TURNING MONOLITHS INTO MICROSERVICES: TIPS AND TRICKS

Microservices | Software as a Service (SaaS)

beSharp | 7 June 2019

In the first article of our 3 part journey on how to break down a Monolithic application we have started talking about the advantages of a distributed application composed of microservices over a monolithic one. In this second part, we will start approaching techniques and tips to help you overcome the migration process in a more secure and aware way

CLEARLY DEFINE YOUR BUSINESS LOGIC'S DOMAIN

This is by far the most important step as it states clearly if **your knowledge** of your logic's domain is complete. In fact, one possible way to find which part of your application can be converted to a microservice is to check if you have a complete understanding of its domain, which means that it will be easier for your dev team to **isolate and migrate** the logic.

Understanding the domain of a feature also means that, probably, it's **entanglement** with the rest of the application is **very low**, ideally null, thus facilitating the separation from the main application.

Knowing your domain means that you can **define with clarity** which services are **vertical to the application**, or in other words, which services are most important and **target specific needs** of a user base (thus defining the goal of your application) or are strategic for your company.

Vertical services are by no means made of easy code and can be very big, but as said before, we are not referring to microservices by the quantity of code contained, but by **definition of scopes** and **context boundaries**.

DEFINE YOUR TECHNOLOGICAL TARGET

Come to this point is important to clearly define our **technological target**, in the sense that we want to understand what kind of **platform** or **framework** we want to use for development and procedures of CD/CI (e.g Serverless Framework and AWS Lambda as a development environment) and what kind of programming language is best suited for migrating that precise part of your business logic.

Having clear this part is very crucial as it helps thinking about the **feasibility of that particular migration**. Also, we have to understand that, even if migrating to microservices is, in general, a good behavior, it doesn't mean that for your particular application, or part of it, it's a good choice, but we will analyze this aspect later.

In general, during the brainstorming involved in this step, we need to verify the costs in terms of **code migration** and **infrastructure migration** and the **strategic value** compared to **developing new features**.

WHERE TO START TO BREAK A MONOLITH: THE STRANGLER PATTERN

Up to this point we have talked about preparations, let's start to explain how to effectively break our monolithic infrastructure and what characteristics a microservice must have to be considered one once it has been extracted successfully.

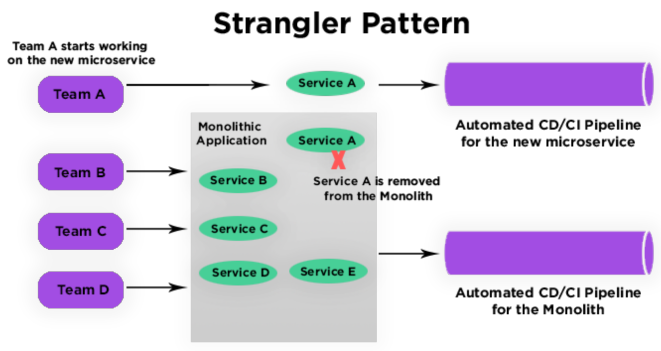
When we think about going from a monolithic approach to a microservice ecosystem we can think of two possible choices: a) to rewrite your code from scratch using the new paradigm or b) to migrate from the old one.

Starting fresh on rewriting the entire application in one go is generally not a good choice because:

- It consumes a lot of time.
- It can bring **technical debt** along during the rewriting process.
- It can't be used until all the rewriting is complete.
- We can't really focus on new features as they need the application logic to be entirely ported as well causing delays for the business team.

So the best option is, as we have in a way discussed up to this point, to extract and convert the application **step by step**.

This approach is called **Strangler Pattern**, a way to **incrementally transform your monolithic application into microservices by replacing functionalities one at a time**. Once the new functionality is coded, the old component is **strangled**, substituted and finally **decommissioned** altogether.

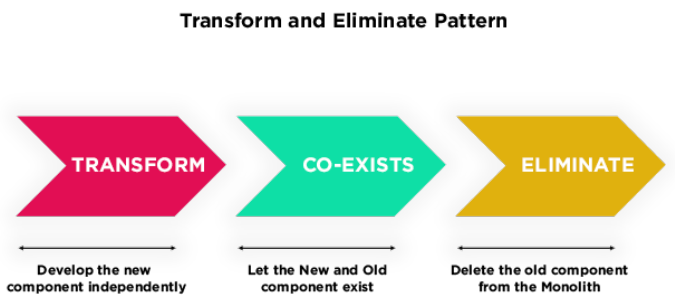

Strangler Pattern
Team A starts working on the new microservice
Team A
Service A
Automated CD/CI Pipeline for the new microservice
Monolithic Application
Service A
Team B
Service B
Service A is removed from the Monolith
Team C
Service C
Team D
Service D
Service E
Automated CD/CI Pipeline for the Monolith

This approach is very good for a number of reasons:

- It requires a **flexible time effort** that can be adapted to the current development situation of your teams.
- The application can still be used because you are migrating only a small part of it at a time.
- You can continue developing new features making them as new microservices.

To implement the Strangler Pattern, you can follow three steps: **Transform**, **Coexist** (coexistence is necessary for testing the new feature with part of the user base), and **Eliminate**.

Let's just spend two words on the co-exist part because it requires a little thinking as it requires your team to maintain both codebases and the relative user support for some time until the old component is put out of order, so always plan ahead to avoid bad surprises.


Transform and Eliminate Pattern
TRANSFORM
CO-EXISTS
ELIMINATE
Develop the new component independently
Let the New and Old component exist
Delete the old component from the Monolith

To start extracting elements of your business logic, especially if you are new to this pattern, choose those that have:

- A clear domain, a precise scope
- No data stored or data that can easily live on its own data source
- No sticky coupling with the rest of your application code

Also:

- If there is a component that has good test coverage and less technical debt associated with it, this is a good candidate (if the domain's logic is clear enough).
- If a component has scalability requirements, go for it.
- If there is a component that has frequent business requirements and needs to be deployed a lot more regularly, you can start with that component.

Finally check for part of code that can be easily reused in other projects, which means that it is by its own nature **atomic** and **reusable.**

There are tools that can help in the splitting decision, such as those for **Social code analysis** which enriches our understanding of the code quality by overlaying a developer's behavior with the **structural analysis of the code**.

It uses data from **version control systems**. Such a tool is **CodeScene.**

WHAT MAKES A MICROSERVICE A MICROSERVICE?

In order to understand if you are creating a true microservice, you have to check if it adheres to specific **properties of isolation** and they are at **infrastructural level** and at the **logic level**.

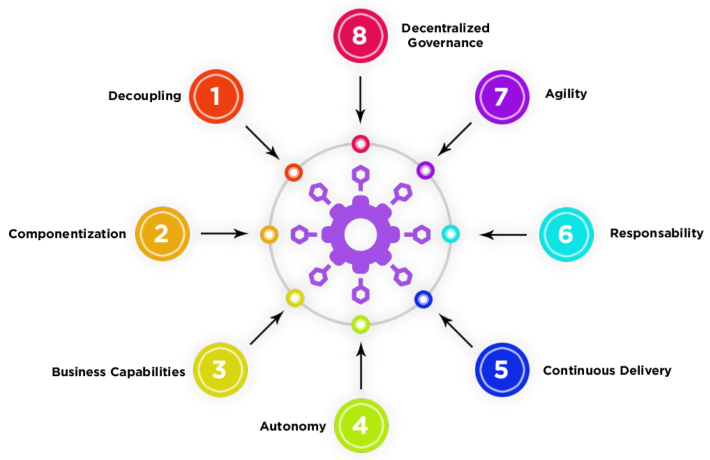
A microservice must be **independent** and **atomic** (can exist and give a service on its own, possibly without knowing anything about the entire application) at the **building**, **testing**, **deploying**, **monitoring**, **debugging** and **recovering** level. This ensures that we can safely build, manage and deploy it **without disrupting** the **life cycle** of the main application.

A microservice must also be independent at **API**, **Business Logic** and **Data level**, to ensure **Code Isolation** (access to the service functionalities is done through APIs).

This means that a good principle is also to prepare and define an isolated database layer for each microservice instead of a shared data source.

Even if this operation means you have to define some **specific descriptors** that must be used to make the different microservices speech to each other in terms of data, we avoid a possible **single point of failure** of having a single database (a malfunction to this can compromise the entire set of microservices instead of a single one).

As a golden rule always verify that **no microservice depends on the monolith**.



In this second part of our 3-article journey on how to break down a Monolithic application we have started to deep dive in some well-known techniques and ideas that helps dev and business team to decide when and how we can split legacy code in atomic and independent microservices, how we can approach the migration life-cycle and in general what properties a microservice must have to be considered one. In the next article we will conclude our journey describing **how to effectively start coding your microservices,** how to approach complex code with high intellectual value and also when switching to a microservices-ecosystem may not be a good choice.

Stay tuned!

beSharp

Dal 2011 beSharp guida le aziende italiane sul Cloud. Dalla piccola impresa alla grande multinazionale, dal manifatturiero al terziario avanzato, aiutiamo le realtà più all'avanguardia a realizzare progetti innovativi in campo IT.

Get in touch

beSharp.it
proud2becloud@besharp.it