An introduction to Microservices and Monoliths, and when to use them

Felix Luthman

May 1, 2020

1 Introduction

The common view of an application would be a single piece of software, installed on the users machine, capable of executing all the tasks the program offers locally, i.e. using the host computer's resources. If we were to use technical terms, this type of software structure could be categorised as a Monolith. If this word does not make sense to you just yet, do not fret, it soon will. You see, with all the technological advancements made throughout the years, this, the Monolithic structure, and perhaps most importantly for this essay, the possibility of completely, or at least somewhat, avoid installing the software on the users hardware, no longer have to be the case. Applications that aren't installed on the user's machine, but rather deployed on a remote server, are called web applications, and I would be extremely surprised if anyone reading this have failed to encounter one in their lifetime. These web services have increased in usage and complexity in recent years, and have allowed for a new type of software architectural approach, the Microservices architecture, to emerge as a solution to a lot of the problems a Monolithic application may face. The definition of this new structure is an application consisting of a set of loosely coupled, collaborating services, as opposed to the Monolithic structure, which describes a single-tiered software application in which different components combines into a single program from a single platform. [5][7]

2 Monoliths

This is the classical way of developing an application, and perhaps the most common way to view a piece of software from an outsiders perspective. A single, large application that deals with all the tasks at the same place so to speak, it could not be easier. But there is actually more to it than that, and to make a comparison between the Monolithic and Microservices approaches, we first need to define what both of those two concepts are, or at least which definition we will use throughout this essay. As it turns out, there are three common interpretations of what the Monolithic architectural style is, all worth exploring below. It is also worth noting that the waters can be a bit muddy of which of these interpretations, or combinations of them, different resources out in the world, are referring to, so the reader is advised to use some common sense if she decides to venture out on the web and further explore this topic. [1]

2.1 Module - Monolith

I would suspect that every single reader of this article, or at least the ones who have a background in coding, have take part in the development of a modular Monolith, whether they know it themselves or not. This is because it is the most common starting point for aspiring developers out there, supported by most development tools used by beginners, as well as seasoned developers (IDE:s, text editors, you name it). What the concept means, is quite simply that the

entire project consists of one single codebase, dealing with everything from the business logic to the front-end, which is compiled as a whole, resulting in one lone artifact. [1] [4]

As a sidenote, the code itself may very well be structured into coherent modules with high cohesion, following every best practises known to man, while still following the Monolithic modular structure. This definition is only concerned with whether or not it is compiled into a single piece of software, which means that we are not currently concerned with anything but the development process at this stage. [1]

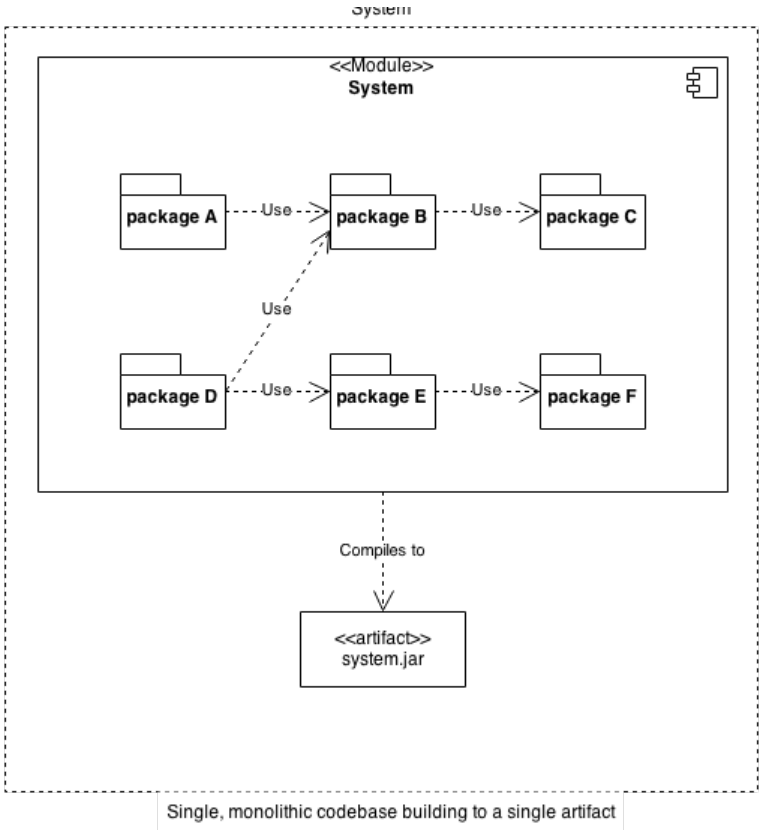

Figure 1: Illustration of a Monolithic codebase [1]

2.2 Allocation - Monolith

After the codebase has reached a stable state, it is time for deployment, i.e. shipping the finished code onto one, or more likely, multiple, server which then run our application. But, as we all know, a product is never finished, and soon enough, we will have a new and updated version of our application ready to be made available to our customers. If we are to once again follow the Monolithic

approach, this would be done on a global scale, meaning that the new version is deployed to all our servers at the same time, resulting in them all running the same version of our application at all time. In practise, this commonly would mean that the entire system is stopped in preparation for the rollout of the update, which subsequently would occur, allowing the system to go online once again. [1]

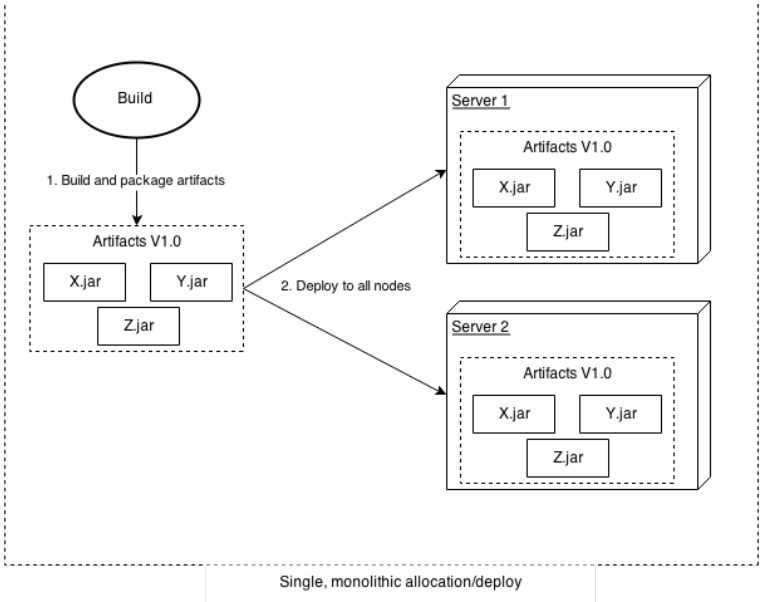

Figure 2: Illustration of a Monolithic deployment [1]

2.3 Runtime - Monolith

So, now we have created and deployed our application, and the only thing left is how it should handle the different tasks, or services, it provides. Once again, the Monolithic solution is quite simple, a single application should be in charge of all services, performing the whole task alone. However, to be clear, there may still be multiple instances of the application running, across multiple servers, to handle heavy loads. What we are currently concerned with, is how one instance of a request of any service is handled, which is that one instance of the application takes care of it, no matter which kind of request it is. This means that given the scenario that we have four services available (A, B, C and D), no matter which one we choose, an instance of the entire application will handle the request. [1]

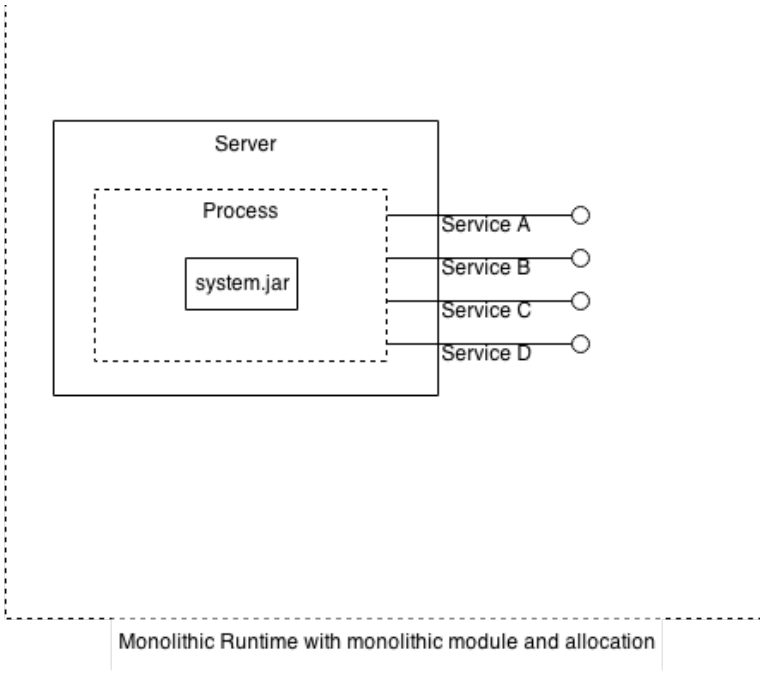

Figure 3: Illustration of a Monolithic runtime [1]

2.4 The interpretation used in this essay

Earlier, I mentioned that the definition of the Microservices architecture is that an application is structured as a set of loosely coupled, collaborating services. The alert reader may notice that this definition concerns neither how the code is built, nor how it is deployed, only how the tasks provided by the application is handled. This in turn mean that a direct comparison to the interpretations of the Monolithic approaches above is only possible in the case of a runtime Monolith. However, while this Microservices definition is confined to the application's runtime, most often it is beneficial to consider the other two domains as well, since they are logical conclusions of a Microservices mindset. While it may be possible to deploy multiple instances of the entire application to different node, and have each instance only expose a subset of its functionality at its node, it is not exactly the most efficient use of resources. Therefore, each of these three facets will be considered as we continue on in this essay. [1]

3 Microservices

The previous section was split up into 3 main topics, each describing the Monolithic approach at different stages of the software development life cycle. The structure of this section will mimic this structure as well, as to keep the differ-

ences between the two approaches clear and to the point. And just to reiterate, the Microservices architectural model separates the different functionalities of an application into standalone modular services that communicates with each other over well defined channels to accomplish the same tasks a monolithic application would. [7]

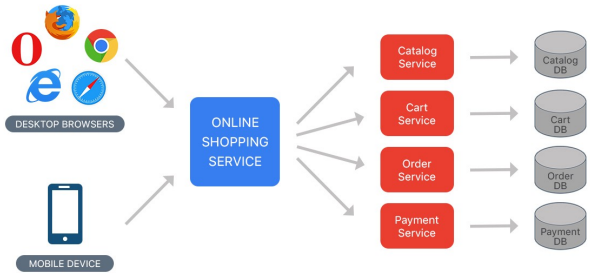

Figure 4: Illustration of a simple application implementing the Microservices architecture [7]

3.1 Module - Microservices

So let's start at the core of the application, the code. Since the essence of the Microservices architecture relies on the modulation of its different functionalities, there is no longer a need to write and compile the whole project in one fell swoop. This approach instead encourages the developers to split up into highly independent teams, each in charge of their own models, with the authority to themselves decide which language and framework to use, technologies to implement, etc. All in all, this option gives the developers a highly flexible environment where they are free to solve the problem at hand as they wish. [3] [5]

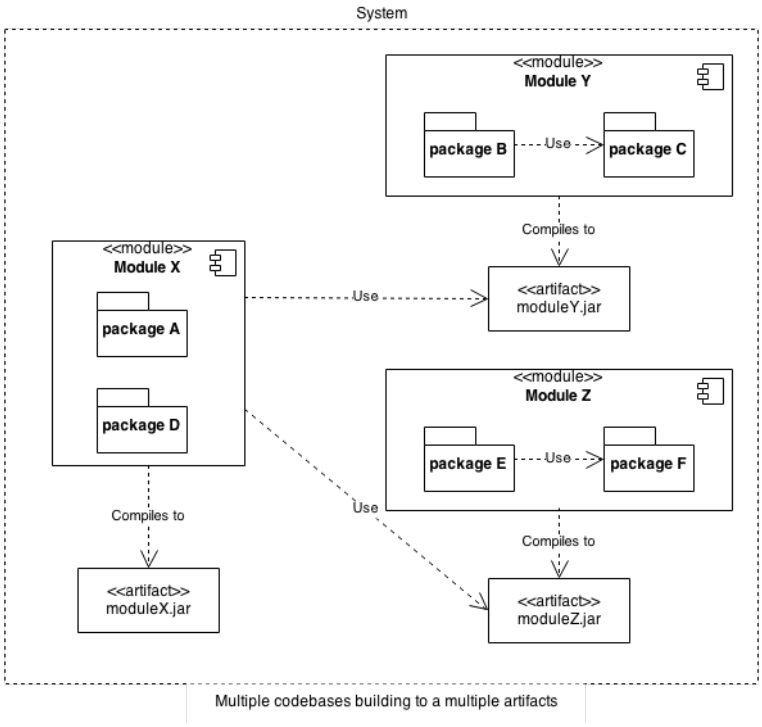

Figure 5: Illustration of a modular division of the codebase [1]

3.2 Allocation - Microservices

The difference for this section of the development process is quite simple. Since, each service runs independently, each one will have its own version, which can be deployed at the will of its team without disrupting the rest of the application. There is really no need to shut down the entire application each time one of the services needs an update because of this, instead only the modified components need to be redeployed. Since each module in itself is smaller than a Monolithic application would be, the time it takes to deploy a single module is substantially smaller as well. [5] [7] [2]

3.3 Runtime - Microservices

Now we have reached the core idea behind the Microservices structure, to expose the different functionalities, or services, in the form of independent pieces of software that each are designed for one specific task in mind, as opposed to one large application dealing with the entirety of the functionality. These services communicate with each other over API:s, and one important aspect to note is that each of the services has its own database, to ensure that they are loosely coupled from each other, and so changes in one service will not affect the others.

[6]

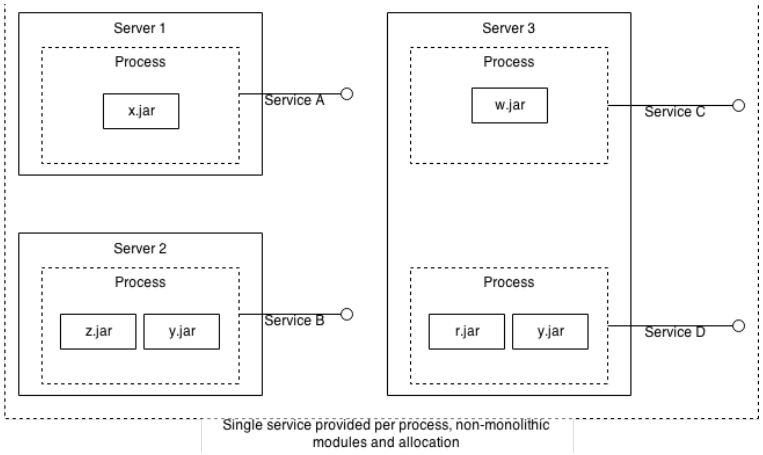

Figure 6: Illustration of a Microservices runtime [1]

4 Pros and cons

This section will go through the benefits and detriments of each of the two models, and it will be split up into 3 sections, one for different parts of the lifecycle of an application. It will explore the differences in the development process, i.e. the actual coding of the application, the deployment of the piece of software, as well as the maintainability of the application at runtime.

4.1 Development

Let's first view the Monolithic application from a developer's point of view. Since there is no need to consider how the different components of your program have to communicate with each other (as opposed to the additional complexity the Microservices architecture introduces just by the sheer fact of being a distributed system), it is quite easy to get up and running. Dividing up the program into highly cohesive and loosely coupled modules is still possible, with the difference that the boundaries between them is not as clear as they would have been if they were developed independently. Implementation of functionalities spanning multiple modules, as well as the testing of them, is less demanding than it would have been in the Microservices equivalent environment. The same also holds for end-to-end testing. [4][7]

However, the cracks start to show once the size of the program reaches a certain size, and the codebase grows larger. It can be close to, if not even completely, impossible for one single individual to understand how it all fits together, not to speak of the daunting experience newcomers to the team who looks at it with fresh eyes face. The result is that valuable time that could have been

spent improving the product, instead is spent at understanding the context of the code. Declines in code quality often occurs as well, as a result of developers not properly understanding how changes should be implemented correctly. The Microservices model combat these issues by separating the large application into smaller, more manageable services, which is often soon followed by a restructured development team, introducing smaller groups, each with complete autonomy and authority to develop one or more of the services independently from the rest of the company. One important note to make is that this division allows each individual service to be iterated on, and code can go through the DevOps pipeline on a smaller, service per service basis. What this means is that the introduction of the Microservices architecture allows for large and complex applications to adopt some of the teachings of DevOps. This division also makes it possible to identify errors in the application more easily, since, in general, only one service would be affected by it, while the others are running normally. The working environment for the developers may also be improved, since they no longer are held back by the shortcomings and dependencies of other services, and can instead develop their component with the sole intention to solve a singular task, and freely choose technologies accordingly. Had the application been Monolithic, these decisions would have been made on a global level instead. Given the scenario that a team would like to completely rewrite their service due to improvements made in the field, and the current dependencies no longer being the optimal choices, a process that would most likely be infeasible given a Monolithic application due to the different modules being dependent of each other, is instead made possible. Last, but not least, when new functionalities are to be incorporated into the application, instead of growing out the Monolith even further, exacerbating all the problems such a piece of software may encounter, the Microservices equivalent application quite simply need to create a new service. [4][5][3]

4.2 Deployment

Once again, the Monolithic approach is quite simple, as it is only a single artifact that needs to be deployed, nothing more, nothing less. The same can be said when it comes to scaling the application, simply deploy multiple copies of the entire Monolith and balance the number of requests to each one. But what happens when the load is focused on a subset of the application's functionalities, while others are left largely idle, or the entire program needs to be taken offline because of an error in a specific function? In the first scenario, there is not much to be done. Since the entire Monolith is only scalable horizontally, our only option is to waste resources on scaling the entire program, instead of only the parts that actually needs it. Also, if this increased demand was due to a peak, scaling out the entire application may be time consuming, and the peak may have subsided before we even see the effects of it. In the case of the second situation, a similar story will be told, since the option to redeploy only the faulty service is unavailable. Of course, there are drawbacks to the Microservices deployment as well. Once again, the nature of the architecture carries with it an

inherent increase in complexity, and the deployment and operational cycles are not left unscathed. The resource consumption is not strictly improved either, as this approach introduces some extra overhead due to the fact that **X** application instances are replaced by **X*Y** service instances, each of which running in their own isolated environment. [4][5][3]

5 Conclusions

So, with all that said, which one of the two approaches is superior? The boring answer sadly is that neither one is strictly better than the other, which one you as a developer should use is entirely dependent on the situation. If the application is developed by a small team of just a few people, and you do not suspect you will see major benefits from being able to utilize the Microservices architecture to its fullest potential, then the Monolithic approach is most likely just fine. If your codebase is getting out of hand, and you start to see major dips in performance due to scaling issues, and every little detail matters, it might be time to consider if it is time for a switch of structure. It is worth noting also, that implementing the Microservices architecture is not trivial, and a tidbit of technical expertise is required.

5.1 DevOps

I mentioned this briefly earlier, but the main reason that this topic is vital from a DevOps point of view, is the fact that the approach allows for complex, large-scale applications to be developed according to DevOps directives.

References

[1] Annett, R. (2014). *"What is a Monolith?"*. Viewed apr 2020 at `http://www.codingthearchitecture.com/2014/11/19/what_is_a_monolith.html`

[2] Fowler, M & Lewis, J. (2014) *"Microservices a definition of this new architectural term"* Viewed apr 2020 at `https://martinfowler.com/articles/microservices.html`

[3] IBM Cloud. (2019). *"What are Microservices?"*[Video file]. Retrieved from `https://www.youtube.com/watch?v=CdBtNQZH8a4&t`

[4] Richardson, C. (nd). *"Pattern: Monolithic Architecture"*.Viewed apr 2020 at `https://microservices.io/patterns/monolithic.html`

[5] Richardson, C. (nd). *"Pattern: Microservice Architecture"*. Viewed apr 2020 at `https://microservices.io/patterns/microservices.html`

[6] Richardson, C. (nd). *"Pattern: Database per service"*.Viewed apr 2020 at `https://microservices.io/patterns/data/database-per-service.html`

[7] ul Haq, S. (2018). *"Introduction to Monolithic Architecture and MicroServices Architecture"*.Viewed apr 2020 at `https://medium.com/koderlabs/introduction-to-monolithic-architecture-and-microservices-architecture-b211a5955c63`