Microservices Architecture on Energy4Life Monolithic Application

Friday 7th May, 2021 - 00:31

Kevin L. Biewesch
University of Luxembourg
Email: kevin.biewesch.001@student.uni.lu

Alfredo Capozucca
University of Luxembourg
Email: alfredo.capozucca@uni.lu

Abstract

This document is a template for the scientific and technical (S&T for short) report that is to be delivered by any BiCS student at the end of each Bachelor Semester Project (BSP). The Latex source files are available at: ***https://github.com/nicolasguelfi/lu.uni.course.bics.global***

This template is to be used using the Latex document preparation system or using any document preparation system. The whole document should be in between 6000 to 8000 words [1] *(excluding the annexes) and the proportions must be preserved. The other documents to be delivered (summaries, ...) should have their format adapted from this template.*

1. Introduction (± 5% of total words)

This paper presents the bachelor semester project made by Motivated Student together with Motivated Tutor as his motivated tutor. It presents the scientific and technical dimensions of the work done. All the words written here have been newly created by the authors and if some sequence of words or any graphic information created by others are included then it is explicitly indicated the original reference to the work reused.

This report separates explicitly the scientific work from the technical one. In deed each BSP must cover those two dimensions with a constrained balance (cf. [BiCS(2018b)]). Thus it is up to the Motivated Tutor and Motivated Student to ensure that the deliverables belonging to each dimension are clearly stated. As an example, a project whose title would be "A multi-user game for multi-touch devices" could define as scientific [Armstrong and Green(2017)] deliverables the following ones:

- Study of concurrency models and their implementation
- Study of ergonomics in human-computer interaction

The length of the report should be from 6000 to 8000 words excluding images and annexes. The sections presenting the technical and scientific deliverables represent ± 80% of total words of the report.

1. i.e. approximately 12 to 16 pages double columns

2. Project description (± 10% of total words)

2.1. Domains

2.1.1. Scientific . Provide a description of the scientific domain(s) in which the project is being made.

2.1.2. Technical. Provide a description of the technical domain(s) in which the project is being made.

2.2. Targeted Deliverables

2.2.1. Scientific deliverables. Provide a synthetic and abstract description of the scientific deliverables that were targeted to be produced. Each BSP must contain some work done according to the principles of the scientific method. It basically means that you should define at least one question related to the knowledge domain of your BSP and follow part of the scientific method process to answer to this question. The description of the work done to answer this question is a scientific deliverable.

Examples of question could be:

- Is Python an adequate language for concurrent programs?
- How can we measure the ergonomic of a graphical user interface?
- How can we ensure that a program will not fail?

An answer to such question should be the result of applying partly or totally the scientific method according to its standard definition which can be found in the literature.

As you can see in this template, the scientific deliverable is entirely separated from the technical deliverable. Of course it addresses a question more or less closely related to the technical deliverable.

2.2.2. Technical deliverables. Provide a synthetic and abstract description of the technical deliverables that were targeted to be produced.

3. Pre-requisites ([5%..10%] of total words)

Describe in these sections the main scientific and technical knowledge that is required to be known by you before starting the project. Do not describe in details this knowledge but only abstractly. All the content of this section shall not used, even partly, in the deliverable sections.

3.1. Scientific pre-requisites

3.2. Technical pre-requisites

4. Scientific Deliverable 1 – What is a Microservice?

For each scientific deliverable targeted in section 2 2 provide a full section with all the subsections described below.

4.1. Requirements (± 15% of section's words)

Since Microservices (MSs) were a previously unknown concept to me, this scientific deliverable targets the familiarisation of this architectural style. In the design section we are going to elaborate on motivations behind the architecture. In the production we will explore what constitutes a MS and the assessment will highlight a few drawbacks of the MS architecture.

4.2. Design (± 30% of section's words)

To understand the benefits and motivations behind using MS over the monolithic style, we will first look at the latter. [1]

Monolithic architectures build applications using a single unit. This basically means that all of an application's functionality, logic, classes, function and name spaces are in the end located within a single executable and thus run within a single process. This model is very much usable in fact and allows us to create functioning systems—as has demonstrated the time of software development before MSs were introduced. However, this architectural style raises some issues nonetheless. For instance, even the smallest of changes in an application require the whole product to be rebuilt and redeployed or redistributed which is not necessarily an ideal scenario.

As the application's life cycles goes on, it is unavoidable to modify it over time. In a monolithic system however, it becomes increasingly difficult to maintain a modular structure—if one was present to begin with—which in turn also leads to modifications becoming more expensive. The expensive modifications are explained by the fact that if a modular structure cannot be guaranteed, it possibly entails that changes to one component may impact other components which were not supposed to be affected. This may result in many adjacent changes and unexpected issues to be performed and treated respectively.

Lastly, if we want to scale an application we actually need to scale the application as a whole rather than the individual components that need to be scaled.

Figure 1 on the following page nicely demonstrates some of the mentioned issues and how the MS architecture tries to tackle them.

Since the application is now split into separate services, making modifications will not require us to build and deploy the whole application anew. It will be enough to only update the services in question.

Further, the fact that the application is built in a modular fashion from the ground up and divided into smaller chunks and services, makes it easier to maintain a modular structure within the application as a whole. As a result, making changes to one such service will guarantee that none of the other services will be affected and thus keep undesired or unforeseen side effects to a minimum.

We can also see that scaling an application becomes much less of a burden. No longer do we need to scale the whole application, but we can simply introduce new services where needed.

4.3. Production (± 40% of section's words)

A first simple definition can be found in the MSs article [1] that lays out some aspects which characterize a MS architecture.

> In short, the microservice 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 [. . .]. These services are built around business capabilities and independently deployable by fully automated deployment machinery. There is a bare minimum of centralized management of these services, which may be written in different programming languages and use different data storage technologies.

However, as stated multiple times in the article [1], there is no formal definition for MS. As such, we shall inspect characteristics that such architectures tend to have in common.

Before we move on, an interesting point to note here is the decreased need for *centralized management*. To give an example [2], centralized management is a structure where only a few individuals make most of the decisions in a company. We can find hierarchies within such a structure, each having to respond to the superior's communicate.

Decentralized management on the other hand would for example allow a manager at a call center or retail store to make instant decisions that impact their work environment. Thus, the decisions are not taken solely by the higher ups but responsibilities are spread across sectors—which could further spread responsibilities. This is an interesting analogy to keep in mind for the following.

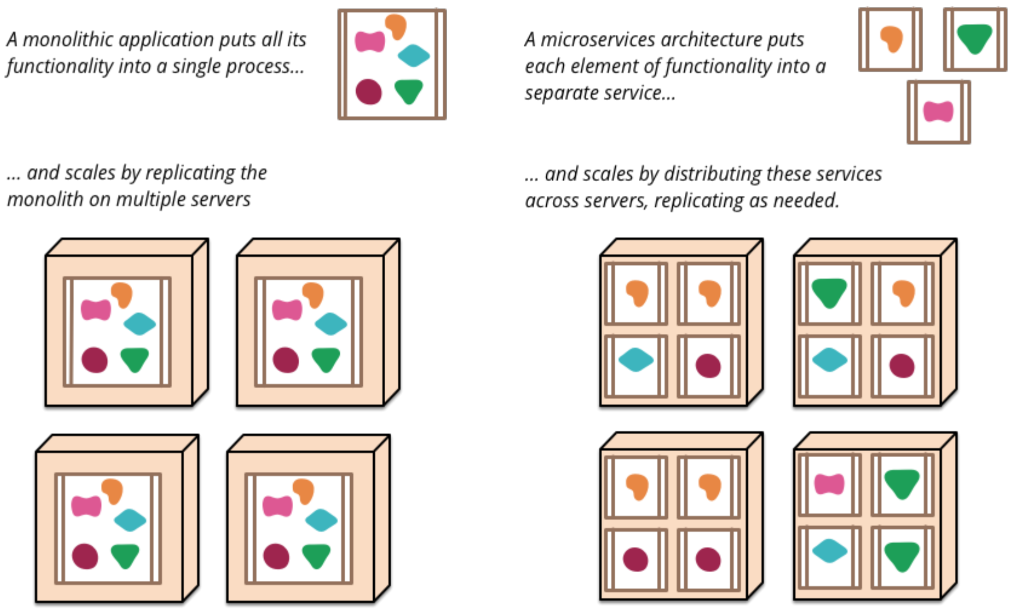
A monolithic application puts all its functionality into a single process...
A microservices architecture puts each element of functionality into a separate service...
... and scales by replicating the monolith on multiple servers
... and scales by distributing these services across servers, replicating as needed.

Fig. 1. Monoliths and Microservices

4.3.1. Componentization via Services. The article [1] distinguishes between two types of components—where *component* refers to a unit of software that is independently replaceable and upgradeable:

libraries which are being linked into a program and are called using in-memory function calls; and

services which are external components where communications happen with mechanisms such as web service requests or remote procedure calls.

The advantage of *services* is that such components can be deployed independently—which is not the case with *libraries* because then a component is essentially directly integrated into another component. It should be noted that some service component modifications may affect its interface, which in turn will require other service components to be adjusted accordingly. The aim however is to minimize these through cohesive service boundaries and evolution mechanisms in the service contracts.

Another advantage of services is a more explicit component interface. Often it's only documentation and discipline that prevents clients from breaking a component's encapsulation, thus leading to overly-tight coupling between components. Services make it easier to avoid this by using explicit remote call mechanisms.

4.3.2. Organized around Business Capabilities. Traditionally, splitting a large application into parts is done by assigning a specialized team to each layer of the application, as illustrated in figure 2. This distribution of work force eventually leads to each team worrying only about their specific task instead of the application as a whole, which is an example of *Conway's Law*:

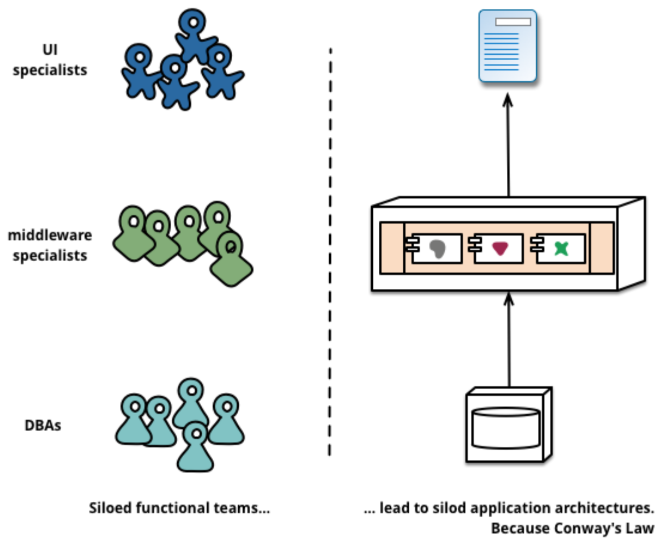
UI specialists
middleware specialists
DBAs
Siloed functional teams...
... lead to silod application architectures. Because Conway's Law

Fig. 2. Conway's Law in action

> Any organization that designs a system (defined broadly) will produce a design whose structure is a copy of the organization's communication structure.

In regards to MS, we divide the work force according to *business capabilities* rather than application layers. This implies that our resulting team will be cross-functional[2], combining the full range of skills required to develop the service in question. This division leads us to figure 3 on the next page.

2. meaning that our team is comprised of people with different expertises, all working closely together for a common goal

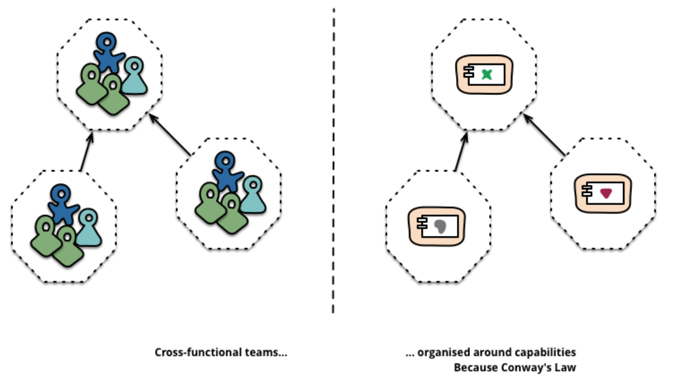

Fig. 3. Service boundaries reinforced by team boundaries

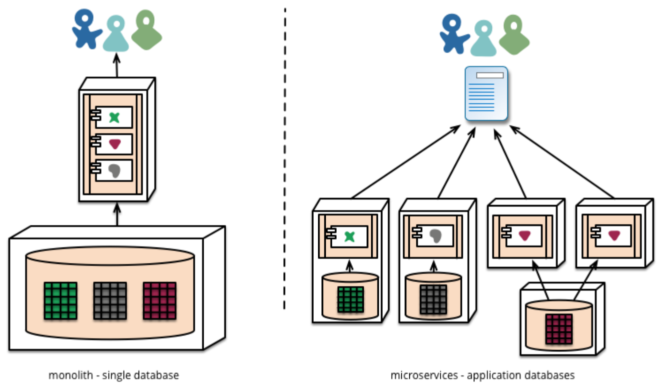
monolith - single database
microservices - application databases

Fig. 4. Monolith and Microservice Databases

4.3.3. Products not Projects. Most of the application development focuses on delivering a piece of software which is then considered completed. It is then handed over to a separate maintenance team and the original team that built it will then completely abandon the project.

The MS philosophy tries to avoid this approach and instead considers that a team should be responsible for its own product over the full lifetime. Here is also a link to *business capabilities* in that software is no longer considered as a set of functionalities that need to be completed. Rather it is seen as an on-going relationship where one strives to enhance the business capabilities by having the software assist its users.

4.3.4. Smart endpoints and dumb pipes. Often times when it comes to process communication, great importance is put on the communication mechanism itself. With regards to microservices however, we place greater importance on the endpoints rather than the pipes.

Thus one should strive to decouple the MSs and keep them as cohesive as possible. One could almost say that they follow the Unix philosophy in that they receive a request, treat it appropriately, and then return a response. MS can achieve this by using simple REST-like protocols.

This further highlights that services in a microservice architecture are to be isolated components. When they communicate, the magic happens on the services' endpoints and not in the message bus—in fact we need nothing more for messages than simply being transferred from one endpoint to the other.

4.3.5. Decentralized Governance. Centralized governance that often comes along with monolithic architectures have the tendency to standardize on single technology platforms, which can be restricting. Rarely can a single language's full potential be leveraged for the entirety of an application.

Splitting our application into services however, we do have the ability and choice of building each service with whatever tool is best for the job.

This approach also brings a different mindset into development. Instead of focusing solely on defined standards for a product, developers prefer the idea of creating useful tools that can potentially also be used by other developers to solve similar problems.

4.3.6. Decentralized Data Management. Monolithic applications usually prefer to have a single logical database for persistent data, which in turn are then often used across a range of applications. Microservices on the other hand let each service manage its own database. Both situations are illustrated in figure 4.

The issue that arises with decentralizing the data, however, is consistency. Managing this is in fact not an easy task and usually comes at the cost of increased computing times, leading to *eventual consistency* and *compensation operations* being the preferred method. In fact, businesses are often capable of handling a degree of inconsistency in favor of quick responses to demands.

4.3.7. Infrastructure Automation. Many products following the MS architecture make heavy use of Continuous Integration and Continuous Deployment (CI/CD) and infrastructure automation tools. This becomes especially relevant in production when you have to manage a lot of different MSs at once, which simply becomes infeasible without the automation.

4.3.8. Design for failure. One needs to take into consideration that a service may fail at any point in time for whatever reason, and the client should respond to this as gracefully as possible. Due to the decoupled nature of MSs, we have an additional complexity to take into account which constantly requires us to reflect on how service failure can affect the user experience.

It is therefore of utmost importance to detect failures quickly and restore the service. To achieve this, a lot of sophisticated monitoring and logging on various parts of a service are performed.

4.4. Assessment (± 15% of section's words)

Given all the above, there are drawbacks that come with the microservice architecture. As already pointed out previously in section Design for failure, a service can fail at any point in time which makes it quite difficult to build distributed systems. Added to this, remote call are slow which can also have an impact of the final product's performance. Another drawback that we discussed previously is the issue of eventual consistency as mentioned in section Decentralized Data Management.

As slightly touched upon in the Infrastructure Automation section, we will run into the issue of *operational complexity*[3], demanding a mature operations team to manage lots of services, which are being redeployed regularly. A product consisting of half-a-dozen applications can quickly expand into hundreds of little microservices. [3]

Here, the role of DevOps comes into play to ease this whole process. It becomes increasingly harder, tedious and eventually impossible to manage an increasing number of services which also calls for a lot of automation and monitoring to take place. Given that MSs are smaller and thus easier to understand, problems are likely to occur in the interconnections of such services which potentially makes it very difficult to trace and debug issues—hence the need for proper and elaborate monitoring.

[KB: Highlight elements pointing to TD?]

5. Scientific Deliverable 2 – What is the relationship between DevOps and Microservices?

5.1. Requirements

This scientific deliverable aims to find a relationship between the DevOps approach and the MS style. In the design section we are therefore first going to introduce aspects of both DevOps and MSs. The production section will then draw parallels between the presented characteristics which in turn allows us to observe how both concepts are connected to each other. In the assessment we will talk about threats to the validity of our discussions.

5.2. Design

[KB: Talk about characteristics of DevOps and MS]

5.3. Production

[KB: Make the connection between both concepts]

5.4. Assessment

[KB: Threats to validity of the discussion in production section]
[KB: Highlight elements pointing to TD]

3. while small independent services are easy to deploy, the number of services adds strain to the complexity regarding operations

6. Technical Deliverable 1 – Does E4L allow for easy deployment of adjacent Microservices?

6.1. Requirements

Our case study consisted in creating a MS relying on the Energy4Life (E4L) application. The latter is built using a monolithic architecture, and we want to find out how easy it is to deploy an adjacent MS application.

The MS to be created should allow us to test the hypothesis that MSs are easy to create and deploy. Further, it should fit into our E4L case study—we will have our service display a graph based on data collected by the E4L application. To close everything off, we shall also create a GitLab pipeline that automates the deployment of this MS.

Hence, the design section will tackle all the steps we had to undertake in order to reach the final and fully automated deployment pipeline. The production section will dive into the technical details and difficulties of each step. The assessment is going to lay out our judgement on how easy it was to deploy the MS given the current E4L application architecture.

6.2. Design

[KB: Methodlogy: Steps required to achieve final solution]

6.3. Production

[KB: More details on each step + difficulties encountered]

6.4. Assessment

[KB: Was is easy to deploy MS?]

Acknowledgment

The authors would like to thank the BiCS management and education team for the amazing work done.

7. Conclusion

The conclusion goes here.

References

[BiCS(2018a)] BiCS Bachelor Semester Project Report Template. https://github.com/nicolasguelfi/lu.uni.course.bics.global University of Luxembourg, BiCS - Bachelor in Computer Science (2017).
[BiCS(2018b)] Bachelor in Computer Science: BiCS Semester Projects Reference Document. Technical report, University of Luxembourg (2018)
[Armstrong and Green(2017)] J Scott Armstrong and Kesten C Green. Guidelines for science: Evidence and checklists. *Scholarly Commons*, pages 1–24, 2017. https://repository.upenn.edu/marketing_papers/181/
[1] Microservices: A definition of this new architectural term https://martinfowler.com/articles/microservices.html
[2] Centralized and Decentralized Management Explained https://content.personalfinancelab.com/finance-knowledge/management/centralized-and-decentralized-management-explained/?v=c4782f5abe5c
[3] Microservice Trade-Offs – Operational Complexity (Con) https://martinfowler.com/articles/microservice-trade-offs.html#ops
[4] [¿VIM¡] [⟩VIM⟨]

8. Appendix

All images and additional material go there.