Microservice Design Patterns

1. Gateway Pattern

→ It acts as a centrally managed gatekeeper for a number of services.
→ It is a reverse-proxy server.

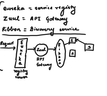
→ Instead of the client directly calling each individual service, it sends all requests to the API Gateway.
→ It acts as the single entry point for all client requests to the microservices architecture.
→ It acts as a reverse proxy that routes requests from clients to the appropriate microservice.

→ Real-world use cases:-
i) Aggregation and composition
ii) Authentication and authorization
iii) Request routing and load balancing.
iv) Protocol translation

→ Pros:-
i) Improved security and governance
ii) Scalability and performance optimization
iii) Simplified client interaction.

→ Cons:-
i) Single point of failure
ii) Increased complexity
iii) Potential performance impact

→ Practical Examples:-
i) Netflix API Gateway (Zuul) ⟶ Eureka → service registry, Zuul → API Gateway, Ribbon → Balancing service.



ii) Spring cloud Gateway
iii) Amazon API Gateway

2. Proxy Pattern

→ It involves introducing a proxy component between a client and a service.
→ The proxy acts as an intermediary.
→ It provides additional functionality such as:-
i) caching
ii) load balancing
iii) security
iv) protocol translation
→ Transparently forwards requests to the underlying service.

↳ Pros:
i) Separation of concerns
ii) Flexibility
iii) Improved performance

↳ Cons:
i) Increased complexity
ii) Additional latency
iii) Potential single point of failure

↳ Practical Examples:
i) Content Delivery Networks (CDNs)
ii) API Gateways
iii) Service meshes

3) Aggregation Pattern

→ Involves combining data from multiple services or sources into a single unified response.
→ The API Gateway acts as a central service that orchestrates requests to other services, gathers the responses, and aggregates them to form a cohesive result.
→ Example:-

```
public OrderDetailsDTO getOrderDetails(String orderId) {
    // Make requests to other services
    Product product = productService.getProduct(orderId);
    Shipping shipping = shippingService.getShipping(orderId);

    // Aggregate the data into a single response
    OrderDetailsDTO orderDetails = new OrderDetailsDTO();
    orderDetails.setProduct(product);
    orderDetails.setShipping(shipping);

    return orderDetails;
}
```

→ Real-world use cases:-
i) Composite APIs
ii) Client-specific responses
iii) Data transformation
iv) Reduced chattiness

→ Pros:-
i) Simplified client interaction
ii) Flexibility
iii) Reduced network overhead

→ Cons:-
i) Increased complexity
ii) Potential performance impact

→ Practical Examples:-
i) API Gateway
ii) Aggregated reporting
iii) Backend optimization

4) API Gateway

→ Provides using a central entry point for client requests to the microservices architecture.
→ It acts as a reverse proxy that routes requests from clients, performs routing and composition, and delegates the requests to the appropriate microservice.
→ Real-world use cases:-
i) Aggregation and composition
ii) Authentication and authorization
iii) Request routing and load balancing.
iv) Protocol translation

→ Pros:-
i) Improved security and governance
ii) Scalability and performance optimization
iii) Simplified client interaction.

→ Cons:-
i) Single point of failure
ii) Increased complexity
iii) Potential performance impact

→ Practical Examples:-
i) Netflix API Gateway (Zuul) ⟶ Eureka → service registry, Zuul → API Gateway, Ribbon → Balancing service.

ii) Spring cloud Gateway
iii) Amazon API Gateway

Chained Pattern

→ Useful in scenarios where multiple microservices need to collaborate and share information to fulfill a particular request or business process.
→ Example:- In an e-commerce application, when a new order is created, it might be necessary to automatically generate an invoice order for that order.

Codebase:

```
@MicroService
UserInventoryImpl → UserInventoryBlm
@OrderCreate ⟶ chained pattern
OrderInventoryImpl → OrderCreateBlm

UserInventoryBlm, OrderCreateBlm
        \          /
   InventoryController, InventoryBlm
```

Saga Pattern

→ Used in microservice architecture to manage long-running transactions and ensure data consistency across multiple services.
→ It helps address the challenges of maintaining data integrity and handling failures in a distributed system.
→ Saga is a sequence of local transactions, where each local transaction updates the data within a single service.
Each local transaction is associated with a specific step or operation in the overall business process.
The saga orchestrator coordinates the execution of these local transactions and ensures that either all of them are successfully completed or compensating actions are taken to rollback the changes.
→ For example, in an e-commerce platform, it involves multiple services such as inventory management, payment processing, and shipping. By using the Saga pattern, a saga can be created to handle the order processing, where each local transaction represents actions like:
i) reserving the item in the inventory.
ii) charging the user's payment method.
iii) initiating the shipping process.
If any of these steps fail, the saga can initiate compensating actions to reverse the executed transactions and ensure data consistency.
↳ eventual consistency

```
WithdrawMoneySaga
    |
    |-- withdrawMoneyStep
    |       @Transactional
    |       requestSaga() {
    |           withdrawMoneyStep.executeWithdrawal();
    |           withdrawMoneyTransferStep.executeTransfer();
    |       } catch { }
    |   }
    |
    |-- withdrawMoneyTransferStep
    |   }
    |
    |   public class WithdrawMoneyStep {
    |       executeWithdrawal();
    |       compensateWithdrawal();
    |   }
```

Circuit Breaker

→ Used to handle and manage failures in distributed systems.
→ It helps to prevent cascading failures and provides fault tolerance by detecting and isolating failing services or components.
→ Acts as a safety mechanism between the client and the service.
→ It monitors the availability and responsiveness of the service.
→ When the service is functioning normally, the circuit breaker allows requests to pass through.
→ If the service starts to fail or respond slowly, the circuit breaker opens and stops further requests from reaching the service.
→ The circuit breaker can periodically check the health of the service and attempt to close the circuit when the service recovers.
→ Real-world examples
i) Netflix Hystrix
ii) Spring Cloud Circuit Breaker

Sidecar pattern

→ It is used in microservice architecture to provide additional functionality and capabilities to a service without modifying its core codebase.
→ It involves deploying a separate sidecar service alongside the main service to handle specific tasks or functions.
→ The sidecar service runs in the same context as the main service, typically provides additional capabilities such as logging, monitoring, security, service discovery, and communication with other services.
→ The sidecar service communicates with the main service through local inter-process communication mechanisms, such as shared memory, pipes or localhost networking.
→ Real-world examples:-
i) Netflix Zuul
ii) Istio