WRITING MICROSERVICES, THE RIGHT WAY.

Microservices | Software as a Service (SaaS)

beSharp | 29 November 2019

In this final article of our 3 part analysis (you can read here the first part and the second part) on how to break down a Monolithic application we will exploit some insights on how a microservice can be created from an existing one, what can we do to solve problems related to legacy code with high toxicity and in general what rules can be applied to decide when to migrate to microservices.

Let's get started!

HOW TO WRITE A MICROSERVICE

In practice when we want to create a microservice we have two development choices:

1. **Code porting**
2. **Writing new code**.

CODE PORTING

At first Code porting may seems a logical solution, mainly because the Dev team has a **cognitive bias** towards its own code BUT it can conceal some critical aspects that have **high cost** and **low value**:

- There is a **large amount of boilerplate code that deals with environmental dependencies**, such as accessing application configuration at runtime, accessing data stores, caching, and is built with old frameworks. This can be safely rewritten because the microservice will surely approach things differently in a different environment.
- Existing capabilities may be built around uncertain domain concepts. This results in undergoing a big restructuring.
- An old legacy code that has gone through many iterations of change could have high **code toxicity** and low value for reuse.

So to sum it up always verify if the code you want to reuse has **High Intellectual complexity value** and **Low Toxicity** in terms of stickiness and technical debt**: this is a good candidate** for code extraction and reuse. An example can be a complex recommendation system for a customer with a lot of interactions, personalizations, machine learning algorithms and so on.

WRITING NEW CODE

Writing new code is a very good approach in the sense that enforce the Dev team and the Business team to review the core logic of the functionality they want to rewrite and thus:

1. Approach it with a clear understanding of the domain and with a lot more experience, resulting most probably in a smaller tidier solution.
2. Search for a **technology refresh**, implementing the new service with a programming language and technology stack that is more modern and surely more suitable.

This approach can also be beneficial in the sense that the microservice will probably **be delivered faster**. An example can be **CRUD operations** because they don't contain any intellectual properties and may depend on new technologies and frameworks.

HOW TO APPROACH STICKY CODE INSIDE THE MONOLITH

After rewriting most of your application with microservices, you'll be faced with the remaining of the legacy code for which the domain knowledge is uncertain. In this case, extracting your code and also your **data from the data layer** can be difficult although we would recommend approaching this part of your infrastructure as fast as possible because decoupling your entire data layer is by far the most important part of your job in migrating towards a microservice-ecosystem.

We can safely say that if your data is somehow coupled you don't have a real microservice.

A technique that can help you in this process is called **Reification,** which is a process to redefine context boundaries through the decomposition of uncertain part of logical domain by deconstructing legacy code in well separated and simpler structures which infer from more defined data models as well.

You can also use **dependency and structural code analysis** tools such as **Structure101** to identify the most coupling and constraining factor capabilities in the monolith.

HOW AWS CAN HELP WITH ITS SERVICES

AWS can really help in giving you the instruments for an easy transition to a microservice approach, at least from an infrastructural point of view.

BUILDING, TESTING AND DEPLOYING LEVEL:

CodeCommit, CodeBuild, and CodeDeploy.

MONITORING AND DEBUGGING:

CloudWatch, ElasticSearch with Kibana integration, X-Ray.

DEPLOY AND RECOVERY:

CloudFormation.

LOGIC:

AWS Lambda Functions and Layers, ECS and AWS Step Functions that coordinate workers.

WHY YOU WOULD AVOID MICROSERVICE APPROACH INSTEAD

We talked a lot about how good microservices are and how many benefits you would have by implementing this paradigm but there are some situations in which you may reconsider the idea of switching to microservices.

For example, if you have a small project that will probably not evolve in the future the effort of decoupling it may not be worth as making microservices is, as we have seen, not a simple task at all.

If the application serves a **mission-critical function**, such as **maintaining an irreplaceable legacy database**, you are maybe not likely to be able to replace it entirely in a few years, in other words, your strategic team can't afford **a long-term strategy**.

Some applications by **their nature** require **tight integration** between **individual components and services**. This is often true, for example, for applications that **process rapid streams of real-time data**. Any added layers of communication between services **may slow real-time processing down**.

CONCLUSIONS

To sum up what we have said up to this point let us reassume the most important key points of our discussion:

- Understand your technological target as clearly as possible to make your transition as fitting as possible
- Make your transitioning steps as **atomic** as possible and always remember the **three steps approach** in the **Strangler** Technique, especially to **decommission** old codebase once it is not used anymore.
- **Define which microservices to implement as first** to start working with the right attitude and to see the results of your work as soon as possible. Also, this approach is very good to start **developing experience in the migration process** before attacking more difficult parts of your codebase
- Use the **Reify** Technique extensively to help you redefine your most obscure code parts

- Always **strive to isolate meaningful parts of your data** in order to split it in per-microservice database (if the microservice needs it, it can be stateless). This step is very crucial.
- Always check for the **microservice to live up to its atomic property of isolation** both at infrastructure and logic level.
- Don't be afraid to make **extensive use of tools such as Structure101 or in general of AWS services** to help you in the migration process.
- Try to **rewrite your microservices from scratch as much as possible** (apart from code with **high intellectual value**), to involve your strategic team more and to define a better approach to solve a problem. Rewriting your code also avoid bringing along eventual technical debt.
- Remember that in order to start your microservices migration journey your DevTeam and in general your company must adhere to the DevOps Culture and must be in possession of some prerequisite experience.
- Finally, remember that there are some situations in which trying to migrate to a microservice approach may not be the ideal solution for cases like real-time applications in which communication of layers must be as fast as possible.

With this summary we end our long journey on how we can deconstruct a big monolithic application in several microservices in order to maintain it better, develop, deploy and eventually recovery it very fast, thus focusing much more on your business value. We hope you enjoyed reading this far.

Satisfied? Feel free to contact us, we will be happy to have a chat!

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