Microservices Architecture on Energy4Life Monolithic Application

Thursday 3rd June, 2021 - 01:00

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

Our Bachelor Semester Project (BSP) lives in the software engineering domain, both for the scientific and technical parts. In the scientific part we will briefly explore the concepts and their relations that were of interest in this project. In the technical domains we shall briefly present the tools that we used to achieve our implementation and how they were relevant.

2.1.1. Scientific. In order to build software, there are steps preceding the actual production that have a huge impact on the overall product—be it from the user, developer or operator point of view. Creating software consists among other things in finding both the functional and non-functional requirements. At a later point we also need to start *designing*. This phase is where we make decisions—about the product—that should ultimately lead to fulfilling the requirements. One should also think about methods and techniques that will aid in this endeavour and think long term about the software life cycle.

One way to structure our software is by using the Microservice (MS) architecture which structures everything into services—as will be discussed in section 4 on page 3. Another possibility would be a *monolithic* architecture which would for example structure according to a server-client model. For this BSP we decided to investigate MSs that exist along side monoliths.

Finally, in order to facilitate the managing of MSs, we resort to the DevOps philosophy. [5] If we have multiple services to manage, DevOps will allow us to make the process of building, deploying and maintaining the product fast and efficient. Particularly interesting for us is the focus on automation and the therefrom arising benefits as will be discussed in section 5 on page 6.

2.1.2. Technical. We used a GitLab platform to host our code. The Energy4Life (E4L) code is in fact also hosted using a GitLab platform, however it lives on a different GitLab

instance than our MS. Deploying the MS would be done onto our Juno testing environment.

To deploy our MS we relied on docker which allows us to create isolated containers. Docker is in fact also what allowed us to properly implement the MS in itself architecture-wise.

Further, GitLab provides so-called *runners* that give us the ability to execute tasks or commands automatically. Thus we can make it such that our MS will be automatically redeployed any time we make a modification. The runner can either be set up locally on the server, or run inside a container; we ended up using the former.

The actual application running in the MS was written using Python. We used the `dash` module to easily create a web page containing a graph. Further python modules were used to make HTTPS requests and connect to an SQL database in order to grab some data to be plotted. Within the MS, we also relied on the Linux `crontab` utility to periodically grab new data during the service's life cycle.

Our technical contribution shows how one can deploy a MS along side an existing monolithic application. Further technical insights shall be provided in section 6 on page 7.

2.2. Targeted Deliverables

Here we will present how the deliverables are related to each other. This allows us to briefly present each deliverable as well as why it is relevant and was delved into. Let us start with the technical deliverable.
[KB:include mindmap]

2.2.1. Technical deliverables. The main goal of our technical deliverable was to inspect how easy it is to deploy a MS given the E4L application—which we assume to be a monolithic application. Given the research presented in section 4 on the next page, we concluded that MS would not be fitting the architecture. Prior to working on this BSP—as well as was discovered and confirmed during the project while inspecting the E4L code—it was known that this application had the typical front-end, back-end and database structure. There was also no notion of services in the MS-sense of the word.

Due to the fact that MSs are supposed to be isolated components, it was clear from the start that we should not modify the original E4L code for our purposes. In fact, given our use-case of plotting an arbitrary graph using data provided by E4L, our best bet would be to rely on an API that is exposed or to connect to its database directly. The worst scenario would be one where neither were available; in this case we would have no other choice but to make our own API by modifying the E4L code or by creating a wrapper around other functionality that is exposed by E4L.

This technical deliverable will dive into the whole process that we went through from the beginning until the moment we had the final MS up and running alongside E4L. The way we conducted the experiments at every step was a very iterative process. A little research on a given matter would help us get familiar with a particular topic as well as how to approach the implementation. We would then create a dummy code to test said features and keep it around as a reference in case something breaks in the main implementation. From there it was a matter of fixing issues that arose as well as adapting and refining the solution until we obtained the product we sought to create. If further features were required, we introduced them step by step by following this iterative process which was accompanied by lots of dummy implementations for reference and testing.

However, before being able to fully realise this task there were some notions that we needed to acquire and get familiar with. First of all, what even is a MS? Further, given that we were working using GitLab and wanted to take advantage of its Continuous Integration and Continuous Deployment (CI/CD) framework, we naturally wanted to find out how the concepts of DevOps and MSs are related—since CI/CD is also employed in the DevOps context.

2.2.2. Scientific deliverables. The first scientific deliverable therefore consists in inspecting what a MS is. Given how MSs are a fundamental part of of this BSP it was essential to learn about them and what is considered to be a MS. Once we had this notion we could expand on it and construct a meaningful service.

The second deliverable consisted in finding out how DevOps plays a role in the MS philosophy. There were indeed quite a few aspects that hinted at a correlation but we wanted to find out where and how both interplay. This in turn would directly feed back into the technical deliverable as we would obtain a better overview on the overall structure and approach.

The MS part was of importance in order to actually create the service in question, while the DevOps side allowed us to automate our solution and take care of the many manual steps needed to finally deploy an application—which would be of great importance if we start to manage more than one MS.

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

3.1. Scientific pre-requisites

The main scientific knowledge needed for this BSP was a general knowledge on software engineering approaches. It was of great value given how MSs—studied during this project—are a further evolution in the software engineering domain.

3.2. Technical pre-requisites

The technical knowledge needed here was familiarity with the Python programming language as we ended up relying on it for the implementation of our MS. Also being able to work in a Linux environment was important for the later stages of the project where we wanted to manually deploy the application on a Linux server.

4. Scientific Deliverable 1 – What is a Microservice?

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

Since 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.

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.

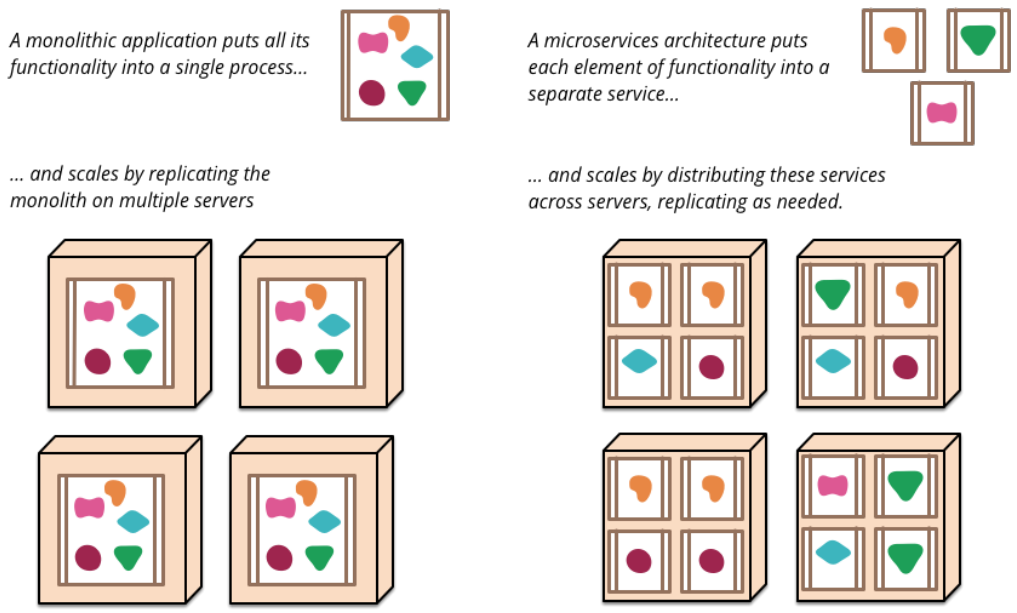

Fig. 1. Monoliths and Microservices

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*:

> 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.

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.

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

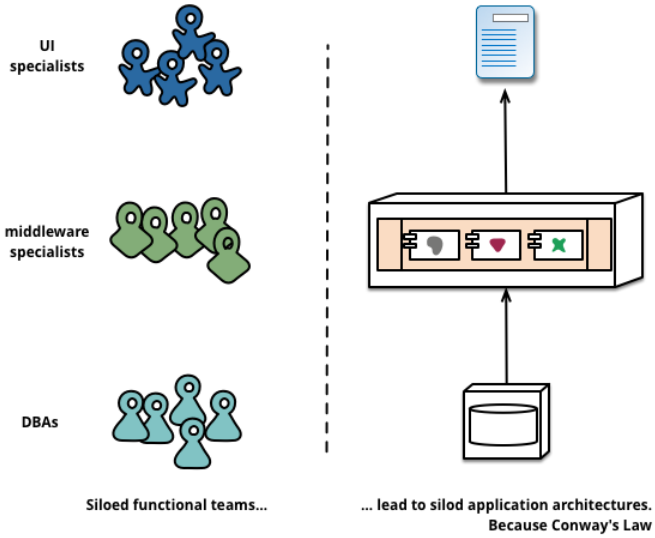

Fig. 2. Conway's Law in action

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

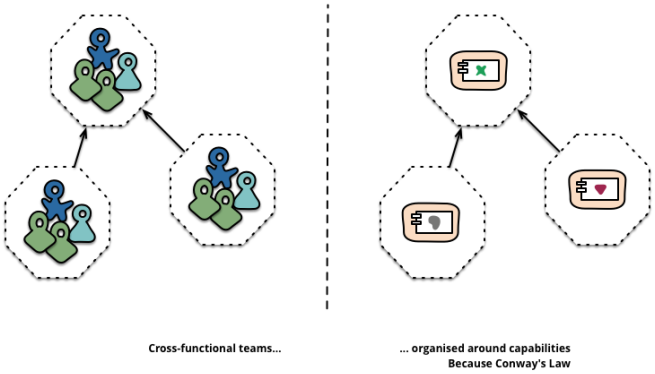

Fig. 3. Service boundaries reinforced by team boundaries

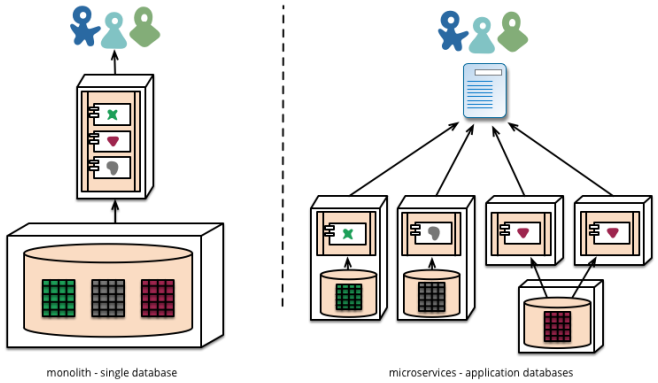

Fig. 4. Monolith and Microservice Databases

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 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.

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

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

5.2.1. Relevant aspects of microservices. In section 4 on page 3 we have introduced characteristics that generally make up a MS. We shall briefly touch upon some of the ideas again and give a bit more insight that is relevant for our comparison with DevOps.

5.2.1.1. Rapid provisioning. To be able to rapidly launch a server—within a few hours for instance—it is important to automate the process of provisioning[4]. To make serious use MSs, one should strive for a fully automated process—as far as this is possible. [6]

5.2.1.2. Basic monitoring. Due to loosely-coupled services working together in production, things are bound to go wrong in ways that are difficult to detect in test environments. It is therefore of utmost importance to make sure serious problems—appearing through technical as well as business issues—are detected swiftly.

Technical issues could encompass run-time errors, or service availability. *Business issues* can be related to a drop in orders, or problems with sending transactions through multiple services. [6]

5.2.1.3. Rapid application deployment. A pipeline comprises the building, testing and deploying of an application which are steps that tend to take a long time to run through. The idea is to break up these steps and the tasks within them into stages. While it will add extra time to run through, we benefit from from greater confidence because early stages can detect problems that may have devastating effects later down the road. Later stages, in turn, can make more in-depth testing to gain extra confidence in the finial product. Stages in a pipeline can be automated or require manual intervention and may even be parallelized—if applicable—to speed up the process. Deployment pipelines are in fact a central part of CI/CD. [4]

Deployment usually involves a pipeline with various steps, each providing some sort of validation of the product. With many services to manage, this pipeline should striven to be fully automated. Achieving this implies close collaboration between developers and operations: the *DevOps Culture*. [6]

5.2.2. Relevant aspects of DevOps. Let us now have a look at the DevOps culture before drawing out the relations that exist between the MS and DevOps philosophies. We will again have a look at aspects that are relevant to us but nonetheless do not obfuscate the overall approach that DevOps seeks to follow.

5.2.2.1. The DevOps approach. Traditionally software development is broken down into distinct categories such as requirements analysis, testing and development as well as deployment, operations and maintenance once the product is released. DevOps aims that merging these categories and favour collaboration between **dev**elopment and **op**eration**s**. [5]

5.2.2.2. Shared responsibilities. A side effect of the DevOps approach are the *shared responsibilities*, encouraging closer collaboration. As briefly mentioned in the Organized around Business Capabilities section, now that developers and operators are working under the same hood, the former is much more inclined to adapt a mindset of finding ways to simplify deployment and maintenance since they are closely involved with the tasks of the operators. If the development team were to just hand over the finished product and not worry about it anymore, they would not see the difficulties operators might need to face in light of their productions. [5]

5.2.2.3. Quality of productions. Given that each party is more involved in each other's tasks, this leads to better understanding of the processes and issues which developers and operators need to tackle. This allows for much more efficient problem solving and leads to teams needing to value building quality into the development process. The aim being to automate the deployments and speeding up the testing cycle which results in greater ease of putting code into production. [5]

5.2.2.4. Automation. Automation is an important factor in all of the above. For one, automated tasks—such as testing, configuration and deployment—free staff of these burdens and reduce human error. Another point is that the automation scripts essentially serve as useful and up-to-date documentation of the system. Should, for example, a developer or operator want to inspect or change how a server is configured, they know where to look. [5]

5.3. Production

Given the above points and a few of the characteristics of MSs, one could derive the implication that DevOps is an inherent aspect that comes along with the microservice philosophy. The two points that lead us to this assumption were the Organized around Business Capabilities and Infrastructure Automation characteristics of MSs.

4. this is the process of adding and configuring tools to a virtual machine

The infrastructure automation becomes increasingly relevant if our network of MSs expands. Managing only a handful of services by hand can still be feasible, but if we have a few dozens or more services this quickly becomes a virtually impossible task as we would need to take care of each of the countless services individually. As for the DevOps side of things, in order to increase production speed the automation is an equally important factor. Not only does it free the staff of work, but it also reduces human error—which seems likely to seep in when managing lots of services in parallel. These viewpoints on automation allow us to draw an assumption on how the DevOps approach is vital in order to make working with microservices a feasible task.

The second point is that the automation scripts serve as useful and up-to-date documentation of the system. This is very useful in that both developers and operators can inspect these scripts and be on the same page with regards to how the system is built and works. This directly leads us to the MS characteristic of *organizing around business capabilities*. The fact that we do not split our teams according to application layers—as is usually the case for monoliths—but rather according to functionalities greatly favours collaboration of each application layer's expert. The automation scripts coming forth through the DevOps aspect serve in this regard as a bridge to ease this collaboration among team members and thus help further boost and encourage the idea of organising in terms of functionalities rather than isolated application layers.

5.4. Assessment

Possible threats to validity of the above discussion and assumptions stem from the fact that they were drawn with disregard to the technologies. Merely the researched material presented until this point were taken into consideration. Further, we did not have the opportunity to experiment with this organisation of teams in practice, which would surely provide a much better insight and grounds for our assumptions. That said, given more expertise on how MSs and DevOps work in practice might give a more precise and elaborate answer.
[KB: Highlight elements pointing to TD]

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 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; figure 5 on the following page shows an example of our graph. 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

6.2.1. Set up repository. The first step was to set up a repository in which we could place our productions. I had to request an account in order to access the https://gitlab.uni.lu GitLab service. An interesting thing to note at this point being that this GitLab service will host our MS code on a different server from the one that contains the E4L code. Thus we can further verify the independence of such services.

6.2.2. Familiarize with docker. We then had to get familiar with the *docker* utility and how to create and manage containers with it. This tool is what would effectively allow us to create isolated services by containerising our application into a docker container.

The rationale behind using docker instead of simply running a process directly comes back to the points mentioned in the Componentization via Services section. Simple processes would not enforce the MS architecture as much as containers would, thus allowing for development to shift into a monolithic architecture.

At a later stage we also inspected the *docker-compose* utility which allows for finer and more automated management of containers.

6.2.3. Familiarize with CI/CD. In order to automate the deployment of our application, we want to take advantage of GitLab's CI/CD framework. At this point in time our MS was not fully functioning yet and we were simply experimenting with dummy code. The main goal here was to understand how it works and what it does.

Creating the actual pipeline for our finished MS was only the very last step of the process. The rationale is that automating such a procedure can only be done if one is familiar with the steps that would otherwise need to be done manually. Therefore it was important to work out everything by hand before finalizing with an automated pipeline.

6.2.4. Obtain data from E4L. Since our MS would display a graph using data from E4L responses, we had to find out how to extract this information. Finding this out was quite an extensive process in itself since the hints were not too clearly laid out in the documentations. More details will be provided in the production.

To give an outline, our first attempt was to sort of brute-force our way into obtaining data. While not the most useful of data, it was something to work with. Only later have we

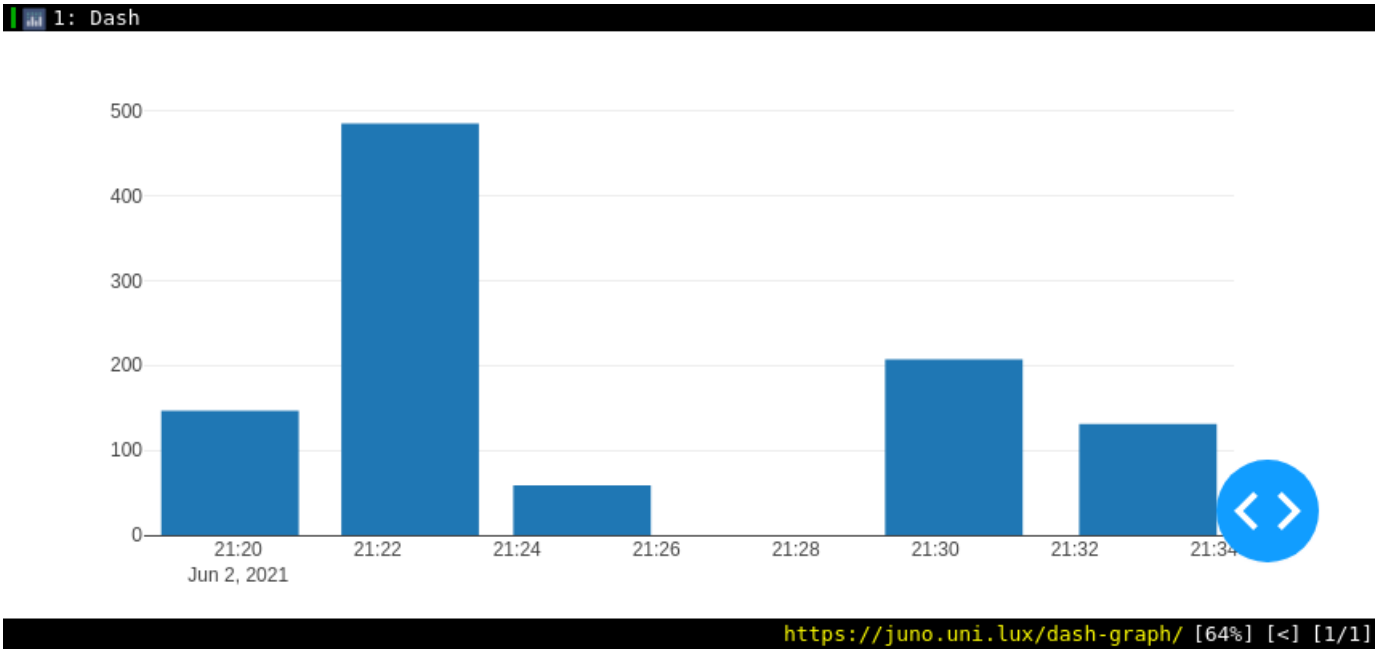

Fig. 5. MS graph relying on E4L API

discovered that E4L exposes an API which greatly alleviates the burden of our brute-force approach.

Obtaining useful data however, was only achievable thanks to the help and guidance of a prior team member of E4L who had great responsibility in creating the back-end. He showed me the proper way to obtain all the data from the API as well as by directly connecting to the E4L database.

6.2.5. Write code which retrieves data and displays it. For this purpose we relied on productions made during my summer job where I was responsible for finding ways to plot graphs. For the sake of simplicity we used Python to write this code. It should be noted that this and the previous step of obtaining data were always running in parallel.

In the end, it was simply a matter of retrieving the data from the API or the database, and then to plot some data using example code produced during the summer job. It should further be noted that we were not deeply interested in *what* data is being displayed, it should rather serve as a proof of concept for our purpose.

6.2.6. Containerize the application. Once the application was working locally, we attempted to make it run inside a docker container. Note that only the version retrieving data from the API was used for experimenting this way. The database version is analogous and did not require great modifications. Further it had a restriction where the database was only accessible to certain docker containers running on the server where the database lives. We will provide more details in the production.

Containerizing the application came with a fair share of issues however both locally as well as once deployed on the testing server. Work-arounds and fixes were implemented an resulted in a fully functional service.

6.2.7. Create pipeline to automate deployment. The final step was to create a pipeline that automates the whole process of deploying the application to Juno. Since we were making a Python application, there were no building stages that had to be made in the pipeline. Thus we only ended up with a deployment stage which was very short thanks to the docker-compose utility taking care of everything.
[KB:Methodlogy: Steps required to achieve final solution]

6.3. Production

[KB:screenshot showing final product: E4L page with graph]
[KB:Reference *replication package* in README of code/repository]

6.3.1. Set up repository. Once access to GitLab was obtained, it was relatively straight forward to create a new repository. The only special thing we did was to host the repository under a group instead of the personal space. This would later allow us to have shared GitLab runners for executing the pipelines.

Projecting a little, nothing would have prevented us from setting up project specific runners for our two solutions, but given they require the same type of runner it made sense to share a runner, thus the rationale for creating the projects under a group.

6.3.2. Familiarize with docker. Besides following tutorials, we also played around with the tool in order to better grasp how it works. For example we wrote a simple hello world

program in C. We then tried compiling it before building the container and only put the binary into it. And we tried compiling the program while building the container.

6.3.3. Familiarize with CI/CD. Better understanding the CI/CD pipeline was done in the same fashion as for docker, using a bunch of experiments to understand how it works. For instance again compiling the C program using docker, or compiling it using the CI/CD pipeline.

6.3.4. Obtain data from E4L. Our MS should obtain data from E4L and plot it using a Dash/Plotly application. Obtaining the data was quite a journey however, let us explore this process.

6.3.4.1. First attempt at obtaining data from the back-end. Upon inspection of the E4L backend, we were able to find a potentially relevant endpoint that would allow us to query answers from the questionnaire. However, it quickly became apparent that the back-end endpoints were specifically made to be used by the front-end page. Further evidence for this can be found in the functionalities described in the README, which also informs us that the only data that can be extracted is specific to a session. All of this essentially means that we have no direct way to access any data beyond what we are presented with on the web page—which would only comprise our results from after answering the questionnaire.

Upon completing the questionnaire in E4L, we are presented the following page[5]: https://juno.uni.lux/e4l/result/MzA5.-7-sMYXmv3tXyuLQT2s1ZgULRKY. The so-called *sessionId* is unique to our provided answers, so by taking note of it we are able to review our *energy scores* at any point in time. The obvious course of action now would be to use the `/calculate/session/{sessionId}` endpoint—which is our only of two GET handlers regarding the questionnaire—to obtain our results.

6.3.4.2. Scraping the results page for data. One problem rose quickly however: We had no information on where or how to access the back-end in order to issue our GET request. Upon inspection in the code for both the front-end and back-end, we were unable to find any concrete traces on how to reach it. The closest we managed to find was the definition of a `baseUrl` using `axios`, but that still left us clueless. So we set out to create a web scraper that would parse the HTML of the above results page.

We used `selenium` to perform this task. However, running the selenium browser in headless mode—as is required for running this code in a docker container—caused issues with our scraper. It turns out that the page contents would not be loaded unless the page was actually rendered by the browser. In addition to this approach being extremely slow due to `selenium`, it was simply not usable for our use-case.

6.3.4.3. Obtaining data from the back-end API. It was only after a third and final inspection of the E4L code—performed while documenting here what we did—with the help of GitLab's search utility, that we discovered how to access the back-end. It was hidden in plain site in the front-end README, however it was not emphasised very well which caused us to completely overlook it at multiple occasions.

That said, we were able to retrieve our results using the API by issuing a GET request on https://juno.uni.lux/e4lapi/calculate/session/MzA5.-7-sMYXmv3tXyuLQT2s1ZgULRKY. However, some data about global statistics (i.e. Luxembourg, Europe, World) were not available as they seemed to be hard coded into the front-end.

6.3.4.4. Obtaining all the data by querying the API and database. A former member of E4L helped in querying the application for data in the correct manner. This would also allow us to obtain data for all responses that were collected from the questionnaire, and not only those specific to our own answers. We had to perform two HTTPS requests with special information to make this happen.

Connecting to the E4L database directly worked a little differently however because of the way everything is setup. The database is in fact only accessible to applications that are deployed as docker images on Juno and are part of a docker network. Given that our solution already relied on docker, this was quite easy to implement: We only had to register the image to the network using docker-compose. Then we could simply connect to the database using SQL. However, some information was not available here as it was computed by the back-end using information from the database.

6.3.5. Write code which retrieves data and displays it. For the API solution on obtaining data, we simply had to include the corresponding HTTPS requests into the code. We can then plot a graph using this data. Making the graphs was done using Python Dash. As a little side note: obtaining the data could have been done by hand. Writing this procedure as code allowed us essentially to automate this task.

The database solution required us to include the necessary lines into our docker-compose file in order to be able to reach the database on Juno. Once this was done, we used a Python SQL library to connect to the database and perform queries. This approach however was not directly possible to be done by hand as we were sort of restricted by the docker network for accessing the database. Figure 6 on the next page shows an example graph using data from the database. One should note again that the data plotted does not necessarily bear meaning, the main goal was for this to be a proof of concept. You can also see that we plot different types of graphs to avoid confusing the API and database versions.

6.3.6. Containerize the application. Both our solutions—for the API and for the database—were in the end nothing more than a single Python script. Containerising this was therefore as simple as copying the files into the container and execute them using Python. This lead to an unexpected issue however where the Dash page would not render any graph for both the API and database solutions.

5. Note that the ID will be different. This is an example for a results page that we obtained at some point.

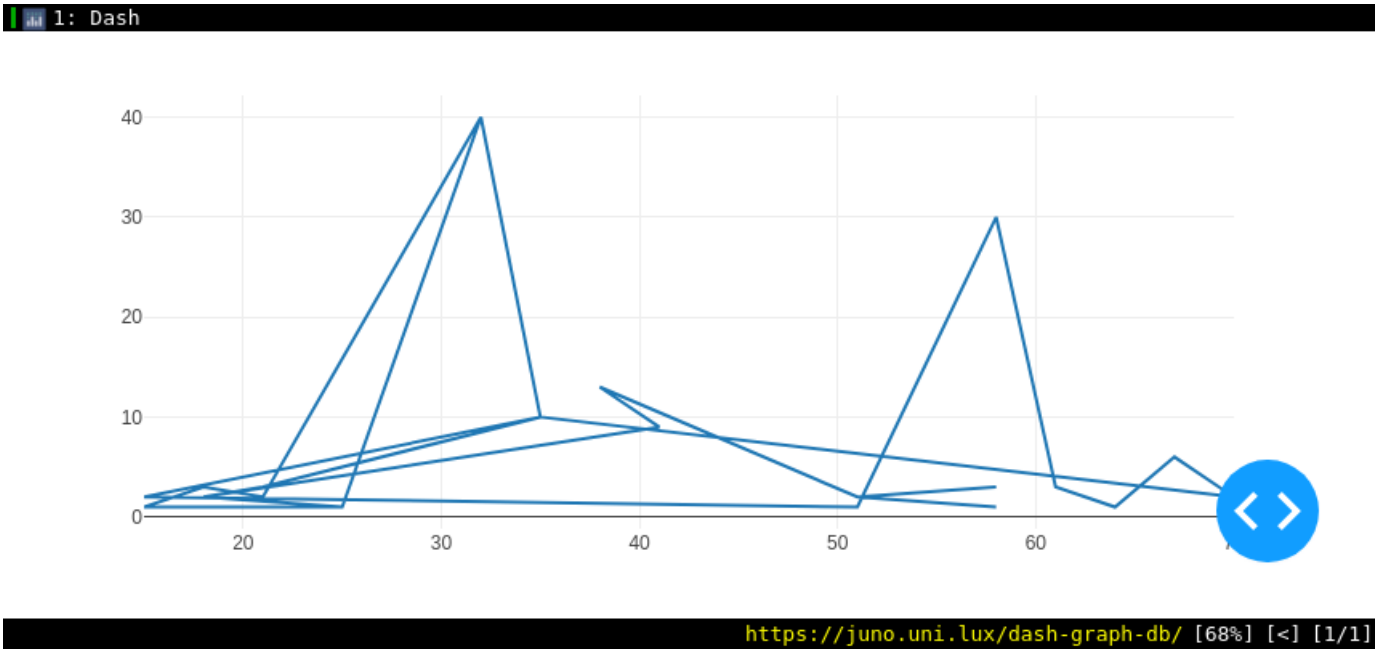

Fig. 6. MS graph relying on E4L DB

Extensive experimentation using dummy code and our actual solutions[6] clearly indicated that this issue only appeared if we query for data and launch the Dash application in the same script file. We further made a lot of research and experiments but we could not find the cause of this issue or how to solve it. Few resources on Dash with Docker could be found where the code resembled what we have created, none containing any answers to our problem.

Finally, the workaround we implemented was to simply put the query of data and the plotting of the graph using Dash into separate Python scripts which we would call one after the other. The querying would write the results to a file, and the Dash graph would read the information from this file. This effectively solved this mysterious issue that only appears inside a Docker container.

6.3.7. Create pipeline to automate deployment. Before being able to automate the deployment we had to get familiar with how to do it manually. Deploying the application to Juno was very straight forward thanks to the fact that we used a Python application that does not require any compiling or building. Further our containerized solution also takes care of everything for running the code. So in the end we only had to create and launch our docker container which was as simple as running one single command taking advantage of docker-compose.

As such, the automated pipeline will need to do nothing more than to execute this exact command. Given how E4L used docker-in-docker[7] in the pipeline for hosting the application, we naturally tried to do the same. But here again we encountered an issue that is still not solved. For some reason docker related commands gave birth to a Python docker API exception, thus making the pipeline fail—which was not observed for E4L which used the same approach.

After a lot of vainly research, our workaround was therefore to not use docker-in-docker, but to have a new GitLab runner that runs directly on Juno and not inside a container. As such, the commands in the pipeline will be executed on Juno directly—exactly the same way we did manually.

6.3.8. Update the graph information periodically. The way our solution was built made it such that data was only retrieved once when launching the container. If new data gets added to E4L, it would not reflect in the graph as no new data will be fetched. Our solution to periodically grabbing new data fully took advantage of our previous solution where we split our script in two; one taking care of querying the data and the other for making the graph.

We changed the Dash script in a way where it would read data from a file—produced by the first script—every time the page is loaded. So if our first script grabs new data and overwrites the file, the Dash application will read the data again when reloading the page. This was a very simple change, so now we just need to periodically call the data query script in order to update the data.

Given that our code runs in a docker container relying on Linux, the simplest was to implement a system-wide service—but container local in this case—that would run the query code

6. We only experimented on the API version, as the database version required us to work on the Juno server and the issue was analogous in both solutions. It was simply a matter of convenience It was simply a matter of convenience.

7. GitLab runner executes in a docker container, and we deploy our container using it.

every once in a while. Making a systemd timer service seemed like the best idea as most Linux distributions use systemd. However, the image that we used in our container seemingly did not ship with any systemd utilities, although evidence suggested systemd was probably present.

After a bit of research, we ultimately decided to create a cron job using the `crontab` utility. We set up this job to run our data query script every hour. This means that the data file will be updated every hour, regardless of whether new information is actually present or not. Thus, if we load or reload the Dash page, the new data will be read from the file.

6.4. Assessment

Regarding the gathering of data from E4L, one could further improve this workflow by better highlighting the existence and features of the back-end as they seem to be somewhat scattered throughout the application's documentations and are easy to miss.

Also, given that the endpoints were specifically made to be used by the front-end, it implies that some of the data obtained is not fully usable for *general* API usage (i.e. the hard coded data in the front-end). Further, as could be observed in the implementation of the back-end endpoints, not all available database and service functionalities were reachable through the API (i.e. no way to directly query all answers for making statistics). A tight coupling of front- and back-end can be observed here.

Taking a step back and disregarding all the big technical roadblocks we encountered when trying to obtain data from E4L, when containerising our application and when trying to create the automated pipeline using docker-in-docker, it was actually very simple to set up this specific MS. It essentially only consisted in grabbing the data, making the graph, and updating the data periodically. Seeing how—in the end—we had a very precise entry point into E4L for obtaining the data, we had a lot of freedome in how to actually process the data. The isolation of MSs therefore provided us great flexibility which offered us the opportunity to work with tools we are most comfortable with.

One should keep in mind however, that this is the first and only MS created for E4L at this point in time, so we did not have to worry about managing clusters of services and taking care of all the complexities that come along with them. We were able to work with a simplified example of MSs.

A few things that are interesting to note now are that our MSs are hosted in a different place from the E4L code, which further accentuates the independence of MSs.

Further, the deployment of both our API and database solutions are the exact same thanks to the docker utilities. Inside these docker configurations there are slight differences, but they do not reflect in the actual deployment process. In fact, both deployments are completely unrelated from each other given that each solution has its own pipeline and repository. Making changes to one MS will not have an impact on the other one, since there is no common code base between the two. We only need E4L to be running for both solutions to function.

To assess our initial goal of finding out how easy it is to deploy a MS, one has to note that it was quite a bumpy ride all along due to the many unexpected and inexplicable issues encountered. However, once we finally had our automated pipeline we did not need to worry about anything anymore. We simply push our changes to the repository, and the application will automatically be redeployed without any further intervention. After going through the manual steps, and automating them, we have indeed achieved a state where deploying a MS is very simple and fast.

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] DeploymentPipeline https://martinfowler.com/bliki/DeploymentPipeline.html

[5] DevOpsCulture https://martinfowler.com/bliki/DevOpsCulture.html

[6] MicroservicePrerequisites https://martinfowler.com/bliki/MicroservicePrerequisites.html

[7]

8. Appendix

All images and additional material go there.