

<https://learn.microsoft.com/en-us/azure/architecture/patterns>

<https://microservices.io/index.html>

<https://www.udemy.com/course/design-microservices-architecture-with-patterns-principles/>

Contents

[Decomposition Patterns: 2](#_Toc143799881)

[Strangler Fig Pattern 2](#_Toc143799882)

[Database per service 6](#_Toc143799883)

[Bulkhead 8](#_Toc143799884)

[Sidecar 9](#_Toc143799885)

[Integration Patterns 10](#_Toc143799886)

[BFF (Backend For Frontend) 10](#_Toc143799887)

[API Gateway Pattern 11](#_Toc143799888)

[Database Patterns 12](#_Toc143799889)

[API Composition 12](#_Toc143799890)

[Query Service pattern 15](#_Toc143799891)

[API Composition pattern and the Query Service pattern 17](#_Toc143799892)

[CQRS (Command Query Responsibility Segregation) 18](#_Toc143799893)

[Saga 19](#_Toc143799894)

[Event-Driven/ Publish-Subscribe 24](#_Toc143799895)

[Observability Patterns 25](#_Toc143799896)

[Cross-Cutting Concern Patterns 25](#_Toc143799897)

[Service Discovery 25](#_Toc143799898)

[Circuit Breaker 25](#_Toc143799899)

[Some Important internal service design 25](#_Toc143799900)

[Retry 25](#_Toc143799901)

[Trace 25](#_Toc143799902)

[gRPC 25](#_Toc143799903)

[Applying Domain-Driven Design principles to a Nest.js project 25](#_Toc143799904)

[Kafka 26](#_Toc143799905)

# Decomposition Patterns:

These patterns help you divide a monolithic application into smaller, manageable microservices.

## Strangler Fig Pattern

a solution that allows you to gradually replace parts of the legacy system without causing business disruption.

The Problem: Legacy Systems

these systems often resemble an intricate web with various dependencies. Alter one part, and it could impact the entire network.

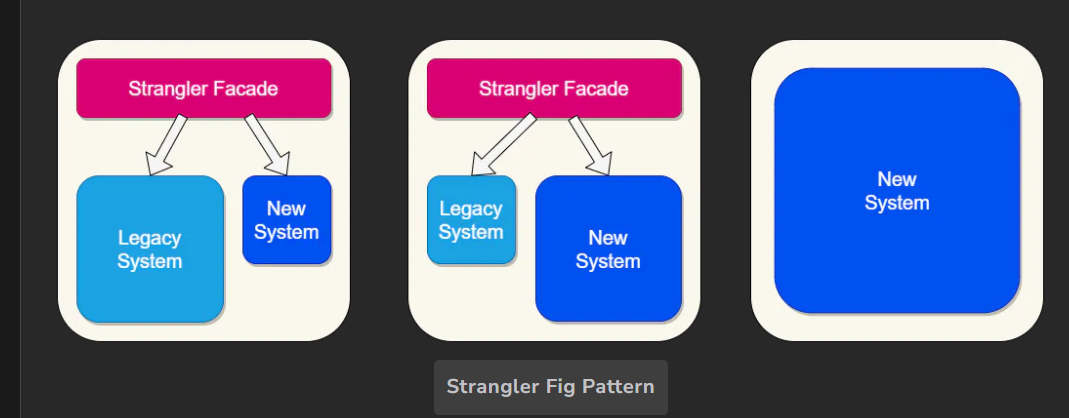
A Solution:

It's based on the strangler fig tree's strategy, which grows around a host tree, gradually overshadowing and replacing it. Similarly, the Strangler Pattern suggests slowly creating a new system around the edges of the old one, progressively replacing its functionality.

Implementing the Strangler Pattern

The implementation of the Strangler Pattern begins by identifying a single functionality of the legacy system that can be rebuilt and redirected to the new system.

Once a functionality is identified, it's duplicated in the new system, and traffic is rerouted to the new implementation using a Facade interface. This interface is the gatekeeper, directing incoming requests either to the new system or to the old one. Over time, more and more functionality is migrated to the new system, and traffic to the old system decreases until it's finally phased out.



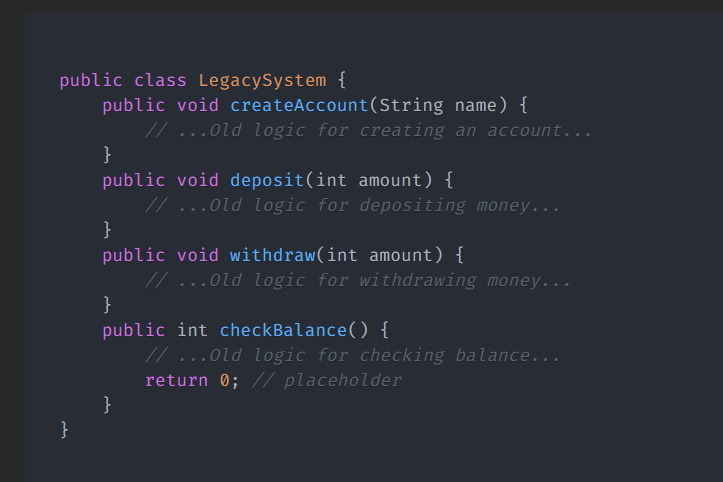
The Architecture of the Strangler Pattern

Facade Interface: The Gatekeeper

The central component in the Strangler Pattern's architecture is the Facade Interface. This is the gatekeeper that decides whether a request should be handled by the new system or the legacy system.

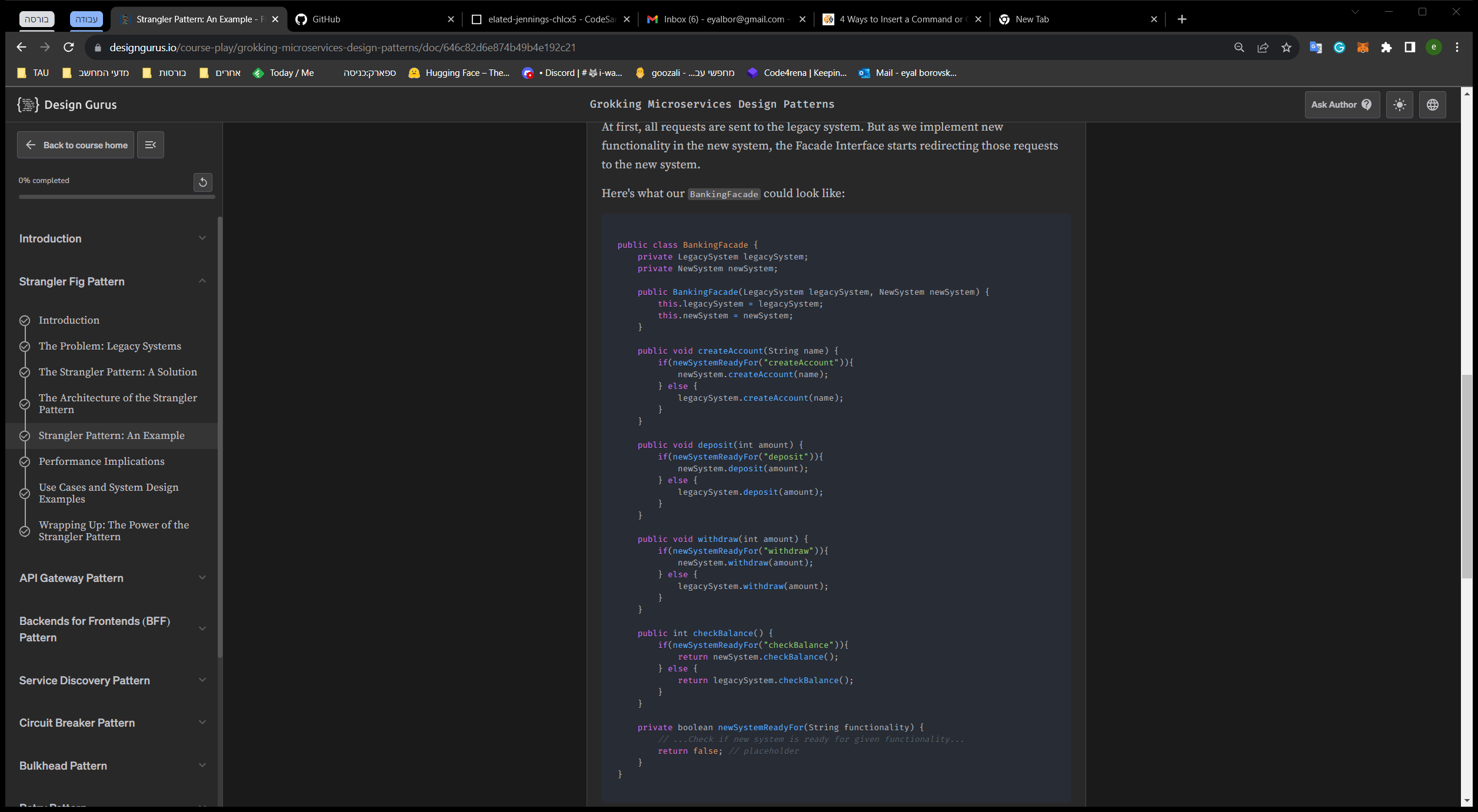
When a request arrives, the Facade Interface checks if the requested functionality has been migrated to the new system. If it has, the request is directed to the new system. If not, the request continues to be handled by the legacy system.

Consider a banking system as an example



A screenshot of a computer program

Description automatically generated



Not a Silver Bullet:

For instance, if you're dealing with a small, non-critical legacy system, it might be faster and more cost-effective to simply rewrite it instead of applying the Strangler Pattern.

Another consideration is the potential performance overhead. Just as a busy highway junction can cause traffic congestion

keeping the data in sync between the legacy and new systems is essential for a smooth transition.

Moreover, this coexistence means double the maintenance work.

Finally, let's not forget the human factor. The transition to a new system often requires training for staff members

System Design Example: Microservices Architecture

One of the most common applications of the Strangler Pattern is in transitioning from a monolithic architecture to a microservices architecture.

Applying the Strangler Pattern in this context involves creating a new microservice for a specific functionality and then using the facade to route the relevant requests to the new microservice. Gradually, the monolithic application will be 'strangled,' leaving only a set of loosely coupled microservices.

## Database per service

**Problem:** What’s the database architecture in a microservices application?

services must be loosely coupled so that they can be developed, deployed and scaled independently

Some business transactions must enforce invariants that span multiple services. For example, the Place Order use case must verify that a new Order will not exceed the customer’s credit limit. Other business transactions, must update data owned by multiple services.

Some business transactions need to query data that is owned by multiple services. For example, the View Available Credit use must query the Customer to find the creditLimit and Orders to calculate the total amount of the open orders.

Some queries must join data that is owned by multiple services. For example, finding customers in a particular region and their recent orders requires a join between customers and orders.

Databases must sometimes be replicated and sharded in order to scale. See the Scale Cube.

Different services have different data storage requirements. For some services, a relational database is the best choice. Other services might need a NoSQL database such as MongoDB, which is good at storing complex, unstructured data, or Neo4J, which is designed to efficiently store and query graph data.

**Solution**:

Keep each microservice’s persistent data private to that service and accessible only via its API. A service’s transactions only involve its database.

The service’s database is effectively part of the implementation of that service. It cannot be accessed directly by other services.

There are a few different ways to keep a service’s persistent data private. You do not need to provision a database server for each service. For example, if you are using a relational database then the options are:

Private-tables-per-service – each service owns a set of tables that must only be accessed by that service

Schema-per-service – each service has a database schema that’s private to that service

Database-server-per-service – each service has it’s own database server.

Private-tables-per-service and schema-per-service have the lowest overhead. Using a schema per service is appealing since it makes ownership clearer. Some high throughput services might need their own database server.

It is a good idea to create barriers that enforce this modularity. You could, for example, assign a different database user id to each service and use a database access control mechanism such as grants. Without some kind of barrier to enforce encapsulation, developers will always be tempted to bypass a service’s API and access it’s data directly.

Benefits:

* Helps ensure that the services are loosely coupled. Changes to one service’s database does not impact any other services.
* Each service can use the type of database that is best suited to its needs. For example, a service that does text searches could use ElasticSearch. A service that manipulates a social graph could use Neo4j.

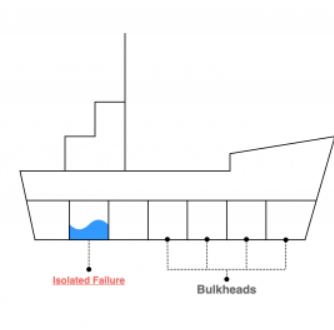
Drawbacks:

* implementing business transactions that span multiple services is not straightforward. Distributed transactions are best avoided because of the CAP theorem. Moreover, many modern (NoSQL) databases don’t support them.
* Implementing queries that join data that is now in multiple databases is challenging.
* Complexity of managing multiple SQL and NoSQL databases

There are various patterns/solutions for implementing transactions and queries that span services:

* Implementing transactions that span services - use the [Saga pattern](https://microservices.io/patterns/data/saga.html).
* Implementing queries that span services:
  + [API Composition](https://microservices.io/patterns/data/api-composition.html) - the application performs the join rather than the database. For example, a service (or the API gateway) could retrieve a customer and their orders by first retrieving the customer from the customer service and then querying the order service to return the customer’s most recent orders.
  + [Command Query Responsibility Segregation (CQRS)](https://microservices.io/patterns/data/cqrs.html) - maintain one or more materialized views that contain data from multiple services. The views are kept by services that subscribe to events that each services publishes when it updates its data. For example, the online store could implement a query that finds customers in a particular region and their recent orders by maintaining a view that joins customers and orders. The view is updated by a service that subscribes to customer and order events.

## Bulkhead



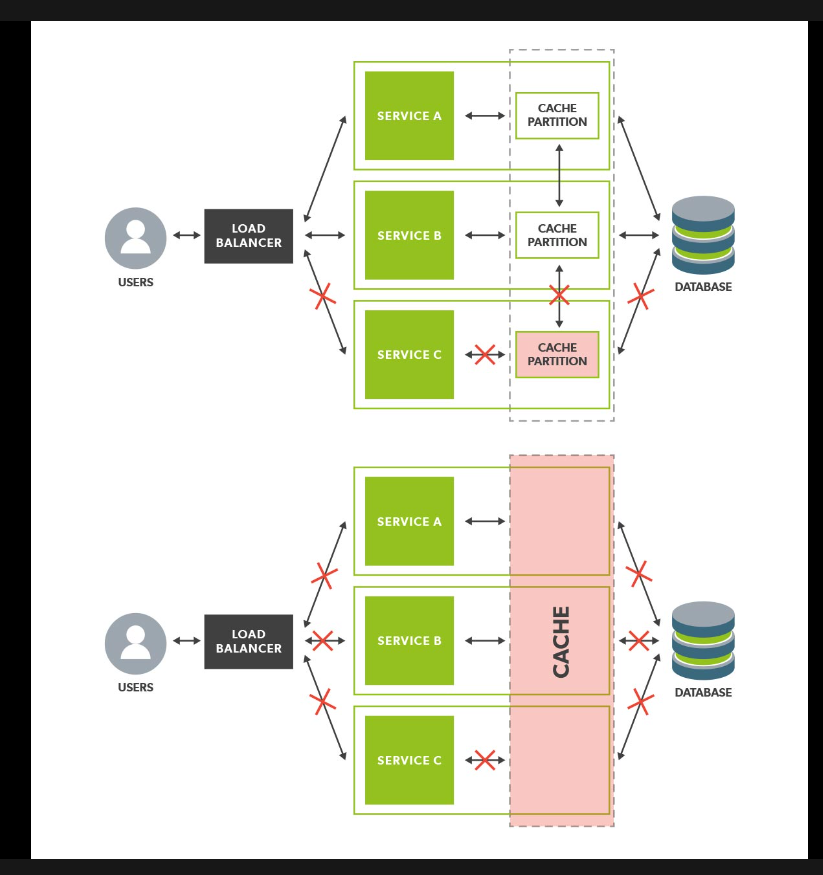
Problem: A failure in one microservice can impact other services that share resources.

Microservices are distributed in nature. It has more components and moving parts. In the distributed architecture, dealing with any unexpected failure is one of the biggest challenges to solve. It could be a hardware failure, network failure, etc. The ability of the system to recover from the failure and remain functional makes the system more resilient. It also avoids any cascading failures.

Solution: In a bulkhead architecture, elements of an application are isolated into pools so that if one fails, the others will continue to function.

Isolate services by allocating separate resources (e.g., thread pools) to prevent one service's failure from affecting others.

Code Implementation: Design services to have dedicated resources, ensuring failures are contained.



When Not to Use: In scenarios with minimal resource sharing or where the potential failure impact is low.

Life Example: Amazon isolates its services using the bulkhead pattern to prevent resource exhaustion and improve overall system resilience.

## Sidecar

Problem: Adding cross-cutting concerns like logging, monitoring, or security can bloat the codebase of microservices.

Solution: Use a sidecar pattern to attach a secondary container (sidecar) to the main microservice container, handling these concerns separately.

Code Implementation: Deploy a separate container alongside the microservice container to handle specific concerns.

When Not to Use: For microservices with minimal cross-cutting concerns that can be efficiently managed within the main container.

Life Example: Istio uses a sidecar approach to manage service mesh capabilities like traffic management and security.

# Integration Patterns

These patterns focus on communication and interaction between microservices.

## BFF (Backend For Frontend)

A diagram of a company

Description automatically generated

Problem: Different client types (web, mobile, etc.) require different data and interactions from the same microservices.

Solution: Introduce a BFF that acts as an intermediary between clients and microservices, tailoring responses to each client's specific needs. BFF acts as a perfect abstraction for underlying microservices. For an API, it can connect to necessary microservices, gather the responses, and respond. This ensures that we are not fetching data that we don't need.

Code Implementation: Develop separate BFFs for different client types, aggregating and transforming data as required.

When Not to Use: For small-scale applications with minimal client diversity, introducing BFFs might be unnecessary complexity. a large chunk of code would be duplicated. there is a slight increase in latency.

Life Example: Spotify uses BFFs to handle different aspects of their web and mobile applications, optimizing the user experience.

## API Gateway Pattern

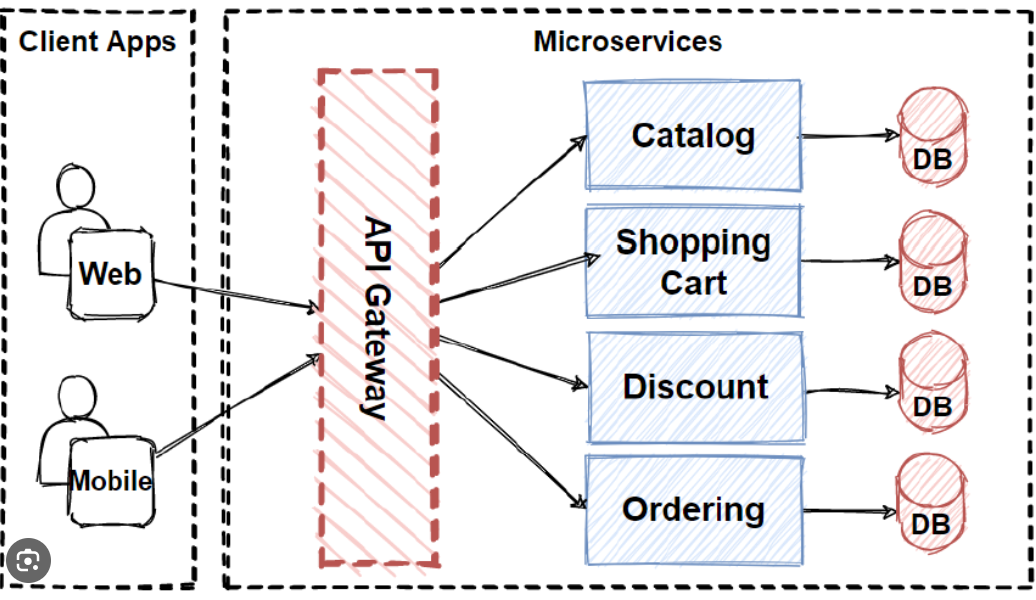
Problem: Handling multiple client requests and routing them to appropriate microservices can lead to complexity.

Solution: Use a single entry point (API Gateway) to handle client requests, route them to relevant services, and potentially handle authentication, rate limiting, authorization, and caching

The API gateway must use either the Client-side Discovery pattern or Server-side Discovery pattern to route requests to available service instances.

The API Gateway may authenticate the user and pass an Access Token containing information about the user to the services

An API Gateway will use a Circuit Breaker to invoke services



Code Implementation:

import cors from 'cors';

import express from 'express';

import proxy from 'express-http-proxy';

*const* host = process.env.GATEWAY\_SERVICE\_HOST;

*const* port = Number(process.env.GATEWAY\_SERVICE\_PORT);

*const* app = express();

app.use(cors());

app.use(express.json());

*const* shopPath = `http://${process.env.SHOP\_SERVICE\_HOST}:${process.env.SHOP\_SERVICE\_PORT}`;

*const* userPath = `http://${process.env.USER\_SERVICE\_HOST}:${process.env.USER\_SERVICE\_PORT}`;

*const* productPath = `http://${process.env.PRODUCT\_SERVICE\_HOST}:${process.env.PRODUCT\_SERVICE\_PORT}`;

*const* paymentPath = `http://${process.env.PAYMENT\_SERVICE\_HOST}:${process.env.PAYMENT\_SERVICE\_PORT}`;

app.use('/shop', proxy(shopPath));

app.use('/user', proxy(userPath));

app.use('/product', proxy(productPath));

app.use('/payment', proxy(paymentPath));

app.listen(port, host, () *=>* {

    console.log(`Service is ready and running on path: http://${host}:${port}`);

});

When Not to Use: For simple applications with only a few microservices, using an API Gateway might introduce unnecessary overhead.

Life Example: [Netflix's API Gateway](https://netflixtechblog.com/embracing-the-differences-inside-the-netflix-api-redesign-15fd8b3dc49d) manages requests to various microservices within its ecosystem. Use tools like Netflix Zuul or Spring Cloud Gateway for routing and filtering requests to microservices.

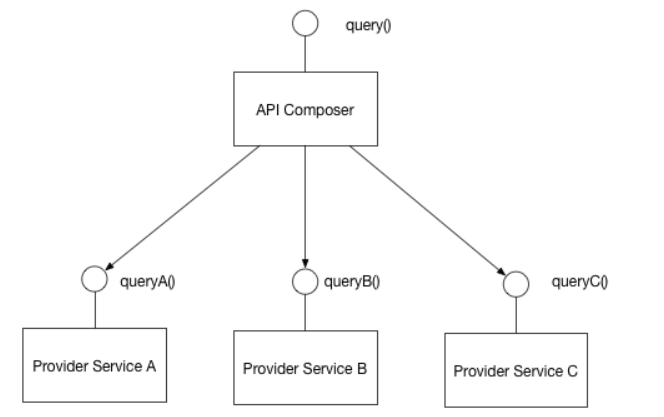
# Database Patterns

These patterns deal with the data storage and access strategies in microservices.

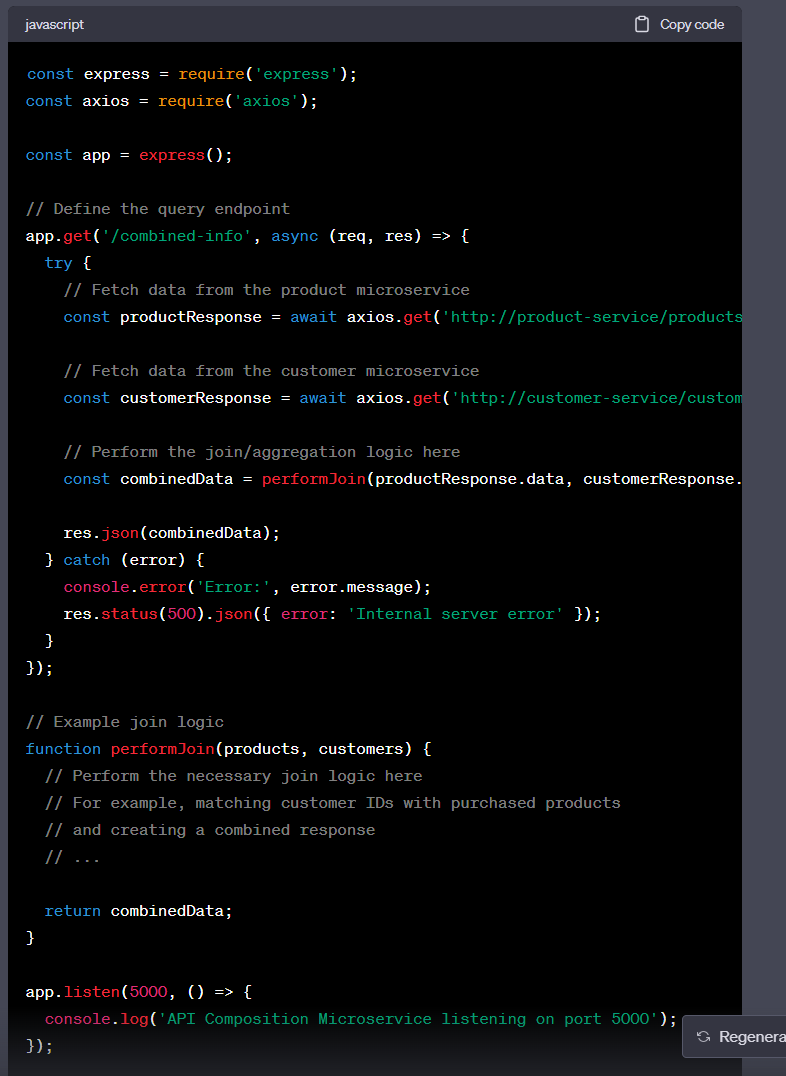
## API Composition

**Problem**: How to implement queries in a microservice architecture?

**Solution**: implement a query by defining an API Composer, which invoking the services that own the data and performs an in-memory join of the results.



**Code Implementation:**

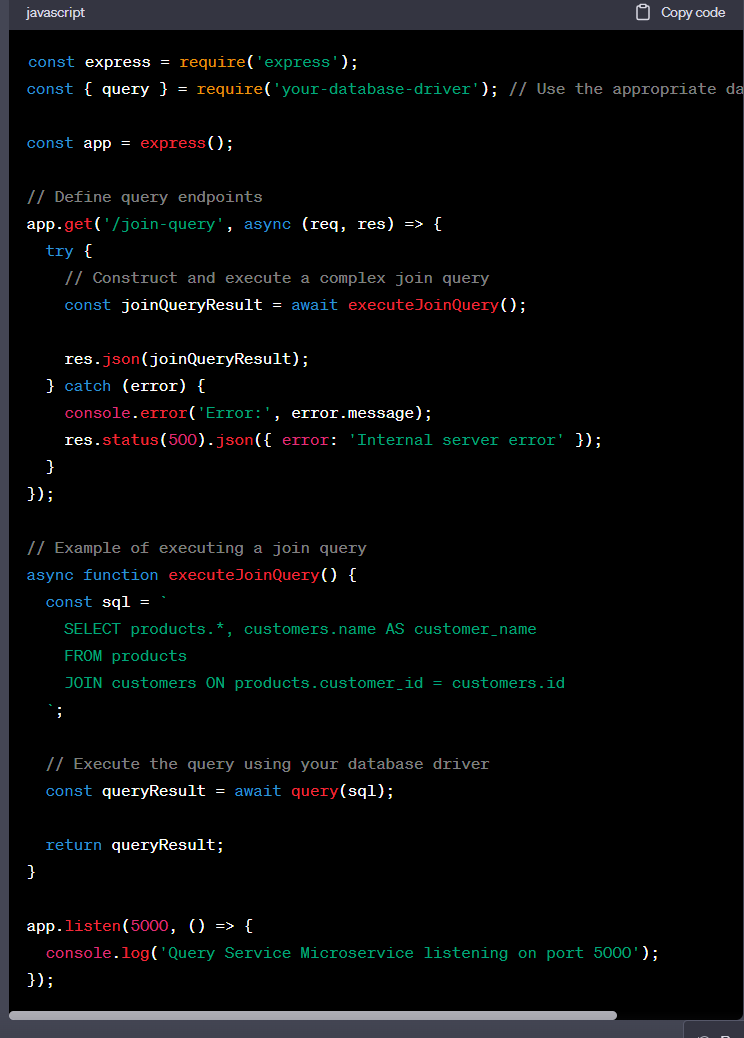
****

**When Not to Use:** Some queries would result in inefficient, in-memory joins of large datasets.

**Life Example:** API Gateway

## Query Service pattern

create a dedicated microservice specifically for executing database queries that involve joining data, you're essentially considering a Query Service pattern. This pattern involves setting up a microservice responsible for handling complex database queries, including joins, aggregations, and other data manipulation operations.



This pattern can be beneficial for offloading complex database queries from other microservices, ensuring optimized query execution and avoiding putting unnecessary load on the microservices responsible for other business logic.

## API Composition pattern and the Query Service pattern

Deciding between the API Composition pattern and the Query Service pattern depends on your application's requirements, architecture, and trade-offs. Both patterns have their strengths and weaknesses, so it's important to consider your specific context. Here's a comparison to help you make an informed decision:

API Composition Pattern:

Pros:

Modularity: API composition keeps microservices focused on their core responsibilities. Each microservice provides well-defined APIs, reducing complexity and coupling.

Flexibility: Clients can request specific data combinations and transformations, providing a flexible way to tailor responses.

Contextual Data: Microservices have access to their domain data, making it easier to contextualize data for composition.

Cons:

Additional Network Calls: The pattern can involve multiple network calls, potentially introducing latency and increasing the likelihood of network failures.

Performance Considerations: Complex compositions might lead to performance challenges, especially if data retrieval and processing are distributed across multiple microservices.

Over-fetching/Under-fetching: Depending on client needs, you might encounter situations where you're fetching more or less data than required.

Query Service Pattern:

Pros:

Performance Optimization: Query services can be optimized for specific query types, including joins, aggregations, and complex operations.

Single Network Call: A query service consolidates data retrieval into a single network call, reducing potential network overhead.

Centralized Logic: Complex query logic is centralized, making it easier to manage and optimize.

Consistency: A query service ensures consistent query execution, avoiding duplication of logic across microservices.

Cons:

Microservice Overhead: Adding another microservice introduces deployment, management, and operational complexities.

Latency: There's a potential latency introduced by querying and aggregating data from various sources.

Duplication of Data: In some cases, a query service might require duplicated or replicated data to optimize queries.

Factors to Consider:

Query Complexity: If your queries involve complex joins, aggregations, or transformations, a Query Service might be a better fit.

Performance Requirements: Consider the performance requirements of your application. If optimizing query performance is critical, a Query Service might offer advantages.

Microservice Architecture: Evaluate your existing microservice architecture. If your microservices are already well-structured and optimized, the API Composition pattern might fit well.

Network Overhead: Consider the potential network overhead introduced by the API Composition pattern. Multiple network calls could impact latency and reliability.

Scalability: Consider the scalability needs of your application. Query services can be optimized for specific workloads, making them more scalable for read-heavy scenarios.

Developer Experience: Consider the developer experience and ease of maintenance. A well-organized Query Service might make complex querying more manageable.

Application Evolution: Think about the potential growth and complexity of your application. A Query Service might better handle evolving and expanding query needs.

In the end, there isn't a universally "better" approach—it depends on your specific requirements. You might even find a hybrid approach where certain queries are handled via API composition while others are managed by a Query Service. Striking the right balance between performance, modularity, and complexity is key to making the right choice for your application.

## CQRS (Command Query Responsibility Segregation)

**Problem**: is a design pattern that addresses the challenges of managing read and write operations in complex applications. How to implement a query that retrieves data from multiple services in a microservice architecture?

In a traditional monolithic application, a single data model is often used for both read and write operations. This can lead to challenges in performance optimization, scalability, and maintainability. CQRS aims to address these challenges by introducing a clear separation between the data models and operations.

**Solution:**

Separation of concerns. The model separates the read and write operations into separate models.

Scalability. The read and write operations can be scaled independently.

Flexibility. The model allows for the use of different data stores for read and write operations.

Performance. The model allows for the use of different data stores optimized for read and write operations

Instead of simple crud queries your queries become a task: like updateObject and send email

**Code Implementation:** Create separate read and write models with appropriate data structures and storage mechanisms.

Commands

Commands are used to change the application state. They should be task-based, rather than data centric. When a command is dispatched, it is handled by a corresponding Command Handler. The handler is responsible for updating the application state

Queries are used to retrieve data from the application state. They should be data centric, rather than task-based. When a query is dispatched, it is handled by a corresponding **Query Handler**. The handler is responsible for retrieving the data.

Events are used to notify other parts of the application about changes in the application state. They are dispatched by **models** or directly using the EventBus

Commands vs events

https://medium.com/ingeniouslysimple/command-vs-event-in-domain-driven-design-be6c45be52a9

**When Not to Use:** For simple applications with limited data complexity and low demand for scalability.

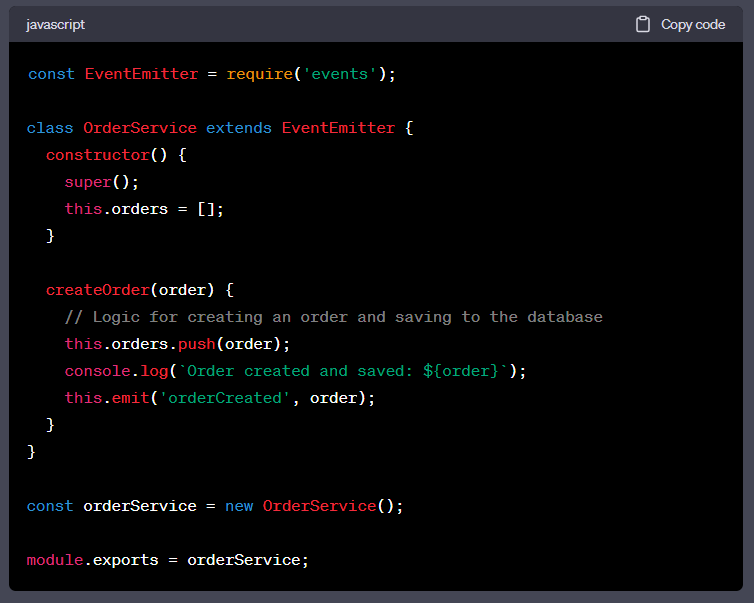
**Life Example:** Airbnb uses CQRS to manage the high volume of read operations while maintaining write operations for booking updates.

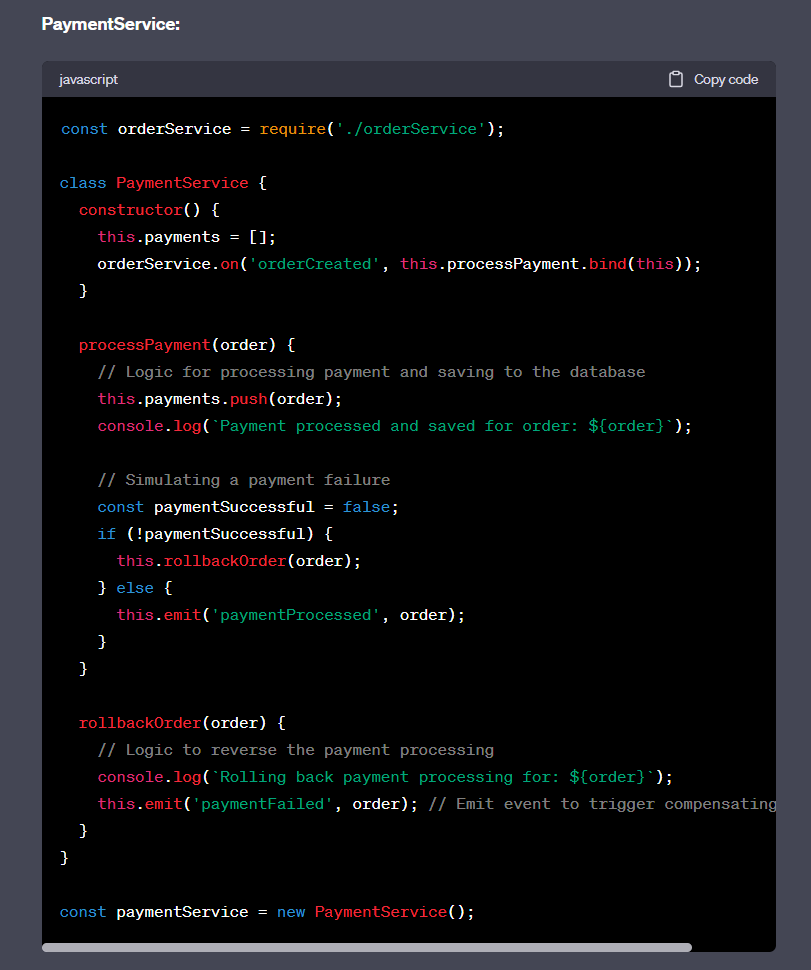
## Saga

Problem: Transactions must be atomic, consistent, isolated, and durable (ACID). Transactions within a single service are ACID, but cross-service data consistency requires a cross-service transaction management strategy. Solution: Implement a saga pattern to manage a series of local transactions that together form a larger business process.

Code Implementation: Design and implement a series of compensating transactions to handle partial rollbacks in case of failures.

**choreography-based saga pattern**. In a choreography-based saga, each service knows its own responsibilities and communicates directly with other services to coordinate the overall process. When a service completes its part of the process or encounters a failure, it triggers events that inform other services to take appropriate actions.





A screen shot of a computer program

Description automatically generated

In a real-world scenario, when a user initiates an order creation process, there are a few common ways to provide feedback about the success or failure of the entire saga:

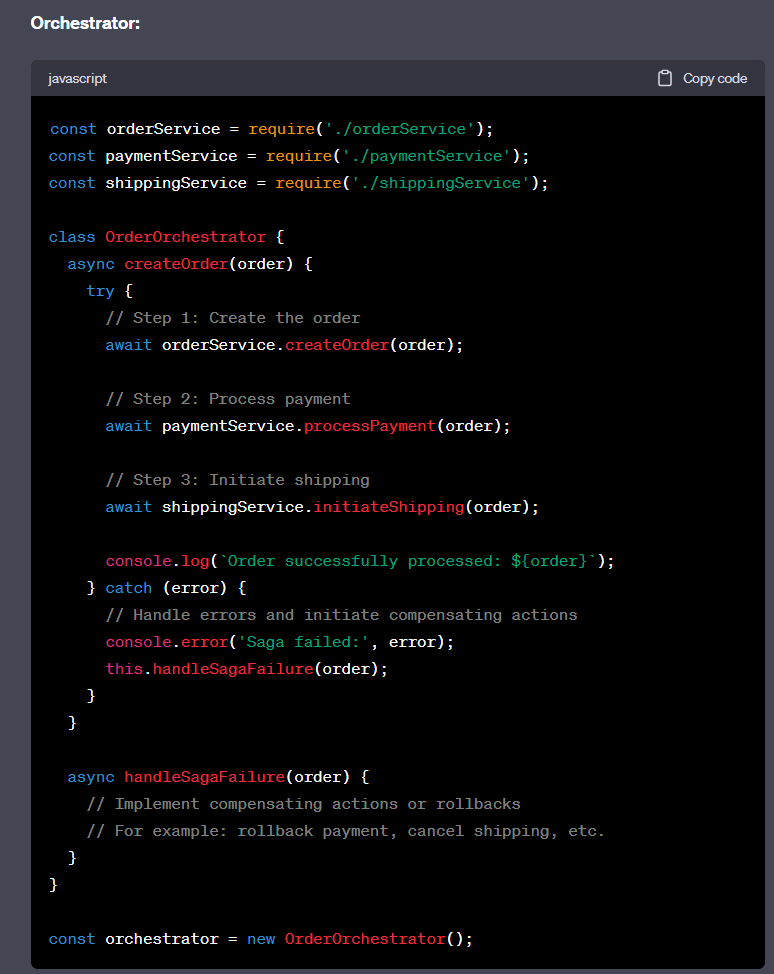
Asynchronous Feedback: After the user's request to create an order is accepted by the OrderService, the service can respond with an acknowledgment or a provisional order ID. The actual outcome of the saga (successful order creation, payment, and shipping) occurs asynchronously. You might send notifications or emails to the user when the saga successfully completes, or if any step fails, you can send an appropriate notification explaining the issue.

Polling or Long Polling: The user's request can return an acknowledgment, and the client can then periodically poll the system to check the status of the order. This approach involves repeatedly querying the system until the saga is completed and the final outcome is known.

Webhooks: Provide a webhook URL during the order creation process. The microservices involved can send callbacks to this URL whenever significant events occur in the saga, updating the user or external system on the progress of the order.

APIs for Status Inquiry: Implement an API endpoint that allows users to query the status of their orders. This endpoint can provide details about the different stages of the order process and whether it has been successfully completed.

In an orchestration-based saga, there is a central coordinator (orchestrator) that controls the flow of the saga by explicitly defining the sequence of steps and interactions between microservices. Unlike the choreography-based saga,



When Not to Use: For simple workflows without complex interactions between microservices.

Life Example: Uber uses the saga pattern to manage complex ride-hailing processes involving multiple microservices.

## Event-Driven/ Publish-Subscribe

Problem: Synchronous communication between microservices can lead to tight coupling and scalability challenges.

Solution: Implement an event-driven architecture where microservices communicate asynchronously through events.

Code Implementation: Use message brokers like Apache Kafka or RabbitMQ to publish and subscribe to events.

A screen shot of a computer program

Description automatically generated

A computer screen shot of a program

Description automatically generated

When Not to Use: For applications where real-time communication isn't critical or when event complexity is low.

Life Example: Amazon's shopping platform uses event-driven communication to handle inventory updates, order processing, and shipping notifications.

# Observability Patterns

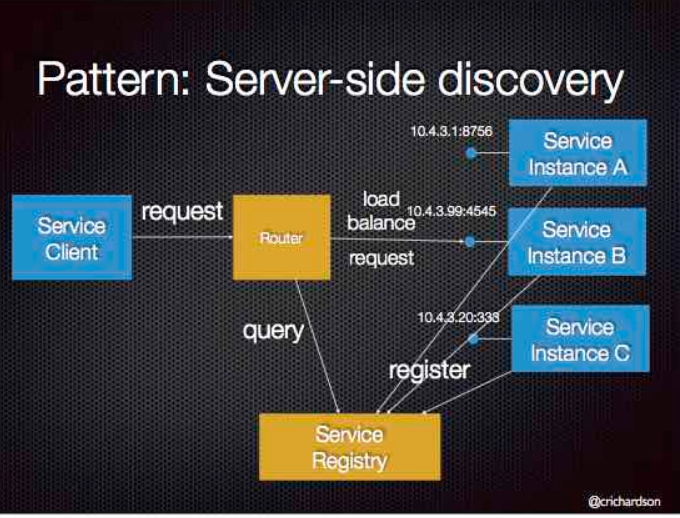
These patterns enhance the ability to monitor, trace, and debug microservices.

As a service, you must implement various cross-cutting concerns such as metrics, reporting exceptions to an exception tracker, logging, distributed tracing, health checks, externalized configuration, and security.

# Cross-Cutting Concern Patterns

These patterns address common concerns that span multiple microservices.

## Service Discovery



Problem: In a dynamic microservices environment, it's challenging for services to locate and communicate with each other.

Solution: Implement a service discovery mechanism where services can register themselves and discover others. When making a request to a service, the client makes a request via a router (a.k.a load balancer) that runs at a well known location. The router queries a service registry, which might be built into the router, and forwards the request to an available service instance.

Code Implementation: Use tools like Netflix Eureka or HashiCorp Consul to manage service registration and lookup

When Not to Use: In very small applications where the number of services is limited and static.

Unless it’s part of the cloud environment, the router must is another system component that must be installed and configured. It will also need to be replicated for availability and capacity.

The router must support the necessary communication protocols (e.g HTTP, gRPC, Thrift, etc) unless it is TCP-based router

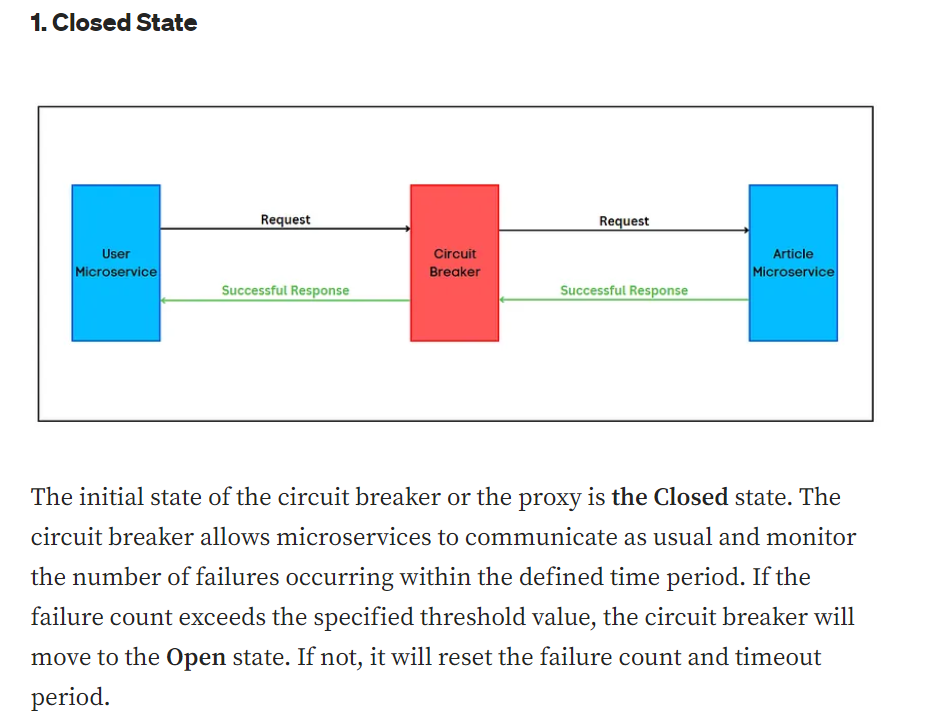
Life Example: An AWS Elastic Load Balancer (ELB) is an example of a server-side discovery router.

## Circuit Breaker

Problem: Microservices might experience failures or slowdowns, leading to cascading failures in the calling services.

Solution: Implement a circuit breaker pattern to detect and handle failures by temporarily stopping requests to a failing service.

Code Implementation: Understanding and implementing the circuit breaker pattern is pretty easy. It has three states: Closed, Open, and Half Open.



A screenshot of a computer

Description automatically generated

A diagram of a circuit breaker

Description automatically generated

Advantages of Circuit Breaker Pattern:

Helps to prevent cascading failures.

Handles errors gracefully and provides better under experience.

Reduces the application downtimes.

Suitable for handling asynchronous communications.

State changes of the circuit breaker can be used for error monitoring.

When Not to Use: For services that are not critical or where failures won't have significant impacts.

Need good infrastructure management to maintain circuit breakers.

Throughput issues in services if not properly configured.

Difficult to test.

Life Example: Netflix's Hystrix is used to prevent service failures from affecting the overall system performance.

Other [link](https://blog.logrocket.com/use-circuit-breaker-node-js/)

For Java SpringBoot — gs-cloud-circuit-breaker

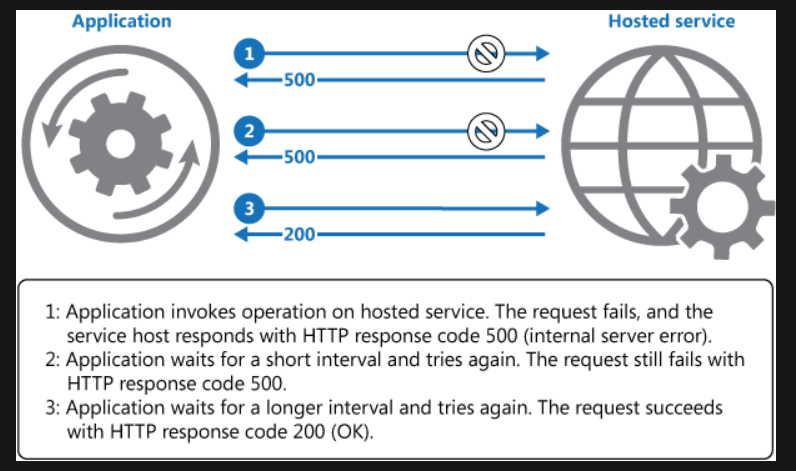
For TypeScript — circuit-breaker-js, @fastify/circuit-breaker

For Python — pycircuitbreaker

For .NET — Polly

# Some Important internal service design

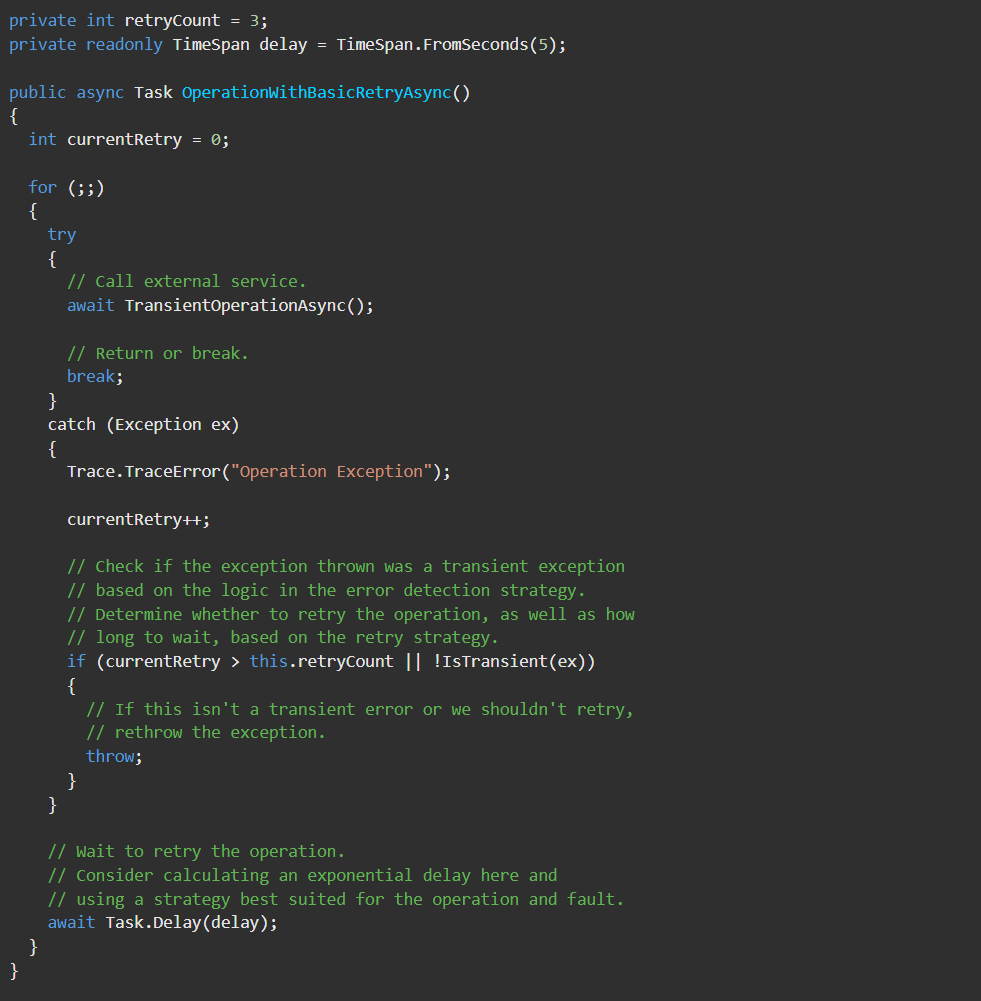
## Retry

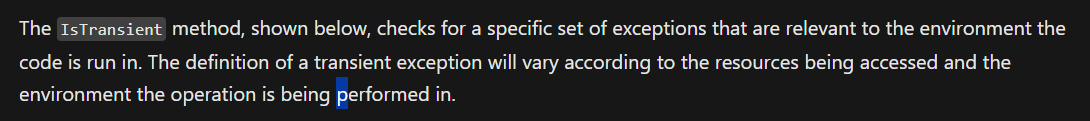


Problem: Transient failures can occur due to network issues or service instability, leading to incomplete requests.

Solution: Implement retry mechanisms to automatically retry failed requests after a delay.

Code Implementation: Incorporate retry logic using libraries like Spring Retry or Polly.





When Not to Use: When dealing with failures that are likely to persist for a longer duration.

For some noncritical operations, it's better to fail fast rather than retry several times and impact the throughput of the application.

Consider whether the operation is idempotent. If so, it's inherently safe to retry. Otherwise, retries could cause the operation to be executed more than once, with unintended side effects.

Life Example: For example, a database service that's processing a large number of concurrent requests can implement a **throttling strategy** that temporarily rejects any further requests until its workload has eased. An application trying to access the database might fail to connect, but if it tries again after a delay it might succeed.

<https://www.npmjs.com/package/async-retry>

## Trace

A trace in the context of microservices refers to the ability to track and monitor the flow of requests as they travel through various components of a distributed system.

Popular tools for distributed tracing include OpenTelemetry, Jaeger, Zipkin.

Trace ID: uniquely identifies an entire transaction or request across different services.

Span ID: uniquely identifies an individual operation or work within a service.

Parent ID: connects spans in a hierarchical manner, showing the sequence of operations in the processing of a request.

## gRPC

gRPC is a modern open source high performance Remote Procedure Call (RPC) framework that can run in any environment. It can efficiently connect services in and across data centers with pluggable support for load balancing, tracing, health checking and authentication.

In gRPC, a client application can directly call a method on a server application on a different machine as if it were a local object, making it easier for you to create distributed applications and services.

gRPC's design and use of HTTP/2 make it faster and more efficient than traditional HTTP in many cases, especially for specific use cases common in distributed systems and microservices architectures.

gRPC is generally faster than traditional HTTP in certain scenarios. gRPC stands for "Remote Procedure Call" and is a framework developed by Google for building efficient and high-performance APIs. It uses the HTTP/2 protocol as its underlying transport mechanism, which offers several performance improvements over the older HTTP/1.1 protocol used by traditional HTTP.

Some factors that contribute to gRPC's potential speed advantage over HTTP:

Binary Protocol: gRPC uses a binary serialization format (Protocol Buffers) instead of text-based formats like JSON used in traditional HTTP APIs. This makes data transmission more compact and efficient.

Multiplexing: HTTP/2, which gRPC is built on, supports multiplexing, allowing multiple requests and responses to be sent over a single connection simultaneously. This reduces the overhead of establishing multiple connections for different requests, leading to better resource utilization.

Header Compression: HTTP/2 employs header compression techniques that reduce the overhead associated with sending headers, resulting in faster communication.

Bidirectional Streaming: gRPC supports bidirectional streaming, which means the client and server can send multiple messages in a single connection without waiting for previous responses. This can lead to improved latency and better utilization of network resources.

Connection Reuse: gRPC encourages the use of long-lived connections, which can be more efficient than creating new connections for every request in traditional HTTP.

However, it's important to note that the performance advantage of gRPC over HTTP is most pronounced in situations where there are frequent small requests or a need for real-time communication. For larger payloads or scenarios where compatibility with a wider range of clients is important (such as web browsers), traditional HTTP might still be a better choice

[A screenshot of a computer program

Description automatically generated](https://github.com/mguay22/nestjs-grpc/blob/main/proto/auth.proto)

## Applying Domain-Driven Design principles to a Nest.js project

<https://dev.to/microtica/the-concept-of-domain-driven-design-explained-1ccn>

<https://dev.to/bendix/applying-domain-driven-design-principles-to-a-nest-js-project-5f7b>

<https://github.com/hbendix/domain-driven-nest/blob/master/src/Domain/User/User.ts>

Microservices is an architecture design model with a specific bounded context, configuration, and dependencies. These result from the architectural principles of the domain-driven design and DevOps. Domain-driven design is the idea of solving problems of the organization through code.

Domain-driven design bases on the business domain. Modern business environments are very complex and wrong moves can lead to fatal outcomes. Domain-driven design solves complex domain models, connecting to the core business concepts.

Domain logic - Domain logic is the purpose of your modeling. Most commonly, it’s referred to as the business logic. This is where your business rules define the way data gets created, stored, and modified.

Domain model -Domain model includes the ideas, knowledge, data, metrics, and goals that revolve around that problem you’re trying to solve. It contains all the rules and patterns that will help you deal with complex business logic. Moreover, they will be useful to meet the requirements of your business.

Subdomain - A domain consists of several subdomains that refer to different parts of the business logic. For example, an online retail store could have a product catalog, inventory, and delivery as its subdomains.

Design patterns - Design patterns are all about reusing code. No matter the complexity of the problem you encounter, someone who’s been doing object-oriented programming has probably already created a pattern that will help you solve it.

Bounded context - Bounded context is a central pattern in domain-driven design that contains the complexity of the application. It handles large models and teams. This is where you implement the code, after you’ve defined the domain and the subdomains.

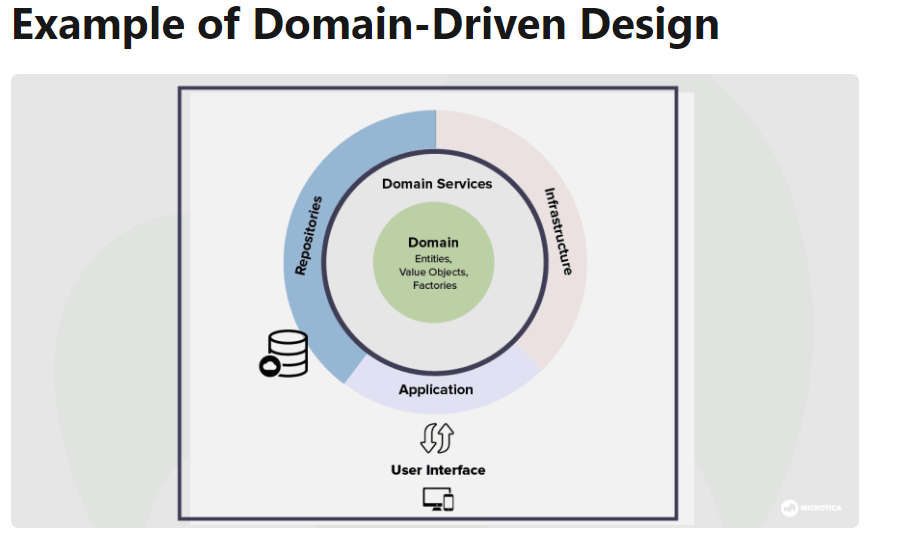
Bounded contexts actually represent boundaries in which a certain subdomain is defined and applicable. Here, the specific subdomain makes sense, while others don’t. One entity can have different names in different contexts. When a subdomain within the bounded context changes, the entire system doesn’t have to change too. That’s why developers use adapters between contexts.

Entities - Entities are a combination of data and behavior, like a user or a product. They have identity, but represent data points with behavior.

Value objects and aggregates - Value objects have attributes, but can’t exist on their own. For example, the shipping address can be a value object. Large and complicated systems have countless entities and value objects. That’s why the domain model needs some kind of structure. This will put them into logical groups that will be easier to manage. These groups are called aggregates. They represent a collection of objects that are connected to each other, with the goal to treat them as units. Moreover, they also have an aggregate root. This is the only entity that any object outside of the aggregate can reference to.

Domain service - The domain service is an additional layer that also contains domain logic. It’s part of the domain model, just like entities and value objects. At the same time, the application service is another layer that doesn’t contain business logic. However, it’s here to coordinate the activity of the application, placed above the domain model.

Repository- The repository pattern is a collection of business entities that simplifies the data infrastructure. It releases the domain model from infrastructure concerns. The layering concept enforces the separation of concerns.



## Kafka

kafka is a distributed event streaming platform designed for high-throughput, fault-tolerant, and real-time data streaming. It's commonly used for building real-time data pipelines and streaming applications. Here are some key concepts:

* 1. Topic: A topic is a category to which records are published. Data in Kafka is organized into topics, and producers publish records (messages) to these topics.
  2. Producer: Producers are responsible for publishing records to Kafka topics. They send data to specific topics, which are then stored in the Kafka cluster.

const clientOptions: ClientOptions = {

transport: Transport.KAFKA,

options: {

client: {

clientId: 'nestjs-kafka-producer',

brokers: ['localhost:9092'], // Kafka broker addresses

},

consumer: {}, // You can configure more options here

},

};

this.kafkaClient = new ClientKafka(clientOptions);

await this.kafkaClient.emit('topic1', message).toPromise();

* 1. Consumer: Consumers subscribe to one or more topics and read records from them. Each record in a topic has an offset, which represents its position in the topic. Consumers can control their position in the topic by managing these offsets.

constructor() {

const clientOptions: ClientOptions = {

transport: Transport.KAFKA,

options: {

client: {

clientId: 'nestjs-kafka-consumer',

brokers: ['localhost:9092'], // Kafka broker addresses

},

consumer: {

groupId: 'nestjs-consumer-group',

},

},

};

this.kafkaClient = new ClientKafka(clientOptions);

}

async subscribeToTopics(topics: string[]) {

await this.kafkaClient.connect();

topics.forEach((topic) => {

this.kafkaClient.subscribeToResponseOf(topic);

this.kafkaClient

.getResponsePatternHandleByPattern(topic)

.subscribe((message) => {

this.logger.log(`Received message from ${topic}: ${message.value}`);

this.logger.log(`Offset: ${message.offset}`);

});

});

}

* 1. Broker: Kafka brokers are individual servers or nodes in the Kafka cluster. They store the published records and serve them to consumers. Brokers are responsible for handling the storage and retrieval of data.
  2. Partition: Topics can be divided into partitions, which are ordered and immutable sequences of records. Partitions allow Kafka to scale horizontally across multiple brokers and enable parallel processing of data.

async createTopic(topic: string, numPartitions: number) {

await this.kafkaAdmin.connect();

const newTopic: NewTopic = {

topic,

numPartitions,

replicationFactor: 1,

};

await this.kafkaAdmin.createTopics({

topics: [newTopic],

});

await this.kafkaAdmin.disconnect();

}

* 1. Replication: Kafka uses replication for fault tolerance. Each partition can have multiple replicas distributed across different brokers. If one broker fails, another replica can take over to ensure data availability.
  2. Consumer Group: A consumer group is a group of consumers that work together to consume data from a topic. Each consumer within a group reads from a specific partition. Consumer groups enable parallel processing of data and load balancing.
  3. Offset: An offset is a unique identifier assigned to each record within a partition. Consumers use offsets to keep track of the records they've already processed.
  4. Retention Policy: Kafka allows you to set retention policies for topics. This determines how long records are retained in a topic before they are deleted. Retention can be based on time or size.
  5. Stream Processing: Kafka Streams is a library that allows you to process and analyze data streams in real-time. It enables transformations, aggregations, and joins on data as it flows through Kafka topics.

**Example: Logging System**

Imagine you're building a logging system for a web application that generates a significant amount of log data. You decide to use Kafka to collect and process these logs in real-time. In this scenario:

1. **Topic**: Let's call our topic "web\_logs." This topic will store all the log entries generated by the web application.
2. **Partitions**: Since the logging system needs to handle a large volume of log data, you decide to divide the "web\_logs" topic into multiple partitions. Let's say you choose to have three partitions for this topic.
   * Partition 0: This partition might store log entries related to user authentication and session management.
   * Partition 1: This partition could handle logs related to user interactions, such as page visits and clicks.
   * Partition 2: This partition might store logs related to backend operations, database queries, and API requests.

With this setup, each partition becomes a separate "stream" of log data within the "web\_logs" topic. Logs related to different aspects of the application are stored in different partitions. This separation allows for better parallelism, efficient processing, and ordered delivery within each partition.

**Benefits of Partitions**:

* **Scalability**: If the application generates more logs over time, you can add more brokers and partitions to scale the system horizontally.
* **Parallelism**: Consumers can process logs from different partitions in parallel, allowing for efficient utilization of resources.
* **Fault Tolerance**: With replication and multiple brokers, the loss of one broker or partition doesn't result in data loss.
* **Ordered Delivery**: Logs within a partition are guaranteed to be ordered. While logs from different partitions might not have a global order, you can still process each partition's logs in sequence.
* **Retention Policies**: You can set different retention policies for different partitions based on the importance and relevance of the log data.

**Consumer Groups**:

Now, let's say you have two separate applications: an analytics application and an alerting system. You decide to create two consumer groups, one for each application, to process the log data from the "web\_logs" topic.

* **Consumer Group for Analytics**: This group might have multiple consumers, each responsible for analyzing logs from a specific partition. The consumers within this group process the logs in parallel and generate insights and reports.
* **Consumer Group for Alerting**: This group might also have multiple consumers, with each consumer monitoring logs from a specific partition for anomalies or critical events. The consumers in this group work in parallel to trigger alerts based on the log data.

In this example, the combination of partitions and consumer groups allows you to efficiently manage and process log data from a web application, achieve parallelism, ensure ordered processing, and cater to the different needs of analytics and alerting applications.