 1: Mediator (Centralized)



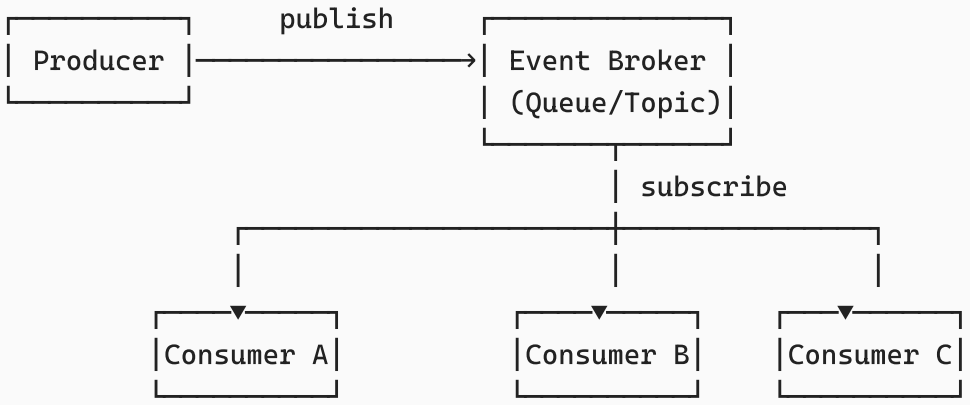
Caratteristiche:

- **Event mediator**: orchestratore centrale
- Conosce tutti i consumer
- Routing complesso
- Gestisce workflow multi-step

Esempio:

```
public class EventMediator {
    private Map<String, List<EventConsumer>> consumers = new HashMap<>();

    public void subscribe(String eventType, EventConsumer consumer) {
        consumers.computeIfAbsent(eventType, k -> new ArrayList<>())
                 .add(consumer);
    }

    public void publish(Event event) {
        List<EventConsumer> listeners = consumers.get(event.getType());
        if (listeners != null) {
            for (EventConsumer consumer : listeners) {
                // Può orchestrare flussi complessi
                if (shouldProcess(event, consumer)) {
                    consumer.handle(event);
                }
            }
        }
    }

    private boolean shouldProcess(Event event, EventConsumer consumer) {
        // Logica di orchestrazione
        return true;
    }
}
```

Topologia 2: Broker (Decentralized)



Caratteristiche:

- **Message broker**: instrada messaggi (RabbitMQ, Kafka)
- Producer non conosce consumer
- Disaccoppiamento massimo
- Scalabilità elevata

Esempio (Spring + RabbitMQ):

```
// Producer
@Service
public class OrderService {
    @Autowired
    private RabbitTemplate rabbitTemplate;

    public void placeOrder(Order order) {
        // Business logic
        orderRepository.save(order);

        // Pubblica evento
        OrderPlacedEvent event = new OrderPlacedEvent(order.getId());
        rabbitTemplate.convertAndSend("orders.exchange",
                                      "order.placed",
                                      event);
    }
}

// Consumer
@Component
public class EmailNotificationConsumer {
    @RabbitListener(queues = "email.queue")
    public void handleOrderPlaced(OrderPlacedEvent event) {
        // Invia email conferma
        emailService.sendOrderConfirmation(event.getOrderId());
```

```
    }
}

@Component
public class InventoryConsumer {
    @RabbitListener(queues = "inventory.queue")
    public void handleOrderPlaced(OrderPlacedEvent event) {
        // Aggiorna inventario
        inventoryService.reserveItems(event.getOrderId());
    }
}
```

Componenti

1. Event Producer

Responsabilità: generare eventi quando succede qualcosa.

Esempio:

```
public class UserRegistrationService {
    @Autowired
    private EventBus eventBus;

    public void registerUser(User user) {
        userRepository.save(user);

        // Pubblica evento
        UserRegisteredEvent event = new UserRegisteredEvent(
            user.getId(),
            user.getEmail(),
            LocalDateTime.now()
        );
        eventBus.publish(event);
    }
}
```

2. Event Consumer (Listener)

Responsabilità: reagire agli eventi.

Esempio:

```
@EventListener
public class WelcomeEmailConsumer {
    @Async
```

```
    public void onUserRegistered(UserRegisteredEvent event) {
        emailService.sendWelcomeEmail(event.getEmail());
        logger.info("Welcome email sent to {}", event.getEmail());
    }
}

@EventListener
public class StatisticsConsumer {
    @Async
    public void onUserRegistered(UserRegisteredEvent event) {
        statisticsService.incrementUserCount();
    }
}
```

3. Event Channel (Broker)

Responsabilità: trasportare eventi da producer a consumer.

Tecnologie:

- **RabbitMQ**: message broker AMQP
- **Apache Kafka**: distributed streaming platform
- **AWS SQS/SNS**: cloud messaging
- **Redis Pub/Sub**: in-memory messaging

Event Types

1. Domain Event

Eventi di business significativi.

Esempio:

```
public class OrderShippedEvent {
    private Long orderId;
    private String trackingNumber;
    private LocalDateTime shippedAt;
}
```

2. System Event

Eventi tecnici/infrastrutturali.

Esempio:

```java
public class DatabaseBackupCompletedEvent {
    private String backupId;
    private long sizeBytes;
}
```

Messaging Patterns

1. Point-to-Point (Queue)

Un messaggio consumato da **un solo** consumer.

```
Producer → [Queue] → Consumer A   ✓
                   ↛ Consumer B   ✗
```

Uso: task distribution, load balancing

2. Publish-Subscribe (Topic)

Un messaggio consumato da **tutti** i subscriber.

```
Producer → [Topic] → Consumer A   ✓
                   → Consumer B   ✓
                   → Consumer C   ✓
```

Uso: notifiche broadcast, event sourcing

Analisi del Pattern

Vantaggi ✅

- **Disaccoppiamento**: producer ignora consumer
- **Scalabilità**: aggiungere consumer senza modificare producer
- **Asincronicità**: performance migliori
- **Resilienza**: fallimento consumer non blocca producer
- **Estensibilità**: aggiungere nuove reazioni senza modificare codice esistente

Svantaggi ❌

- **Complessità**: debugging difficile (flusso non lineare)
- **Consistenza**: eventual consistency, non strong consistency
- **Testing**: difficile testare flussi end-to-end

- **Monitoring**: necessario tracciare eventi nel sistema
- **Error handling**: gestire fallimenti consumer

Rating

Criterio	Rating	Note
Agility	⭐ ⭐ ⭐ ⭐ ⭐	Modifiche non impattano producer
Deployment	⭐ ⭐ ⭐ ⭐	Consumer indipendenti
Testability	⭐ ⭐	Flussi complessi da testare
Performance	⭐ ⭐ ⭐ ⭐ ⭐	Asincrono
Scalability	⭐ ⭐ ⭐ ⭐ ⭐	Eccellente
Ease of Development	⭐ ⭐	Curva apprendimento alta

Quando Usare Event-Driven

✅ **Usa Event-Driven quando:**

- **Scalabilità** critica
- Processi **asincroni** dominanti
- Microservices architecture
- Real-time processing
- Integrazione sistemi eterogenei

❌ **Evita Event-Driven quando:**

- Applicazioni semplici CRUD
- Transazioni ACID necessarie
- Team non ha esperienza con messaging
- Debugging real-time critico

Microservices Architecture

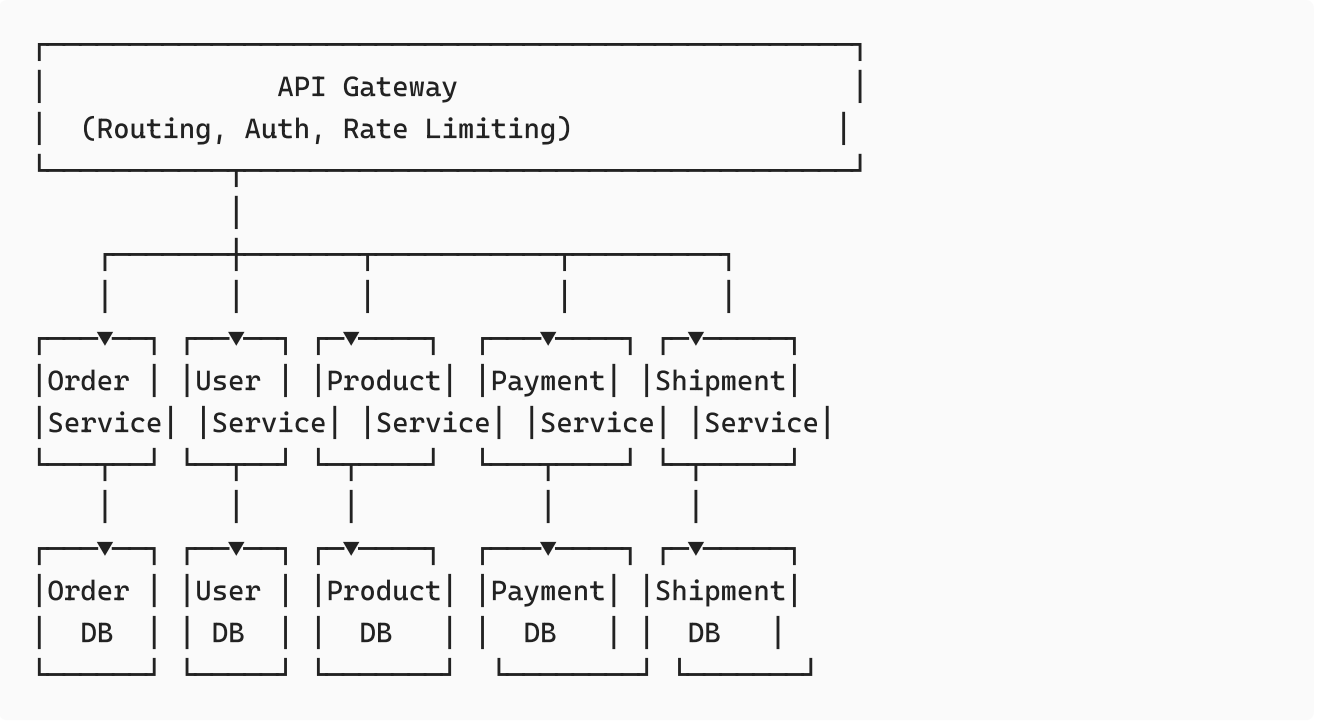
Definizione

Microservices è un'architettura che struttura il sistema come collezione di **servizi piccoli, autonomi e indipendenti**, ciascuno eseguibile in un processo separato e comunicante tramite protocolli leggeri (HTTP/REST, gRPC).

Principi Fondamentali

1. **Deployabilità indipendente**
2. **Ownership di team piccoli**
3. **Bounded context** (DDD)
4. **Decentralizzazione** (dati e governance)
5. **Fault isolation**

Struttura



Caratteristiche:

- Ogni servizio ha il **proprio database**
- Comunicazione via **API REST/gRPC**
- **Team ownership**: un team, un servizio
- **Deployment indipendente**

Componenti Architetturali

1. API Gateway

Responsabilità: punto di ingresso unico.

Funzionalità:

- Request routing
- Authentication/Authorization

- Rate limiting
- Request/response transformation
- Circuit breaking

Tecnologie: Kong, AWS API Gateway, Spring Cloud Gateway

Esempio (Spring Cloud Gateway):

```yaml
spring:
  cloud:
    gateway:
      routes:
        - id: order-service
          uri: lb://ORDER-SERVICE
          predicates:
            - Path=/api/orders/**
          filters:
            - StripPrefix=1

        - id: user-service
          uri: lb://USER-SERVICE
          predicates:
            - Path=/api/users/**
```

2. Service Registry (Service Discovery)

Responsabilità: registro dinamico dei servizi.

Funzionalità:

- Servizi si auto-registrano
- Client scoprono servizi disponibili
- Health checking

Tecnologie: Eureka, Consul, Zookeeper

Esempio (Eureka):

```java
@EnableEurekaServer
@SpringBootApplication
public class ServiceRegistryApplication {
    public static void main(String[] args) {
        SpringApplication.run(ServiceRegistryApplication.class, args);
    }
}

// Servizio che si registra
@EnableEurekaClient
```

```
@SpringBootApplication
public class OrderServiceApplication {
    // ...
}
```

3. Configuration Service

Responsabilità: configurazione centralizzata.

Tecnologie: Spring Cloud Config, Consul

Esempio:

```
# application.yml centralizzato
order-service:
  database:
    url: jdbc:postgresql://db:5432/orders
  feature-flags:
    new-checkout: true
```

4. Circuit Breaker

Responsabilità: prevenire cascading failures.

Tecnologie: Hystrix, Resilience4j

Esempio (Resilience4j):

```
@Service
public class OrderService {
    @CircuitBreaker(name = "paymentService", fallbackMethod =
"paymentFallback")
    public Payment processPayment(Order order) {
        return paymentClient.charge(order.getTotal());
    }

    private Payment paymentFallback(Order order, Exception ex) {
        // Fallback: memorizza ordine per retry successivo
        return Payment.pending();
    }
}
```

Service Communication

1. Synchronous (REST/gRPC)

Pro: semplice, immediato
Contro: accoppiamento runtime, blocking

Esempio (REST):

```
@RestController
public class OrderController {
    @Autowired
    private RestTemplate restTemplate;

    @PostMapping("/orders")
    public Order createOrder(@RequestBody OrderRequest req) {
        // Chiama User Service
        User user = restTemplate.getForObject(
            "http://USER-SERVICE/users/" + req.getUserId(),
            User.class
        );

        // Chiama Product Service
        Product product = restTemplate.getForObject(
            "http://PRODUCT-SERVICE/products/" + req.getProductId(),
            Product.class
        );

        // Crea ordine
        Order order = new Order(user, product);
        return orderRepository.save(order);
    }
}
```

2. Asynchronous (Message Broker)

Pro: disaccoppiamento, scalabilità
Contro: eventual consistency, complessità

Esempio (Kafka):

```
// Order Service: pubblica evento
@Service
public class OrderService {
    @Autowired
    private KafkaTemplate<String, OrderCreatedEvent> kafka;

    public Order createOrder(OrderRequest req) {
        Order order = orderRepository.save(new Order(req));

        // Pubblica evento
        OrderCreatedEvent event = new OrderCreatedEvent(order.getId());
```

```
        kafka.send("order-events", event);

        return order;
    }
}

// Shipment Service: consuma evento
@Service
public class ShipmentService {
    @KafkaListener(topics = "order-events")
    public void handleOrderCreated(OrderCreatedEvent event) {
        // Prepara spedizione
        Shipment shipment = new Shipment(event.getOrderId());
        shipmentRepository.save(shipment);
    }
}
```

Data Management

Database per Service

Principio: ogni microservice ha il **proprio database**.

Vantaggi:

- Indipendenza tecnologica (SQL, NoSQL)
- Nessun accoppiamento via database
- Scalabilità indipendente

Sfide:

- Query cross-service complesse
- Transazioni distribuite (SAGA pattern)
- Consistenza dati

SAGA Pattern

Problema: transazioni distribuite su più servizi.

Soluzione: sequenza di transazioni locali coordinate.

Esempio (Choreography):

```
1. Order Service: crea ordine → pubblica OrderCreated
2. Payment Service: processa pagamento → pubblica PaymentCompleted
3. Shipment Service: prepara spedizione → pubblica ShipmentReady
```

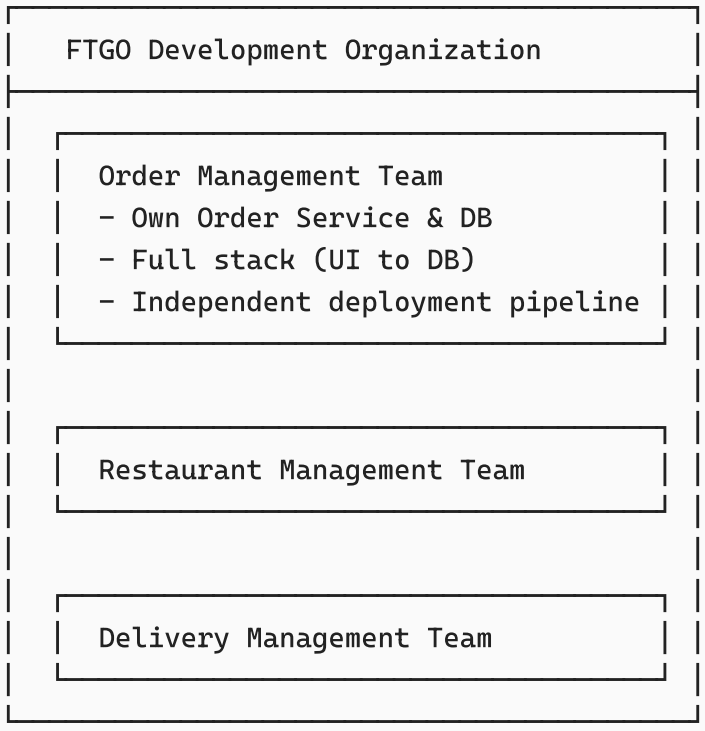
```
4. Notification Service: invia email → pubblica NotificationSent

In caso di errore: eventi di compensazione
```

Organizzazione Team (Conway's Law)

Legge di Conway:

> "Le organizzazioni che progettano sistemi sono vincolate a produrre design che sono copie delle loro strutture di comunicazione."

Approccio Microservices:



Caratteristiche:

- Team **piccoli** (2-10 persone)
- **Autonomous**: ownership end-to-end
- **Cross-functional**: backend, frontend, DB, DevOps

Analisi del Pattern

Vantaggi ✅

- **Scalabilità indipendente**: scala solo i servizi sotto carico

- **Deployment continuo**: deploy un servizio senza impattare altri
- **Fault isolation**: errore in un servizio non crasha il sistema
- **Technology heterogeneity**: ogni servizio può usare stack diverso
- **Team autonomy**: sviluppo parallelo e indipendente
- **Organizational alignment**: struttura team = struttura sistema

Svantaggi ❌

- **Complessità operativa**: monitoring, logging, tracing distribuiti
- **Network latency**: comunicazione inter-service via rete
- **Data consistency**: CAP theorem, eventual consistency
- **Testing**: integration testing complesso
- **Deployment**: orchestrazione container (Kubernetes)
- **Learning curve**: team deve conoscere pattern distribuiti

Rating

Criterio	Rating	Note
Agility	⭐ ⭐ ⭐ ⭐ ⭐	Deploy indipendenti
Deployment	⭐ ⭐ ⭐ ⭐	CI/CD per servizio
Testability	⭐ ⭐	Testing distribuito complesso
Performance	⭐ ⭐ ⭐	Network overhead
Scalability	⭐ ⭐ ⭐ ⭐ ⭐	Granularità fine
Ease of Development	⭐	Richiede expertise distribuiti

Quando Usare Microservices

✅ **Usa Microservices quando:**

- Sistema **grande e complesso**
- Team **distribuiti** geograficamente
- **Scalabilità** differenziata necessaria
- Deployment **continuo** critico
- Bounded context chiari (DDD)
- Budget per infrastruttura (Kubernetes, monitoring)

❌ **Evita Microservices quando:**

- **Applicazione semplice** (< 10 entità)

- Team **piccolo** (< 5 persone)
- Budget limitato
- Infrastruttura cloud non disponibile
- Dominio poco chiaro

Confronto tra Pattern

Tabella Comparativa Completa

Pattern	Complessità	Scalabilità	Deploy	Testabilità	Ca
Layered	⭐ Bassa	⭐ Verticale	⭐ Monolite	⭐⭐⭐⭐ Alta	App trad CR
Microkernel	⭐⭐ Media	⭐⭐ Limitata	⭐⭐⭐ Plugin	⭐⭐⭐⭐ Alta	IDE per
Event-Driven	⭐⭐⭐ Alta	⭐⭐⭐⭐⭐ Eccellente	⭐⭐⭐⭐ Componenti	⭐⭐ Bassa	Rea stre
Microservices	⭐⭐⭐⭐⭐ Molto Alta	⭐⭐⭐⭐⭐ Eccellente	⭐⭐⭐⭐⭐ Servizi	⭐⭐ Bassa	Sis dist

---

Scelta del Pattern

Decision Tree

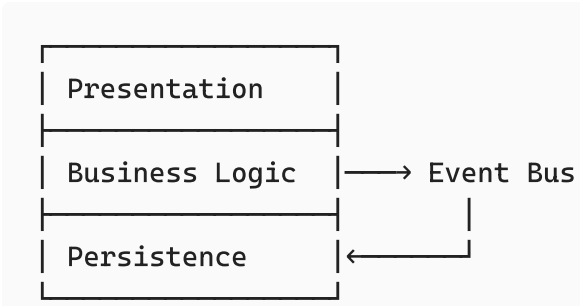
```
È un'applicazione semplice CRUD?
    |
    ├──> SÌ → Layered Architecture
    |
    └──> NO → Servono funzionalità opzionali/plugin?
              |
              ├──> SÌ → Microkernel Architecture
              |
              └──> NO → È critica la scalabilità differenziata?
                        |
                        ├──> SÌ → Serve real-time/streaming?
                        |         |
                        |         ├──> SÌ → Event-Driven + Microservices
                        |         |
                        |         └──> NO → Microservices Architecture
```

```
                        │
                        └──→ NO → Event-Driven Architecture
```

Hybrid Approaches

È possibile combinare pattern:

Layered + Event-Driven



Microservices + Layered (interno a ogni servizio)

```
Order Service:
   ┌──────────────┐
   │ REST API     │
   ├──────────────┤
   │ Service      │
   ├──────────────┤
   │ Repository   │
   └──────────────┘
```

Best Practices

1. Start Simple

Principio: inizia con Layered, evolvi verso Microservices se necessario.

Motivo: over-engineering prematuro è costoso.

2. Bounded Context (DDD)

Principio: identifica chiaramente i confini dei servizi/moduli.

Esempio:

- Order Management
- User Management
- Product Catalog

- Billing

3. Decouple via Interfaces

Principio: usa interfacce e dependency injection.

```
public interface PaymentGateway {
    Payment charge(Order order);
}

// Implementazioni intercambiabili
public class StripePaymentGateway implements PaymentGateway { }
public class PayPalPaymentGateway implements PaymentGateway { }
```

4. Monitoring & Observability

Strumenti essenziali:

- **Logging**: ELK Stack (Elasticsearch, Logstash, Kibana)
- **Metrics**: Prometheus + Grafana
- **Tracing**: Jaeger, Zipkin
- **Alerting**: PagerDuty, Opsgenie

5. Automation

CI/CD Pipeline:

```
Code Push → Build → Test → Deploy → Monitor
```

Tools: Jenkins, GitLab CI, GitHub Actions, ArgoCD

Conclusione

La scelta del **software architecture pattern** è una decisione critica che impatta:

- **Velocità di sviluppo**
- **Manutenibilità a lungo termine**
- **Scalabilità**
- **Costi operativi**

Principi guida:

1. **Inizia semplice**: Layered è un ottimo punto di partenza
2. **Evolvi gradualmente**: non saltare direttamente a Microservices

3. **Valuta trade-off**: ogni pattern ha costi e benefici
4. **Allinea con organizzazione**: struttura del team influenza architettura
5. **Automatizza**: CI/CD e monitoring sono fondamentali

Ricorda: non esiste il "pattern perfetto". La scelta dipende da:

- Dimensione team
- Complessità dominio
- Requisiti di scalabilità
- Budget
- Competenze tecniche

Riferimenti

- **Software Architecture Patterns**, Mark Richards, O'Reilly, 2015
- **Microservices Patterns**, Chris Richardson, Manning, 2018
- **Building Microservices**, Sam Newman, O'Reilly, 2015
- **Domain-Driven Design**, Eric Evans, Addison-Wesley, 2003
- **Enterprise Integration Patterns**, Gregor Hohpe, Addison-Wesley, 2003