[Microservices](https://codefresh.io/learn/microservices/) are a software development approach where a single application is composed of small, independent components that communicate with each other through well-defined interfaces.

This architecture contrasts with traditional monolithic designs, where different functionalities are tightly integrated into a single codebase.

Microservices design patterns are a set of methodologies that provide solutions to recurring design problems.

Think of them as templates (tested solutions) that can be used in the creation of microservices applications.

These patterns are particularly useful when developing complex applications with many microservices.

**The Need for Microservices Design Patterns**

Managing a microservices architecture involves a variety of complex challenges that are rarely encountered in traditional monolithic systems.

These challenges include:

* **Service orchestration:** Ensuring that multiple, independent services communicate seamlessly to execute **complex business processes**.
* **Service discoverability:** How services locate each other in a dynamically scaling environment.
* **Data consistency:** Unlike monolithic systems where you can rely on **ACID transactions** in a single database, microservices often have their own databases, making transactional consistency a big concern.
* **Fault tolerance:** In a distributed system, a failure in one service shouldn’t lead to a system-wide collapse.

To address these challenges systematically, developers can leverage microservices design patterns.

They guide development teams towards best practices in microservices development and provide structures that make it easier to create complex but stable systems.

## Top 10 Design Patterns in Microservices Architecture

There are numerous microservices design patterns that you can use, each with their unique advantages and use cases. Here are some of the more important patterns:

### 1. Service Registry

A service registry is like a map for your services; it keeps track of all the services in your system, making it easier for them to find each other.

Every service in your system needs to register itself with the service registry when it starts up and deregister when it shuts down.

Other services can then query the service registry to locate the services they need to interact with.

This allows your system to be dynamic and adaptable, as services can come and go as required without disrupting the overall functionality.

### 2. Circuit Breaker

A circuit breaker is used to detect failures and encapsulate the logic of preventing a failure from constantly recurring.

In a microservices architecture, the circuit breaker pattern to monitor the interaction between services.

If a service is failing or responding slowly, the circuit breaker trips and prevents further calls to the service to prevent a system-wide failure.

Once the service is back up, the circuit breaker resets, and things go back to normal.

### 3. API Gateway

An API gateway acts as a single-entry point into your system for all clients.

This can be especially beneficial if you have multiple client apps, such as a web app and a mobile app, as it allows you to maintain a single API for all clients, simplifying client-side code.

The API gateway can handle requests in one of two ways.

It could route requests to the appropriate services directly, or it could use a process known as **composition**, where it would combine data from multiple services and return the aggregate result to the client.

This not only simplifies client-side code but also makes your system more efficient and user-friendly.

### 4. Event-Driven Architecture

In an event-driven architecture, when a service performs an action that other services need to know about, it emits an event—a record of the action. Other services then react to the event as necessary.

This is a powerful way to decouple services and allows for **highly scalable and robust systems.**

This architecture allows you to build systems that are more resilient to failure, as the services do not need to be aware of each other. If one service fails, it does not affect the others.

**Additionally, this architecture allows for high scalability, as you can add new services to the system without affecting existing ones.**

### 5. Database per Service

In a traditional monolithic application, you would have a single database that all services interact with. However, in a microservices architecture, each service has its own database.

This is beneficial as it allows each service to be decoupled from the others, which means that a failure in one service does not affect the others.

Furthermore, it allows for better performance, as each service can be optimized independently based on its specific needs.

### 6. Command Query Responsibility Segregation (CQRS)

CQRS is a microservices design pattern that **separates read and write operations**.

In traditional systems, the **same data model** is often used for both these operations.

However, CQRS suggests for a different approach. It proposes the use of separate models for update (Command) and read (Query) operations.

This segregation enables you to **optimize each model for its specific purpose**, thereby improving **performance and scalability**.

However, implementing CQRS is has its challenges. It can complicate your system due to the need to synchronize two data models.

But, when applied correctly, it can significantly enhance the **flexibility and performance of your system.**

### 7. Externalized Configuration

The externalized configuration pattern advocates for the separation of configuration from the code.

This separation allows you to modify the behaviour of your application without the need for code changes or system restarts.

This pattern is particularly useful in microservices architectures where you may have **multiple instances of a service running with different configurations.**

By externalizing the configuration, you can manage all instances efficiently.

However, it requires a **robust configuration management system** to avoid **configuration drift**.

### 8. Saga Pattern

The saga pattern is used to ensure **data consistency** across multiple services in a microservices architecture.

In traditional **monolithic systems, transactions are usually managed using a two-phase commit**. However, in a microservices architecture, where services are loosely coupled and distributed, this approach is not practical.

The saga pattern proposes an alternative solution. **It suggests breaking a transaction into a sequence of multiple local transactions.**

Each local transaction updates data within a single service and publishes an event.

Other services listen to these events and perform their local transactions. If a local transaction fails, compensating transactions are executed to undo the changes.

### 9. Bulkhead Pattern

This pattern helps to prevent failures in one part of a system from cascading to other parts. **It does so by isolating elements of an application into pools so that if one fails, the others continue to function.**

This pattern is inspired by the bulkheads in a ship. Just as **a ship is divided into watertight compartments to prevent it from sinking if one part is breached**, **an application can be divided into isolated groups to protect it from failures.**

### 10. Backends for Frontends (BFF)

The BFF pattern proposes the creation of separate backend services for different types of clients (like desktop, mobile, etc.). This allows you to make the backend services to the specific needs of each client, thereby improving user experience and performance.

However, this pattern can lead to code duplication if not managed properly. Therefore, it is crucial to strike a balance between customization and code reuse when using the BFF pattern.

**How to Choose Design Patterns in Microservices**

Choosing the right design patterns for your microservices architecture is critical for building **a robust and scalable system**.

Here are a few factors to consider while making your choice.

* **Assess service interdependencies:**

If your services are highly interdependent, a transactional pattern like the saga pattern might be beneficial. On the other hand, if your services are more isolated, patterns like bulkhead can help prevent cascading failures.

* **Evaluate resilience requirements:**

Different patterns offer different levels of resilience. For instance, the bulkhead pattern can improve resilience by preventing failures from cascading. Evaluate how crucial resilience is for your system and choose patterns accordingly.

* **Consider security implications:**

Some patterns may have security implications. For example, the externalized configuration pattern requires **a secure configuration management system** to protect sensitive configuration information.

* **Reusability and future-proofing:**

Design patterns like backends for frontends can enhance reusability by allowing you to make backends for different clients.

Meanwhile, patterns like **CQRS can future-proof your system** by providing flexibility and scalability.