**Micro Services**

**Release Number : 0.01**

**Document Purpose**

This document should contain all the information necessary to understand and use Microservices Architecture and know the benefits-drawbacks and implementations

**Document Parameters**

|  |  |
| --- | --- |
| Template version | 0.2 |
| **Document version** |  |
| **Creation date** | *22-02-2021* |
| **Date last update** | *26-02-2021* |
| **Author** | Claes John |

**Table of content**

[1 Software Architecture 5](#_Toc65241062)

[1.1 Modular Monolithic Architecture 5](#_Toc65241063)

[1.2 Modular Monolithic Software Architecture 5](#_Toc65241064)

[1.3 Microservice Architecture 6](#_Toc65241065)

[2 Lets dive into Micro Services ? 7](#_Toc65241066)

[2.1 Important principles behind MicroServices 7](#_Toc65241067)

[2.2 Important Characteristics of Microservice Architecture: 8](#_Toc65241068)

[2.3 Advantages of Microservice Architecture: 8](#_Toc65241069)

[2.4 Disadvantages of Microservice Architecture: 8](#_Toc65241070)

[2.5 When to use Microservice Architecture: 8](#_Toc65241071)

[3 Decomposition Patterns 9](#_Toc65241072)

[3.1 Decompose by Business Capability 9](#_Toc65241073)

[3.1.1 Problem 9](#_Toc65241074)

[3.1.2 Solution 9](#_Toc65241075)

[3.2 Decompose by Subdomain 9](#_Toc65241076)

[3.2.1 Problem 9](#_Toc65241077)

[3.2.2 Solution 9](#_Toc65241078)

[3.3 Strangler Pattern 9](#_Toc65241079)

[3.3.1 Problem 9](#_Toc65241080)

[3.3.2 Solution 10](#_Toc65241081)

[3.4 Volality Pattern( Luc ) 10](#_Toc65241082)

[4 Design Patterns 10](#_Toc65241083)

[4.1 Database per Microservice 10](#_Toc65241084)

[4.1.1 Pros 10](#_Toc65241085)

[4.1.2 Cons 10](#_Toc65241086)

[4.1.3 When to use Database per Microservice 10](#_Toc65241087)

[4.1.4 When not to use Database per Microservice 10](#_Toc65241088)

[4.2 Saga 10](#_Toc65241089)

[4.2.1 Pros 11](#_Toc65241090)

[4.2.2 Cons 11](#_Toc65241091)

[4.2.3 When to use Saga 11](#_Toc65241092)

[4.2.4 When not to use Saga 11](#_Toc65241093)

[4.3 API Gateway 11](#_Toc65241094)

[4.3.1 Pros 12](#_Toc65241095)

[4.3.2 Cons 12](#_Toc65241096)

[4.3.3 When to use API Gateway 13](#_Toc65241097)

[4.3.4 When not to use API Gateway 13](#_Toc65241098)

[4.4 Event Sourcing 13](#_Toc65241099)

[4.4.1 Pros 13](#_Toc65241100)

[4.4.2 Cons 13](#_Toc65241101)

[4.4.3 When to use Event Sourcing 13](#_Toc65241102)

[4.4.4 When not to use Event Sourcing 13](#_Toc65241103)

[4.5 Command Query Responsibility Segregation (CQRS) 14](#_Toc65241104)

[4.5.1 Pros 14](#_Toc65241105)

[4.5.2 Cons 14](#_Toc65241106)

[4.5.3 When to use CQRS 14](#_Toc65241107)

[4.5.4 When not to use CQRS 14](#_Toc65241108)

[4.5.5 Enabling Technology Examples 14](#_Toc65241109)

[5 Conclusion 15](#_Toc65241110)

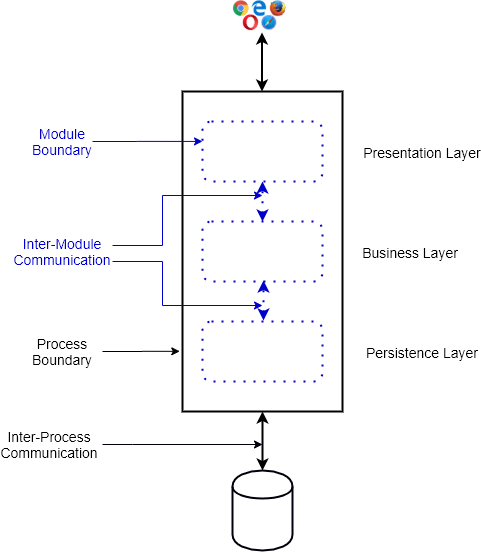
[6 Sources 15](#_Toc65241111)

# Software Architecture

## Modular Monolithic Architecture

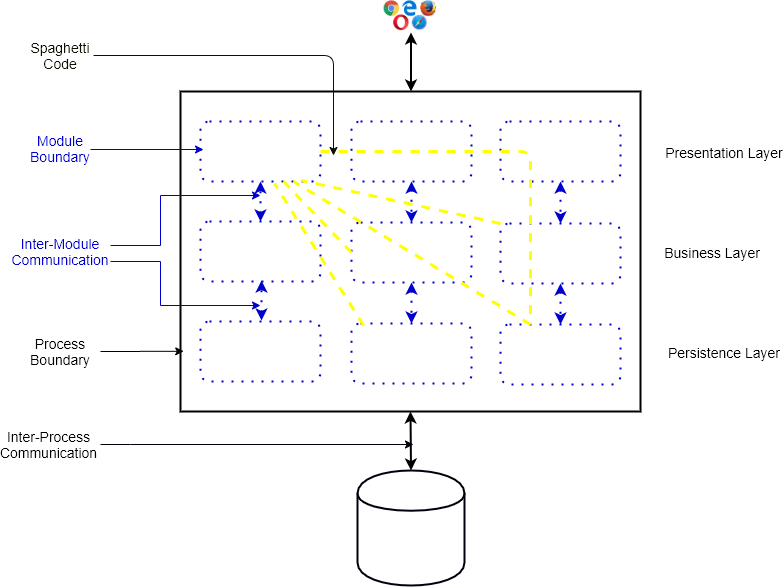
Software Systems were developed as a single system and deployed as a single process

The whole appliance is split into multiple layers (Presentation, Business, Persistence), and the entire application is deployed on an Application Server/Web Server.



## ****Modular Monolithic Software Architecture****

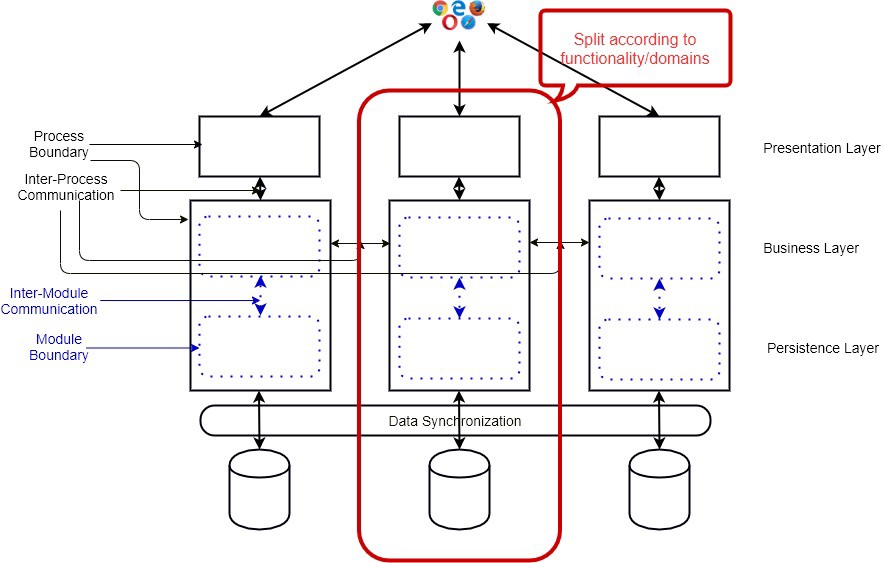
Software Engineers tackled the complexity by decomposing the whole systems into “**Loosely coupled, highly cohesive**” modules.



* The complete Software System is deployed as a whole (all or Nothing)
* The Modular boundary is internal and can be crossed easily which can lead to Spaghetti Code (as shown above by yellow lines)
* The application runs as one single process
* It is one size for all, i.e., one solution for all sizes of application
* No strict data ownership among modules

## Microservice Architecture

In the 2010s, Web-Scale companies found that for extremely large applications, Modular Monolithic Software architecture is not suitable and created **Microservice Software Architecture**.



* The whole application is **split into separate processes where each process can contain multiple modules**.
* Contrary to Modular Monoliths or SOA, a Microservice application is split **vertically**(according to functionality or domains)

# Lets dive into Micro Services ?

Microservice Architecture is about splitting a large, complex systems vertically (per functional or business requirements) into smaller sub-systems which are processes (hence independently deployable) and these sub-systems communicates with each other via lightweight, language-agnostic network calls either synchronous (e.g. REST, gRPC) or asynchronous (via Messaging) way.

## Important principles behind MicroServices

* Scalability
* Availability
* Resiliency
* Independent, autonomous
* Decentralized governance
* Failure isolation
* Auto-Provisioning
* Continuous delivery through DevOps

## Important Characteristics of Microservice Architecture:

* The whole application is split into separate processes where each process can contain multiple internal modules.
* Contrary to Modular Monoliths or SOA, a Microservice application is split vertically(according to business capability or domains)
* The Microservice boundary is external. As a result, Microservices communicates with each other via network calls (RPC or message).
* As Microservices are independent processes, they can be deployed independently.
* They communicate in a lightweight way and don’t need any smart Communication channel.

## Advantages of Microservice Architecture:

* Better development scaling.
* Higher development velocity.
* Supports iterative or incremental modernization.
* Take advantage of the modern Software Development Ecosystem (Cloud, Containers, DevOps, Serverless).
* Supports horizontal scaling and granular scaling.
* It puts low cognitive complexity on the developer’s head thanks to its smaller size.

## Disadvantages of Microservice Architecture:

* A higher number of Moving parts (Services, Databases, Processes, Containers, Frameworks).
* Complexity moves from Code to the Infrastructure.
* The proliferation of RPC calls and network traffic.
* Managing the security of the complete system is challenging.
* Designing the entire system is harder.
* Introduce complexities of Distributed Systems.

## When to use Microservice Architecture:

* Web-Scale Application development.
* Enterprise Application development when multiple teams work on the application.
* Long-term gain is preferred over short-term gain.
* The team has Software Architects or Senior Engineers capable of designing Microservice Architecture.

# Decomposition Patterns

## Decompose by Business Capability

### ****Problem****

Microservices is all about making services loosely coupled, applying the single responsibility principle. However, breaking an application into smaller pieces has to be done logically. How do we decompose an application into small services?

### ****Solution****

One strategy is to decompose by business capability. A business capability is something that a business does in order to generate value. The set of capabilities for a given business depend on the type of business. For example, the capabilities of an insurance company typically include sales, marketing, underwriting, claims processing, billing, compliance, etc. Each business capability can be thought of as a service, except it’s business-oriented rather than technical.

## Decompose by Subdomain

### ****Problem****

Decomposing an application using business capabilities might be a good start, but you will come across so-called "God Classes" which will not be easy to decompose. These classes will be common among multiple services. For example, the Order class will be used in Order Management, Order Taking, Order Delivery, etc. How do we decompose them?

### ****Solution****

For the "God Classes" issue, DDD (Domain-Driven Design) comes to the rescue. It uses subdomains and bounded context concepts to solve this problem. DDD breaks the whole domain model created for the enterprise into subdomains. Each subdomain will have a model, and the scope of that model will be called the bounded context. Each microservice will be developed around the bounded context.

**Note**: Identifying subdomains is not an easy task. It requires an understanding of the business. Like business capabilities, subdomains are identified by analyzing the business and its organizational structure and identifying the different areas of expertise.

## Strangler Pattern

### ****Problem****

So far, the design patterns we talked about were decomposing applications for greenfield, but 80% of the work we do is with brownfield applications, which are big, monolithic applications. Applying all the above design patterns to them will be difficult because breaking them into smaller pieces at the same time it's being used live is a big task.

### ****Solution****

The Strangler pattern comes to the rescue. The Strangler pattern is based on an analogy to a vine that strangles a tree that it’s wrapped around. This solution works well with web applications, where a call goes back and forth, and for each URI call, a service can be broken into different domains and hosted as separate services. The idea is to do it one domain at a time. This creates two separate applications that live side by side in the same URI space. Eventually, the newly refactored application “strangles” or replaces the original application until finally you can shut off the monolithic application.

## Volality Pattern( Luc )

# Design Patterns

## Database per Microservice

### Pros

* Complete ownership of Data to a Service.
* Data Access Technology - Data Storage can be selected :   
  ---- Please consult Enterprise Architect for Enterprise
* Loose coupling among teams developing the services.
* Limited inter Team Dependencies ( As low as possible )

### ****Cons****

* Sharing data among services becomes challenging.
* Giving application-wide ACID transactional guarantee becomes a lot harder.
* Decomposing the Monolith database to smaller parts need careful design and is a challenging task.

### ****When to use Database per Microservice****

* In large-scale enterprise applications.
* When the team needs complete ownership of their Microservices for development scaling and development velocity.

### ****When not to use Database per Microservice****

* In small-scale applications.
* If one team develops all the Microservices.

## Saga

There are mainly two variations of Saga transactions co-ordinations:

* *Choreography*: Decentralised co-ordinations where each Microservice produces and listen to other Microservice’s events/messages and decides if an action should be taken or not.  
    
  
* *Orchestration*: Centralised co-ordinations where an Orchestrator tells the participating Microservices which local transaction needs to be executed.  
    
  

### Pros

* Provide consistency via transactions in a highly scalable or loosely coupled, event-driven Microservice Architecture.
* Provide consistency via transactions in Microservice Architecture where NoSQL databases without 2PC support are used.

### Cons

* Need to handle transient failures and should provide idempotency.
* Hard to debug, and the complexity grows as the number of Microservices increase.

### When to use Saga

* In highly scalable, loosely coupled Microservice Architecture where event sourcing is used.
* In systems where distributed NoSQL databases are used.

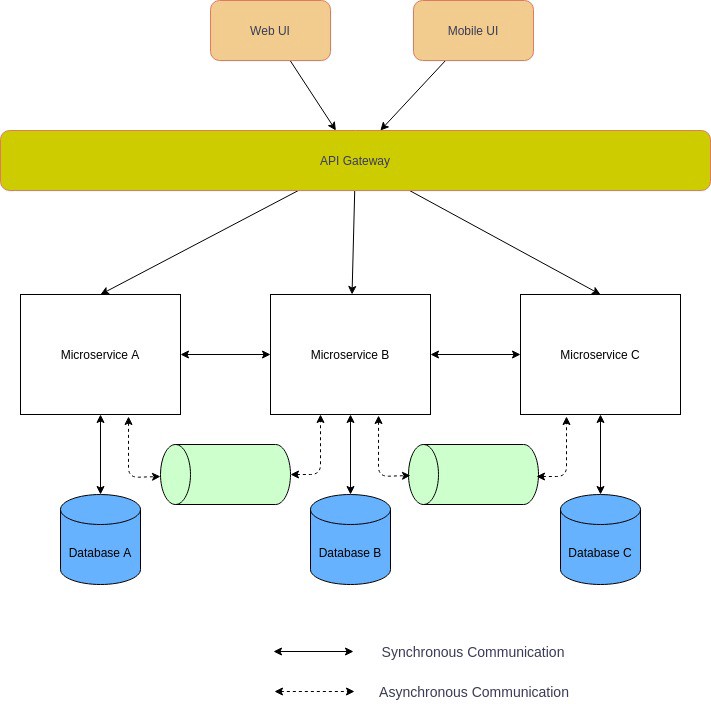
### When not to use Saga

* Lowly scalable transactional systems with SQL Databases.
* In systems where cyclic dependency exists among services.

## API Gateway

In Microservice Architecture, the UI usually connects with multiple Microservices. If the Microservices are finely grained (FaaS), the Client may need to connect with lots of Microservices, which becomes chatty and challenging. Also, the services, including their APIs, can evolve. Large enterprises will like to have other cross-cutting concerns (SSL termination, authentication, authorization, throttling, logging, etc.).

One possible way to solve these issues is to use API Gateway. API Gateway sits between the Client APP and the Backend Microservices and acts as a facade. It can work as a reverse proxy, routing the Client request to the appropriate Backend Microservice. It can also support the client request's fanning-out to multiple Microservices and then return the aggregated responses to the Client. It additionally supports essential cross-cutting concerns.



### Pros

* Offer loose coupling between Frontend and backend Microservices.
* Reduce the number of round trip calls between Client and Microservices.
* High security via SSL termination, Authentication, and Authorization.
* Centrally managed cross-cutting concerns, e.g., Logging and Monitoring, Throttling, Load balancing.

### Cons

* Can lead to a single point of failure in Microservice Architecture.
* Increased latency due to the extra network call.
* If it is not scaled, they can easily become the bottleneck to the whole Enterprise.
* Additional maintenance and development cost.

### When to use API Gateway

* In complex Microservice Architecture, it is almost mandatory.
* In large Corporations, API Gateway is compulsory to centralize security and cross-cutting concerns.

### When not to use API Gateway

* In private projects or small companies where security and central management is not the highest priority.
* If the number of Microservices is fairly small.

## Event Sourcing

In a Microservice Architecture, especially with **Database per Microservice,** the Microservices need to exchange data. For resilient, highly scalable, and fault-tolerant systems, they should communicate asynchronously by exchanging Events. In such a case, you may want to have Atomic operations, e.g., update the Database and send the message. If you have SQL databases and want to have distributed transactions for a high volume of data, you cannot use the [two-phase locking](https://en.wikipedia.org/wiki/Two-phase_locking) (2PL) as it does not scale. If you use NoSQL Databases and want to have a distributed transaction, you cannot use 2PL as many NoSQL databases do not support two-phase locking.

### ****Pros****

* Provide atomicity to highly scalable systems.
* Automatic history of the entities, including time travel functionality.
* Loosely coupled and event-driven Microservices.

### ****Cons****

* Reading entities from the Event store becomes challenging and usually need an additional data store (**CQRS** pattern)
* The overall complexity of the system increases and usually need [Domain-Driven Design](https://en.wikipedia.org/wiki/Domain-driven_design).
* The system needs to handle duplicate events (idempotent) or missing events.
* Migrating the Schema of events becomes challenging.

### ****When to use Event Sourcing****

* Highly scalable transactional systems with SQL Databases.
* Transactional systems with NoSQL Databases.
* Highly scalable and resilient Microservice Architecture.
* Typical Message Driven or Event-Driven systems (e-commerce, booking, and reservation systems).

### ****When not to use Event Sourcing****

* Lowly scalable transactional systems with SQL Databases.
* In simple Microservice Architecture where Microservices can exchange data synchronously (e.g., via API).

## Command Query Responsibility Segregation (CQRS)

If we use Event Sourcing, then reading data from the Event Store becomes challenging. To fetch an entity from the Data store, we need to process all the entity events. Also, sometimes we have different consistency and throughput requirements for reading and write operations.

In such use cases, we can use the CQRS pattern. In the CQRS pattern, the system's data modification part (Command) is separated from the data read (Query) part. CQRS pattern has two forms: simple and advanced, which lead to some confusion among the software engineers.

In its simple form, distinct entity or ORM models are used for Reading and Write, as shown below:

### ****Pros****

* Faster reading of data in Event-driven Microservices.
* High availability of the data.
* Read and write systems can scale independently.

### ****Cons****

* Read data store is weakly consistent (eventual consistency)
* The overall complexity of the system increases. Cargo culting CQRS can significantly jeopardize the complete project.

### ****When to use CQRS****

* In highly scalable Microservice Architecture where event sourcing is used.
* In a complex domain model where reading data needs query into multiple Data Store.
* In systems where read and write operations have a different load.

### ****When not to use CQRS****

* In Microservice Architecture, where the volume of events is insignificant, taking the Event Store snapshot to compute the Entity state is a better choice.
* In systems where read and write operations have a similar load.

### ****Enabling Technology Examples****

Write Store: [EventStoreDB](https://www.eventstore.com/), [Apache Kafka](https://kafka.apache.org/), [Confluent Cloud](https://www.confluent.io/confluent-cloud), [AWS Kinesis](https://aws.amazon.com/kinesis/), [Azure Event Hub](https://azure.microsoft.com/en-us/services/event-hubs/), [GCP Pub/Sub](https://cloud.google.com/pubsub), [Azure Cosmos DB](https://docs.microsoft.com/en-us/azure/cosmos-db/introduction), [MongoDB](https://www.mongodb.com/), [Cassandra](https://cassandra.apache.org/). [Amazon DynamoDB](https://aws.amazon.com/dynamodb/)

Read Store: [Elastic Search](https://www.elastic.co/), [Solr](https://lucene.apache.org/solr/features.html), [Cloud Spanner](https://cloud.google.com/spanner), [Amazon Aurora](https://aws.amazon.com/rds/aurora/), [Azure Cosmos DB](https://docs.microsoft.com/en-us/azure/cosmos-db/introduction), [Neo4j](https://neo4j.com/)

Frameworks: [Lagom](https://www.lagomframework.com/), [Akka](https://akka.io/), [Spring](https://spring.io/), [akkatecture](https://akkatecture.net/), [Axon](https://axoniq.io/), [Eventuate](https://eventuate.io/)

# Conclusion

In the modern large-scale enterprise Software development, Microservice Architecture can help development scaling with many long-term benefits. But Microservice Architecture is no Silver Bullet that can be used in every use case. If it is used in the wrong type of application, Microservice Architecture can give more pains as gains. The development team that wants to adopt Microservice Architecture should follow a set of best practices and use a set of reusable, battle-hardened design patterns.

# Sources

As We all do when we try to get the information we need, we use online Sites and information and we use / adapt what we want and how we want.  
In my humble opinion I’ve used the source that in my opinion is closest to what I usually use

|  |
| --- |
| Sources |
| [Looking Beyond the Hype: Is Modular Monolithic Software Architecture Really Dead?](https://towardsdatascience.com/looking-beyond-the-hype-is-modular-monolithic-software-architecture-really-dead-e386191610f8) |
| [Microservice Architecture and its 10 Most Important Design Patterns](https://towardsdatascience.com/microservice-architecture-and-its-10-most-important-design-patterns-824952d7fa41) |
| [Pattern: Microservice Architecture](https://microservices.io/patterns/microservices.html) |
| [Design Patterns for Microservices](https://dzone.com/articles/design-patterns-for-microservices) |