Microservices Design Patterns and Principles

Introduction

The purpose of this guide is to help you with architectural patterns followed while designing Microservices architecture.

Abstract

This document aims to provide basic know-how of the 'Microservices Design Patterns.' We aim to provide a reference document for design patterns to be used when any application is designed using Microservices architecture.

We aim to publish this whitepaper as a ready reference to ten highly used design patterns related to Microservices in a detailed fashion.

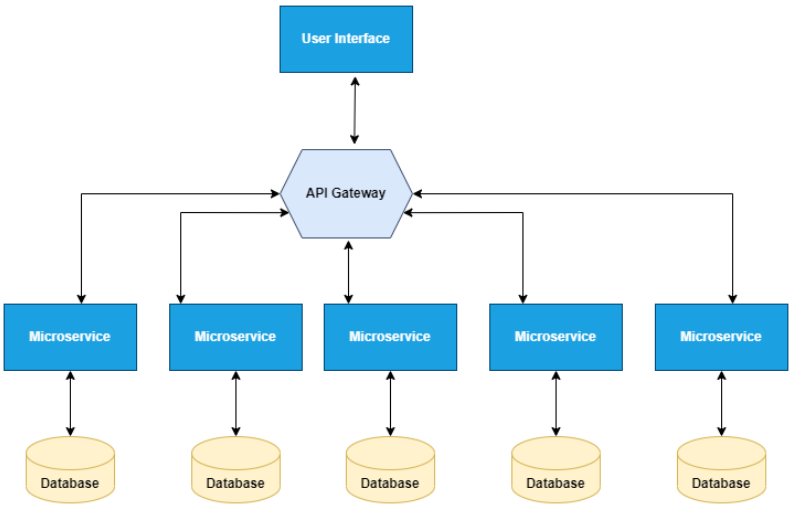
Introduction to Microservices

Microservices are self-contained and independent deployment modules. In Microservices, the application is divided into independent modules based on business domains. Microservices are designed based on **Domain Driven Design (DDD)**, which says the application should be modeled around independent modules with bounded context for the specific business domain. As we now have independent modules, we have faster rollout and quicker release cycles. All the modules can be worked on in parallel, and any changes will only affect that specific module. Therefore, Continuous Integration (CI) & Continuous Delivery (CD) are key advantages of using Microservices architecture.

> In short, the Microservices architectural style is an approach to developing a single application as a suite of small services, each running in its own process and communicating with lightweight mechanisms, often an HTTP resource API. These services are built around business capabilities and are independently deployable by fully automated deployment machinery.
>
> **– Martin Fowler**

As Monolithic applications pose their own set of challenges, here are four main concepts that describe the importance of Microservices architecture over Monolithic architecture. Some of the key points about Microservices are:

1. Loosely coupled components designed around business domains
2. Application can be distributed across different clouds and data centers
3. Change management is easy in Microservices

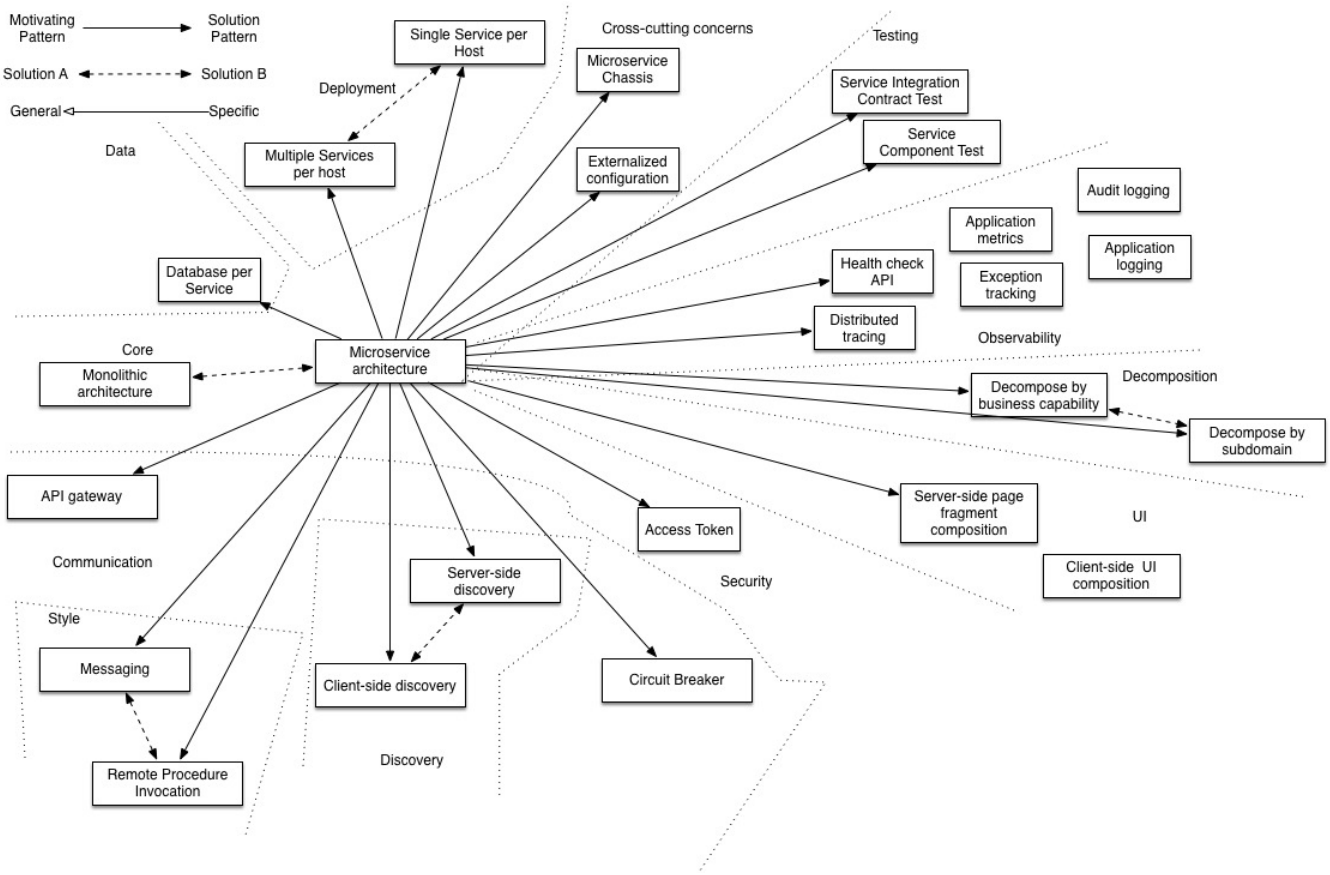


Microservices Design Patterns and Principles

As we now know, Microservices are independent components that encapsulate business domains. If applications are architected using Microservices design principles and patterns, then the application is highly scalable. So, let's list down the principles in which the Microservices architecture has been built.

1. Scalability
2. Flexibility
3. Independent and autonomous
4. Decentralized governance
5. Resiliency
6. Failure isolation
7. Continuous delivery through the DevOps

While adhering to the principles mentioned above, there are some standard sets of problems & complex scenarios that architects and developers need to address. In addressing such common problems, statements bring proven solutions to the table, which become patterns. Here are a few high-level patterns that are used around Microservices.

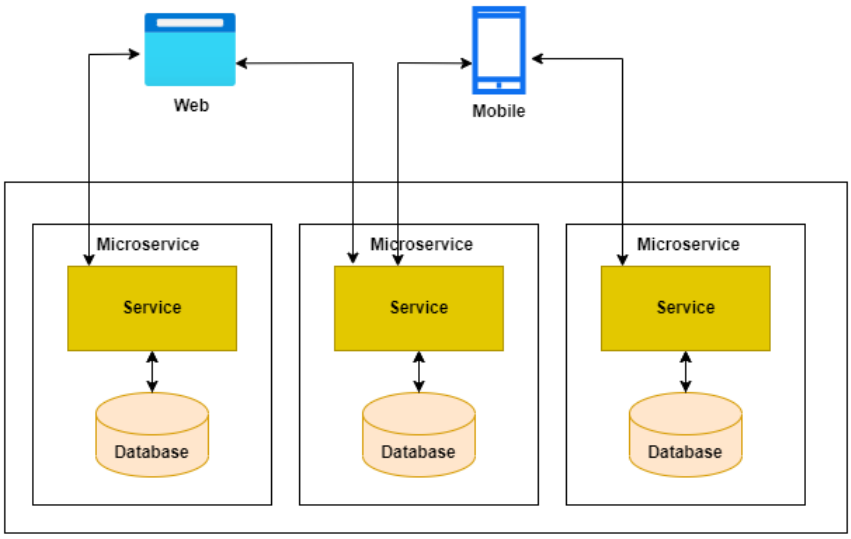


Picture Reference: https://microservices.io/patterns/microservices.html

We will cover the most used and discussed design patterns in detail, along with use cases. Here is the list of design patterns that will be covered:

1. Database per Microservices
2. CQRS
3. Event Sourcing
4. Saga
5. BFF
6. API Gateway
7. Strangler
8. Circuit Breaker
9. Externalized Configuration
10. Consumer-Driven Contract Tracing

Database per Microservices



Microservices are generally independent and loosely bound components. So, to achieve this independent nature, every Microservice must have its own database so that it can be developed and deployed independently.

Let's take the example of an e-commerce application. We will have product, ordering and SKU Microservices that each service interacts (store and retrieve) with data from their own databases. Any changes to one database don't impact other Microservices.

Other Microservices can't directly access each independent database. Each component's persistent data can only be accessed via APIs. So, Database per Microservices provides many benefits, especially to evolve rapidly and support massive scaling as they make each Microservice independent.

Database changes can be made independently without impacting other Microservices:

- Each database can scale independently
- Microservices domain data is encapsulated within the service
- If one of the database servers is down, this will not affect other services

Also ***Polyglot data persistence** gives the ability to select the best-optimized storage needs per Microservices. So, we can use this as an example - an e-commerce application with Microservices, as mentioned earlier. Here are the optimized choices:

- The product Microservices use a NoSQL document database for storing catalog-related data, which is storing JSON objects to accommodate high volumes of read operations
- The shopping cart Microservices use a distributed cache that supports its simple, key-value datastore
- The ordering Microservices use a relational database to accommodate the rich relational structure of its underlying data

Because of the ability to scale and high availability, NoSQL databases are getting highly popular and are widely used in enterprise applications nowadays. Also, their support for unstructured data gives flexibility to developments on Microservices components.

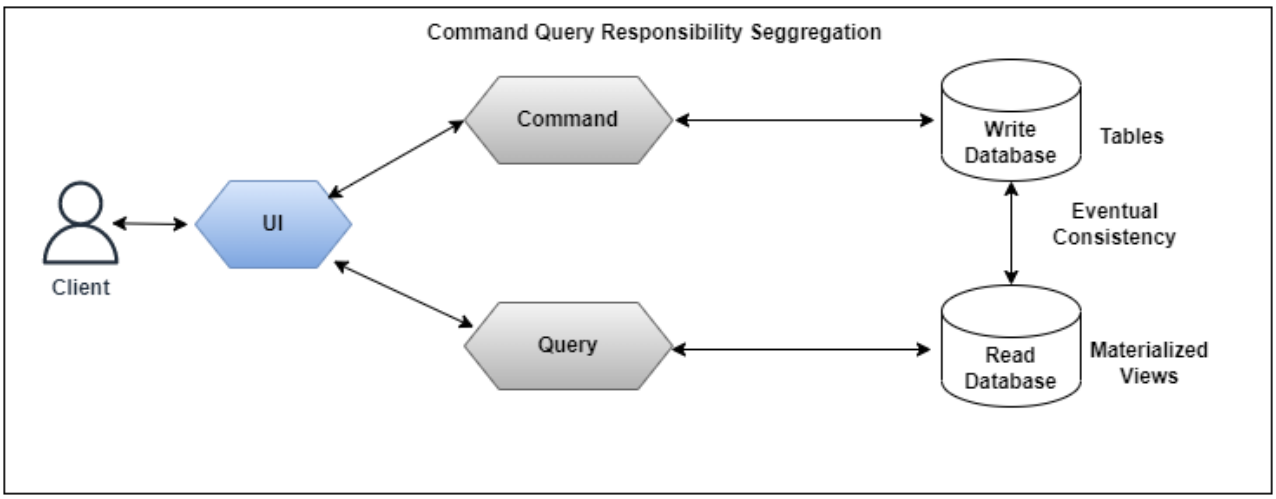
CQRS Design Pattern

CQRS stands for Command and Query Responsibility Segregation. It is one of the most widely used patterns for querying the database in Microservices architecture. CQRS is very handy when we need to eliminate complex queries involving inefficient joins. This pattern promotes the complete separation of read and write concerns in the database.

Traditionally, in Monolithic or SOA architecture, we have a single database for the entire application, and these databases will respond to both read and write requests. As the application becomes more complex over time, reading and writing to the database become non-performant. Sometimes, it has been observed that some applications follow an active-passive database model. However, even then, only one database copy is active at a time, so performance issues still persist.

For instance, in the case of database reads, if an application requires a query that needs to join more than ten tables, it can lead to locking the database due to the latency of query computation. Similarly, when writing to the database, we may need to perform complex validations and process lengthy business logic for some CRUD operations, which can also cause the locking of database operations.

So, reads and writes to the database are different operations for which separate strategies can be defined. To achieve this, CQRS offers to use “separation of concerns” principles and separate reads and writes into two databases. By this principle, we can use different databases for reading and writing database types, like using NoSQL for reading and using a relational database for CRUD operations.



So, another factor that is considered here is the nature of the application. If the use cases of your application mostly need to read the data in comparison to writing, then your application is a read-intensive application. So accordingly, you should focus

and choose your read and write databases. So here, CQRS separates reads and writes into different databases, where commands perform the creation or updating of data, and queries perform read data.

Commands are actions with some defined operations like “add the item to bucket” or “check my balance.” So, commands can be published via message brokers, which help applications process them in an async manner. Queries never modify the database. Queries always return the JSON data with DTO objects. In this way, we can isolate the Commands and Queries.

How to Sync Databases with CQRS?

When we segregate read and write concerns in two different databases, the primary consideration is to sync these two databases properly. So, both databases should always be synced.

This can be achieved using **Event Driven Architecture**. As per Event Driven Architecture, when an update command is issued to the write database, it will publish an update event using message broker systems, and this will be consumed by the read database, which will pull the latest changes from the write database to keep the read database in sync.

But this option creates a consistency issue because the data would not be reflected immediately since async communication is implemented with message brokers. This will operate the principle of “**eventual consistency**.” Eventual consistency is a property of distributed computing systems such that the value for a specific data item will, given enough time without updates, be consistent across all nodes. The read database eventually synchronizes with the write database, which will take some time to sync.

So, if we return to our read and write databases in the **CQRS pattern** when you start on the design, you can initially create a read database from the replicas of the write database. Also, as we have separate read and write databases, it means that both are scalable independently.

Event Sourcing

With CQRS, as reads and writes are separated, we need to keep both the database in sync as well, which is achieved using Event-Driven Architecture. But any failure in this sync or any loss of event can make data inconsistent in one of the databases, so to overcome this scenario, Event Sourcing is used. In Event Sourcing, apart from publishing the events on message broker systems, we store the events in the write database, and this event is stored in a single source of truth. In case of failure, events can be replayed, which will help keep data consistent.

Let's understand Event Sourcing with an example. Suppose there is a user’s table and a user updates their details. Usually, the updated details will override the existing values. This is generally how most applications work, i.e., they always store the entity’s current state.

But with the Event Sourcing pattern, it is changing. Instead of storing just the latest state, we store all operations in the database. The Event Sourcing pattern suggests saving all events into the database with a sequential order of data events. This events database is called an event store.

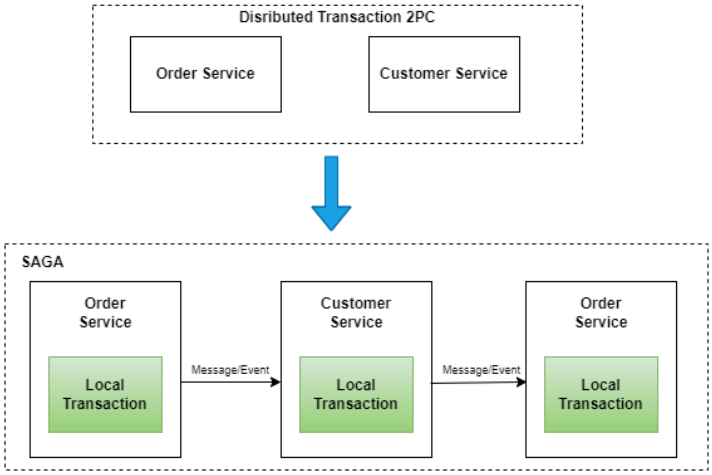
So, the event store is considered as a single source of truth for the data. After that, these event stores are converted to a read database using the materialized views. This conversion operation can also be handled by publish/subscribe patterns with events published through message broker systems. Moreover, this event list gives us the ability to replay events at a given timestamp. This allows us to build the latest status of data in case of any failure.

Let’s take the use case of a shopping cart for our e-commerce application, where we applied CQRS and Event Sourcing. As you can see in the image below, for the shopping cart, every user’s actions are recorded in the event store with appended events. All these events are combined and summarized on the read database with denormalized tables into materialized view database.

By applying these patterns, we can query the latest status of the shopping cart using materialized views, which are used to create a read database. Instead of storing actual data, it stores sequential events that denote user actions, allowing us to know the history of actions the user has taken with timestamps. This enables us to retrieve the status of shopping cart data at any point in time. We can use event store write databases with Azure Cosmos DB, Cassandra, Event Store databases, etc.

SAGA

Usually, Microservices are designed around business domains, so the application is divided into multiple Microservices. However, there is a challenge, as now, any transaction will also be distributed across the services. To handle this distributed transaction scenario Saga pattern is used.



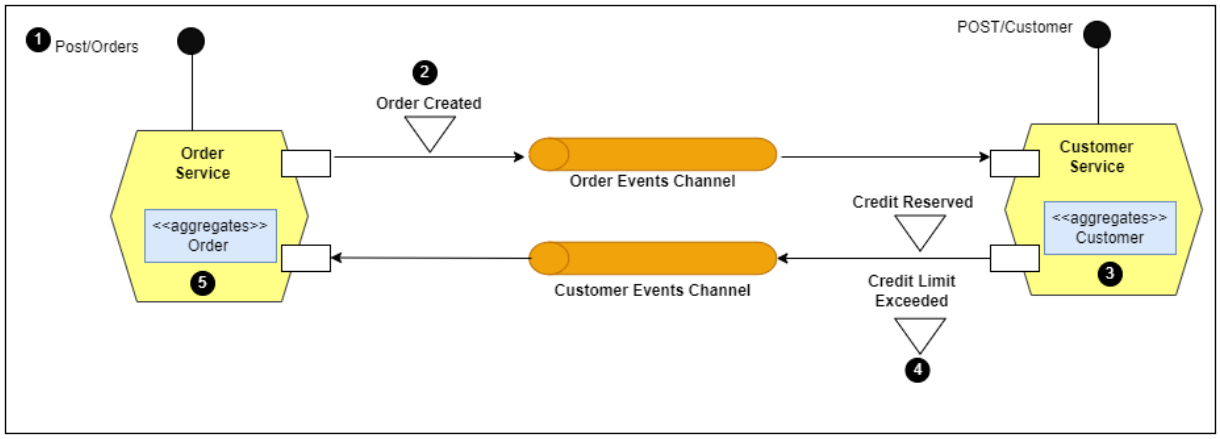
The Saga design pattern is used for managing data consistency in the case of distributed transactions. The Saga pattern tends to create a set of transactions that update each Microservice sequentially and publish an event to trigger the following transaction for the next Microservices. If the transaction fails at any step or service, then the Saga patterns trigger to rollback event, which basically does reverse operations in each of the Microservices.

A saga is a sequence of local transactions. Each local transaction updates the database and publishes a message or event to trigger the next local transaction in the saga. If a local transaction fails because it violates a business rule, then the saga executes a series of compensating transactions that undo the changes that were made by the preceding local transactions.

There are two options for how SAGA is implemented:

- **Choreography** - Each local transaction publishes domain events that trigger local transactions in other services
- **Orchestration** - An orchestrator (object) tells the participants what local transactions to execute

Choreography-based saga



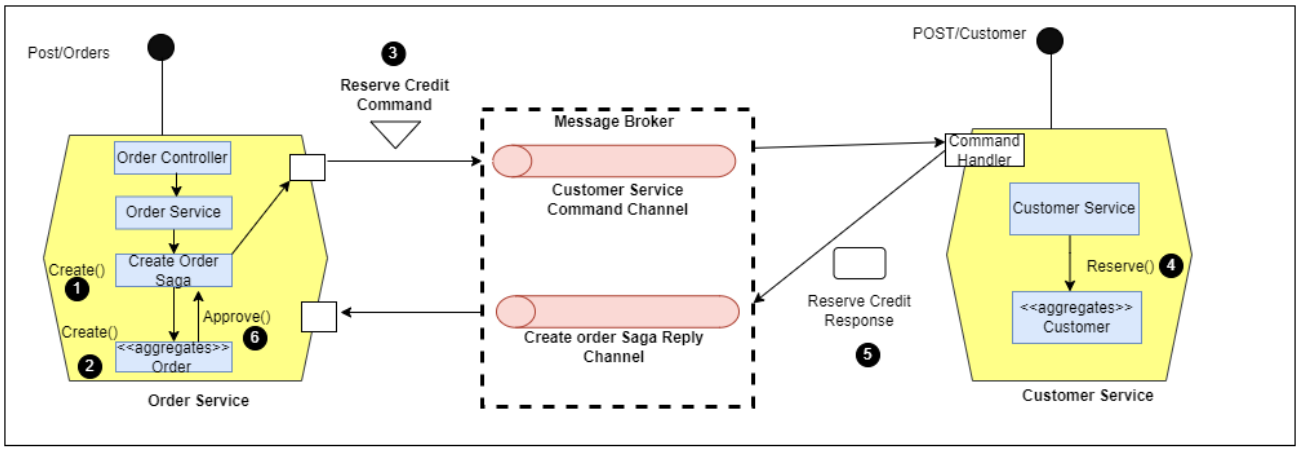
Here is the e-commerce service using events to implement choreography-based SAGA.

Choreography enables the implementation of the saga pattern by applying publish-subscribe principles. With choreography, each Microservice executes its own local transactions and publishes events to a **message broker system** that triggers **local transactions** in other Microservices.

In the above diagram, the order service receives a post request for an order at step 1 and updates the local database. Then, it publishes the 'order created' event on the order event channel. At the same time, the customer service is a subscriber to the order event channel and will receive an event notification. On receiving notification, customer service triggers its own local tasks, like checking if the customer has appropriate credits or not and then reserving credits for the order for which the notification was received. Once done, it will either emit a 'credit reserved' or 'credit limit exceeded' event in the customer event channel, which is subscribed by the order service. This is how the entire distributed transaction works.

For fewer services, this flow works well. Still, suppose the number of services are more. In that case, it becomes complex, and apart from its own local tasks, each service has the additional responsibility of emitting & consuming events as well, which makes things furthermore complex, and that's where the Orchestration-based saga will solve the issue.

Orchestration Based Saga



The Orchestrator-based saga also operates on events, but the responsibility for emitting events and ensuring the atomicity of transactions shifts from the services themselves to the orchestrator. Orchestrator commands services to execute local transactions. The orchestrator acts as an agent who tells services to execute local transactions and maintains the status of the complete transaction.

In the above diagram, the order service receives a post request for placing an order at step 1. Order service will initiate a transaction here and create a saga orchestrator. This orchestrator will ask the order service to create an order in the pending state. Now the orchestrator will send a “Reserve credit” command to customer service. Customer service will reserve credit and send the status of its local transaction channel, which the orchestrator consumes. Now, if the orchestrator receives a positive status from customer service, it will ask the order service to update the order status from “pending” to “In-Progress.” But, if the credit is not reserved, then the entire transaction will be rolled-back from the order and customer service.
As the orchestrator now takes care of emitting events, this type of implementation is suitable for complex workflows involving many services. New services might also be added in the future. Since the orchestrator controls every transaction, there is less chance of getting into cyclical dependency.

Use the Saga pattern when you need to:

- Ensure data consistency in a distributed system without tight coupling
- Roll back or compensate if one of the operations in the sequence fails

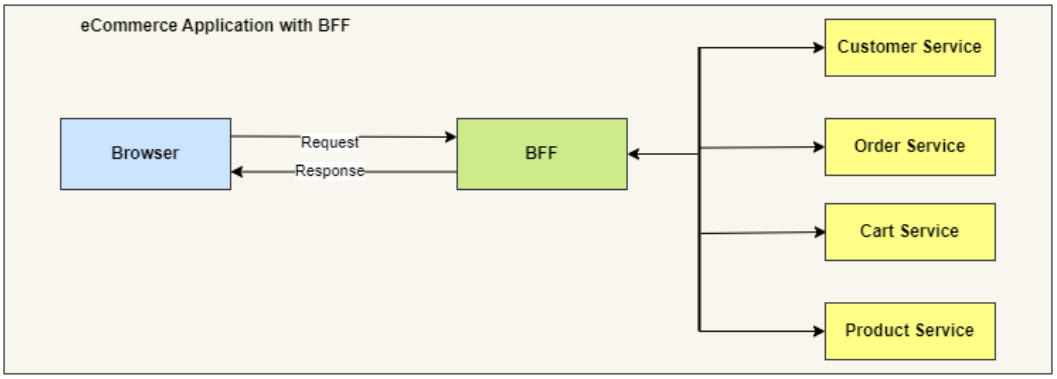
BFF (Backend for Frontend)

As and when an application is designed in Microservices architecture, there may be cases where data to be displayed in the frontend, be it web or mobile, might be coming from multiple services. In such cases, it becomes the responsibility of the frontend team to transform and aggregate responses from multiple Microservices. However, there are a few challenges with this approach:

- The frontend team needs to take care of a lot of transformations and aggregations
- Also, the browser or mobile app interface will use more resources for rendering the page

BFF patterns solve these challenges by introducing an intermediate layer between frontend and backend services. The BFF layer acts as a proxy that will call all backend services, aggregate and transform the response as per the needs of the frontend, and expose ready-to-consume responses that best fit the frontend. Therefore, there will be very minimal logic on the frontend. Hence, a BFF helps to streamline the data formatting process and takes up the responsibility of providing a well-focused response for the frontend to consume.

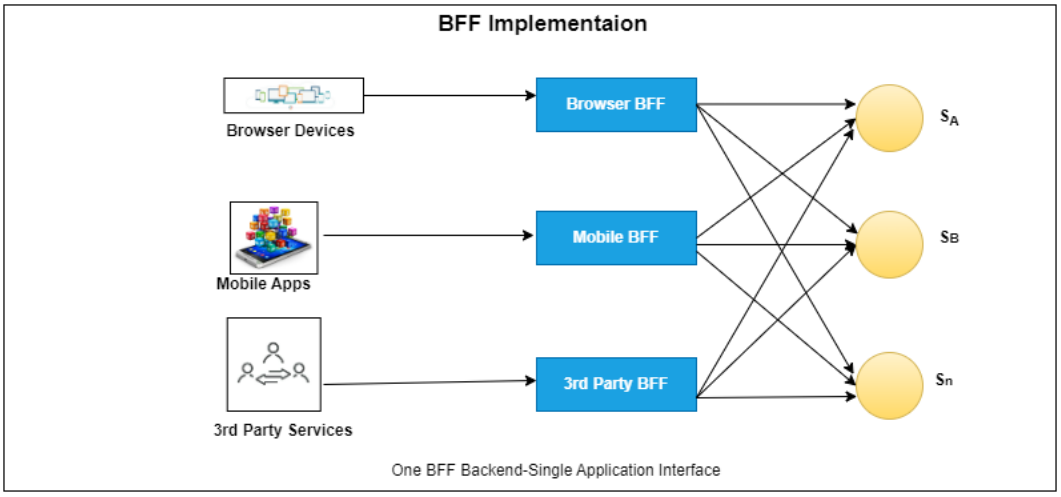
Here is an example showing BFF:



Ideally, the front-end team will be responsible for managing the BFF in most cases. A single BFF is focused on a single UI and that UI only. As a result, it will help us keep our frontends simple and see a unified view of data through its backend.

Usage of BFF again depends on the needs and architectural requirements. If the application you are designing or migrating to Microservices is simple, where each service has its own UI, then BFF is not needed. However, if the application has a complex functionality that involves multiple third-party integrations and front-end screens need to show processed data from multiple services, BFF is the best fit. Also, if the client's requirement is around optimal rendering of the frontend and backend, and you have complex Microservices, a BFF is suitable to use.

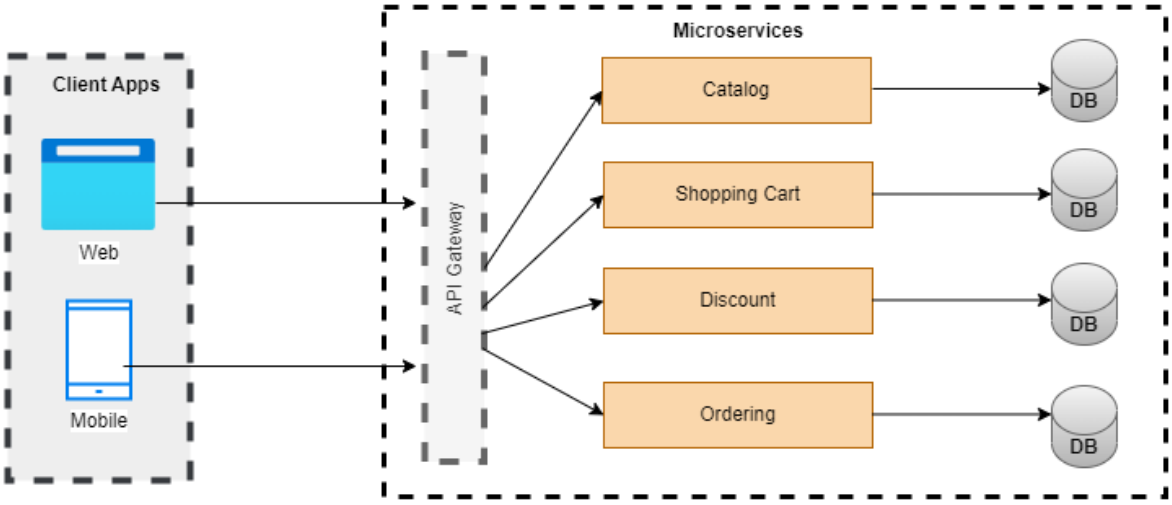
Furthermore, there can be multiple BFFs in the application depending on the needs and requirements of the application. In a scenario where the same backend code serves both web and mobile, where each shows data differently, there can be one BFF layer for the web and one BFF layer for mobile.



To conclude, implementing these patterns should be done judiciously, and code duplication should be avoided.

API Gateway

If we need to build a large, complex Microservices-based application with multiple clients, we can use the API gateway pattern. This pattern applies to distributed systems and acts as a reverse proxy or gateway routing for requests coming in from clients. API gateway acts as communication between the client and underlying services and serves as a single-entry point to the system while abstracting the underlying architecture. It also provides cross-cutting concerns like authentication, SSL termination, and caching.



So, as you can see, the API gateway provides a single entry point to multiple services. There are some things that we need to make sure of while using this pattern.

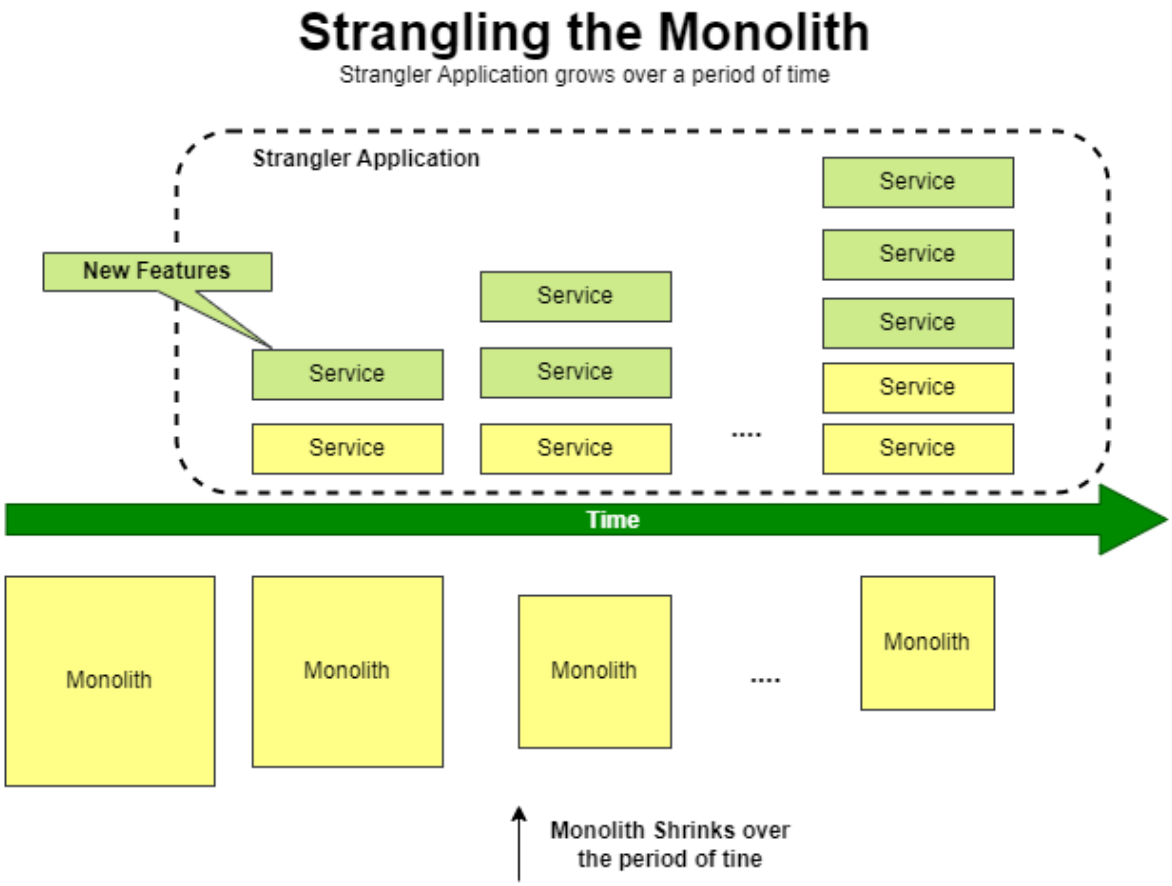
If there is only one node acting as an API gateway, there is a chance it will become a single point of failure. This situation needs to be handled, as if more complex logic is added to the API Gateway, it will become an anti-pattern. As the usage of this pattern grows, it is advised to deploy multiple API Gateways for multiple services based on the use case and the number of services.

In summary, we need to be cautious about using a single API Gateway, as it should be decided based on the business context of the client applications. Ideally, it is not advised to have one API gateway for all the internal Microservices.

Strangler Pattern

The Strangler Pattern is a methodology used in Microservices architecture to gradually migrate a Monolithic application to a Microservices-based architecture. This pattern is useful when the application is too large and complex to migrate simultaneously.

The Strangler Pattern involves designing a new set of Microservices that gradually take over the functionality of the Monolithic application. Initially, all requests for specific functionality are divided between the traditional Monolith and the new Microservice to cater to the same functionality. Over time, as the Microservices mature, they start to handle more of the traffic, and the Monolithic application handles less. Slowly, the entire complex Monolith will be replaced by more agile and scalable Microservices.

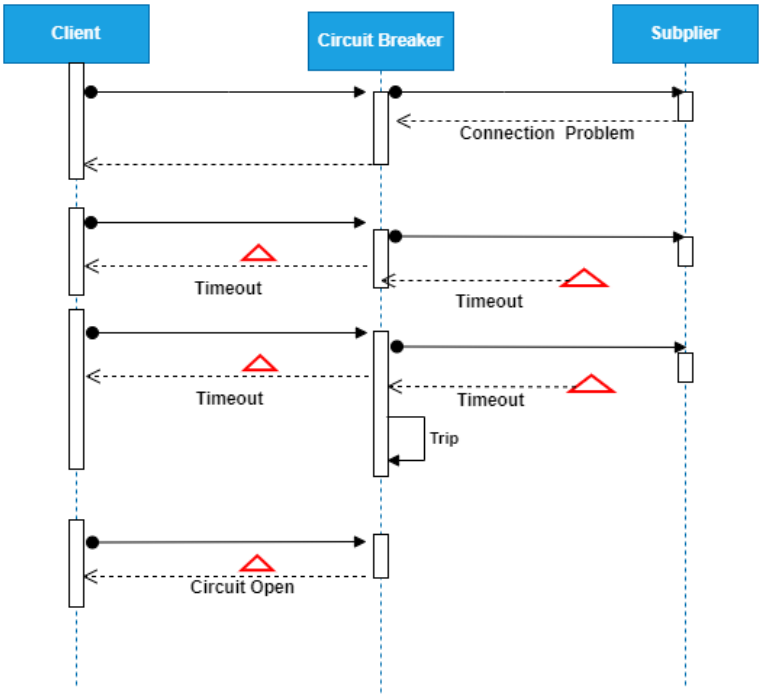


The Strangler Pattern follows these basic steps:

- Identify the functionality of the Monolithic application around the business domain that can be converted to a Microservice
- Create a new set of Microservices from scratch to handle the identified functionality
- Then, we must configure partial traffic to the new Microservices using an API gateway or any proxy server
- Over a period, gradually increase the traffic to the Microservices while decreasing the traffic to the Monolithic application
- Keep monitoring the performance of the new Microservices and adjust the traffic accordingly
- Eventually, route entire traffic to Microservices and bring down the entire Monolithic service

Circuit Breaker

The Circuit Breaker Pattern is a technique used in Microservices architecture to improve the resilience and reliability of distributed systems. The basic idea behind the circuit breaker is very simple. You wrap a function call in a circuit breaker object, which monitors it for failures. Once the failures reach a certain threshold, the circuit breaker trips and all further calls to the circuit breaker return an error, with any further calls skipped. The Circuit Breaker can also be configured to handle failures in different ways, such as returning a default value or a cached response.



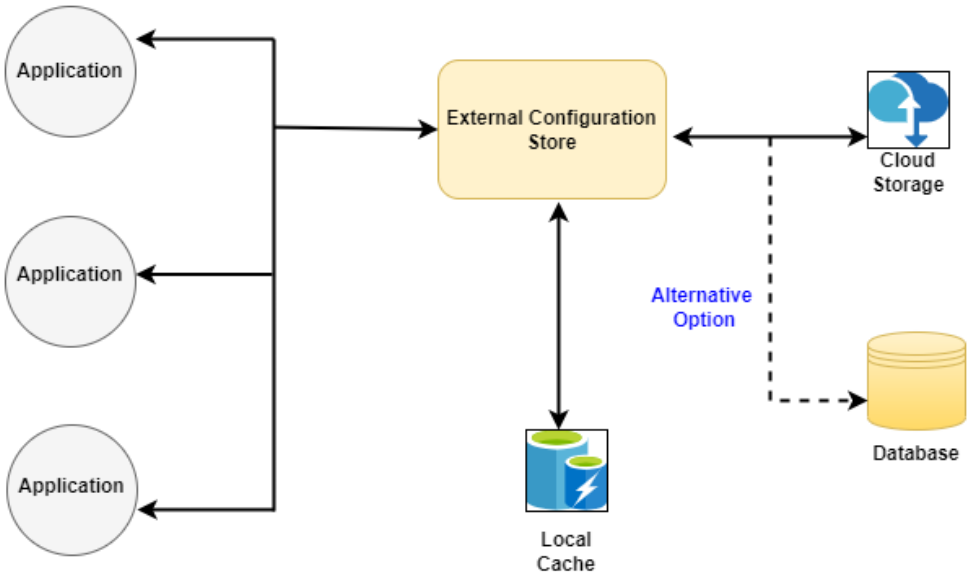
Here are the basic steps involved in the Circuit Breaker Pattern:

- Identify the Microservices that need to communicate with each other
- Add a Circuit Breaker between the Microservices
- Add configurations to monitor the status of the Microservices and detect failures
- If a Microservice fails to respond, the Circuit Breaker can prevent further requests to the Microservice and handle the failure in an appropriate way
- In some cases, the default response is also configured, which will help to handle failures gracefully.

Externalized Configuration

In Microservices architectures, systems are divided into multiple Microservices, each one running in its own container. Each process can be deployed and independently scaled, which means there may be many instances of the same microservice running at a specific time.

Let's say we want to modify the configurations for a microservice that has been replicated almost a hundred times.
If the configuration for this microservice is packaged with the service itself, we need to do deployment again for each instance running. This can lead to an inconsistent state as there are high chances that a few instances still need to upgrade to a new build and are still running on old configurations. Therefore, it is advised that services share the external configuration.



It works by keeping the configurations in an external store, such as a database, file system, or environment variables. When the Microservice is deployed and started, it loads the configuration from the external source. At runtime, if configuration changes occur, those are generally reloaded by Microservices without any new deployments.

Consumer-Driven Contract Tracing

This pattern can be referred to as the test-driven development pattern for the development of Microservices. The pattern suggests taking a design-first approach where negotiation on the expected response between provider and consumer decides the best possible outcome. The developers of a consumer service write “contracts” specifying the responses they expect from the requests made to a service provider. It is “consumer-driven” because the consumer’s developers drive the writing of the contract and lead the negotiations with the provider’s developers.

The contract is typically a JSON or XML file that the developers and their services (consumer and provider) can access. One of the main benefits of consumer-driven contract testing is that it allows collaboration from the start between service providers and consumers. It also helps service developers to understand the requirements and expectations of consumers in a proactive manner.

Another benefit of CDCT is that it helps to prevent integration issues between services. By defining the contract up front, both the consumer and provider can ensure they are on the same page, saving many testing cycles as well.

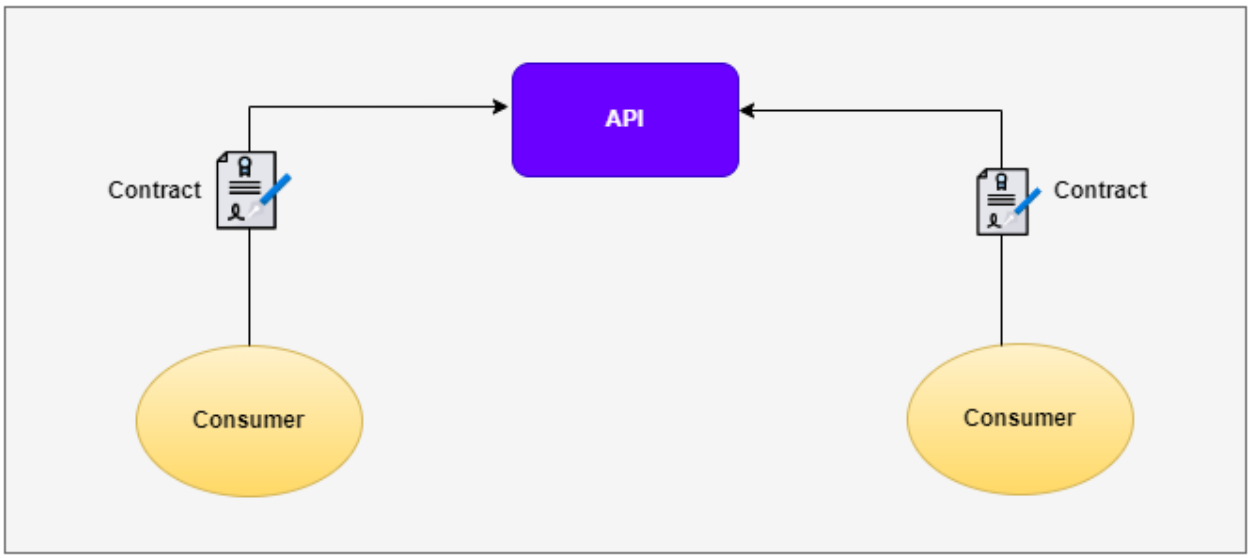
Here are the steps to implement CDCT:

1. **Define the contract**: The consumer of a service should define the contract that the service provider must follow. This should include the expected input and output of the service, as well as any other requirements that the consumer has.
2. **Implement the contract**: The service provider should implement the contract and ensure that their service meets the requirements.

3. **Run tests**: The service provider should then run test cases to ensure the service meets the requirements. This can include both unit tests and integration tests.

4. **Publish the contract:** Once the service provider has implemented the contract and passed their tests, they should publish the contract to the consumer.

5. **Verify compatibility**: Consumers can then verify that they got what was agreed upon.



Conclusion

With the increase in the usage of Microservices-based architecture, complexities arise in managing scalability or handling distributed services' transactions. However, a set of defined patterns, which have been tested repeatedly, give solutions to problems that are very common for Microservice-based architectures. Knowing each pattern provides good insight into how Microservices architecture handles performance, scalability, agility, and maintainability. We hope that the few patterns described above provide good insight for you.