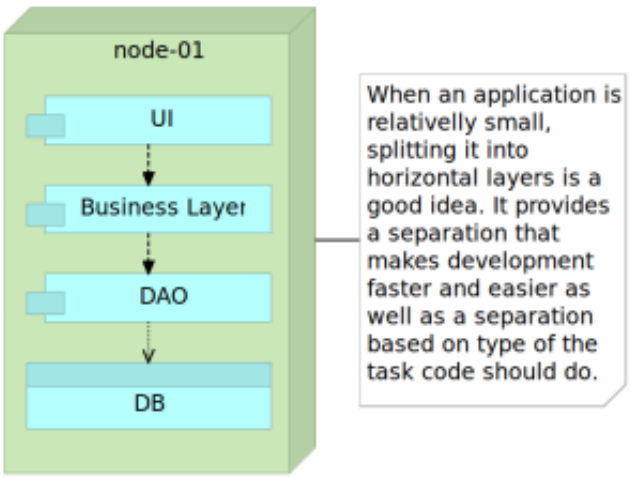
Technology Conversations

Microservices: The Essential Practices

Before we jump and try to explore the practices we must master in order to successfully implement microservices architecture, let us briefly refresh our understanding of monolithic applications.

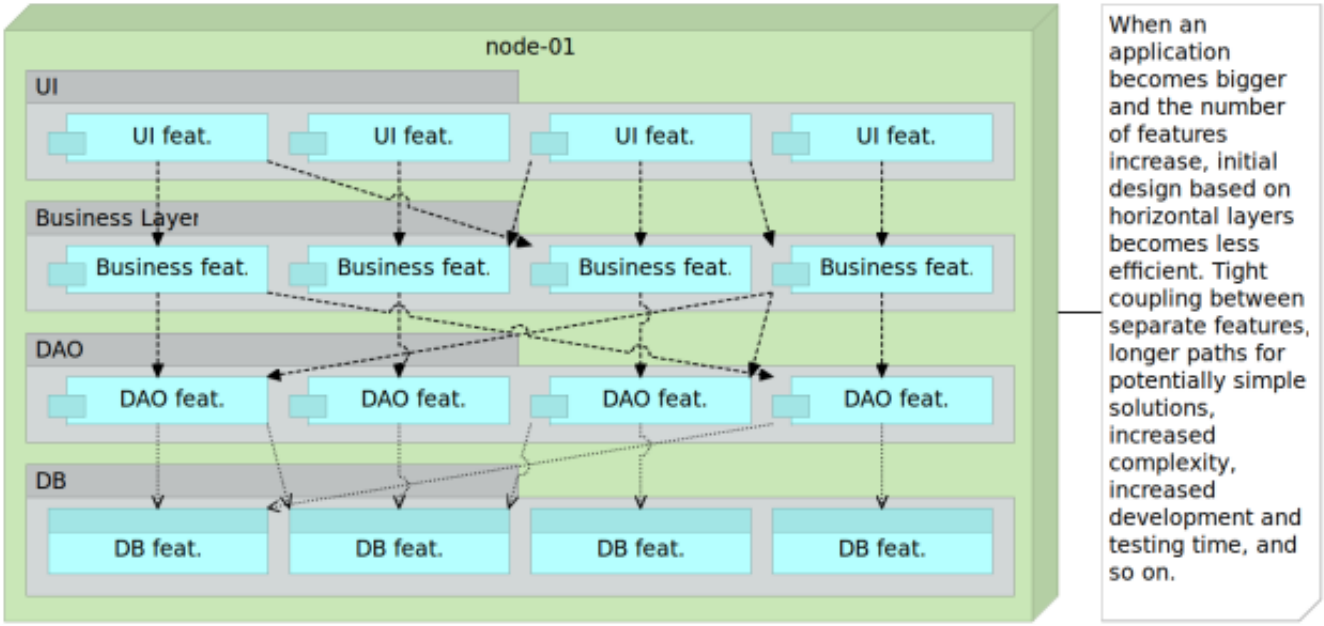
Monolithic Applications



Monolithic application is built as a single unit that, with time, increases its size. While the increase in features is bound to result in increase of the system's complexity, monolithic architecture tends to exponentially multiply that complexity. The reason behind that is partly due to layering approach that tends to be enforced to all use cases. You start with a design that has, let's say, four layers; API, business layer (BL), data access layer (DAL) and database itself. Once that is established, you make a rule that all new features should be developed in a way that all the layers are used. It makes perfect sense. Feature one receives a request through the API layer, that passes to the business layer which in turn goes to the data access layer so that it reaches the database. Once we reach the bottom, the direction changes so that the response is produced. Database returns some data to the DAL, DAL passes it to BL and from there is sent to API that generates the response. It sounds logical (for a while).

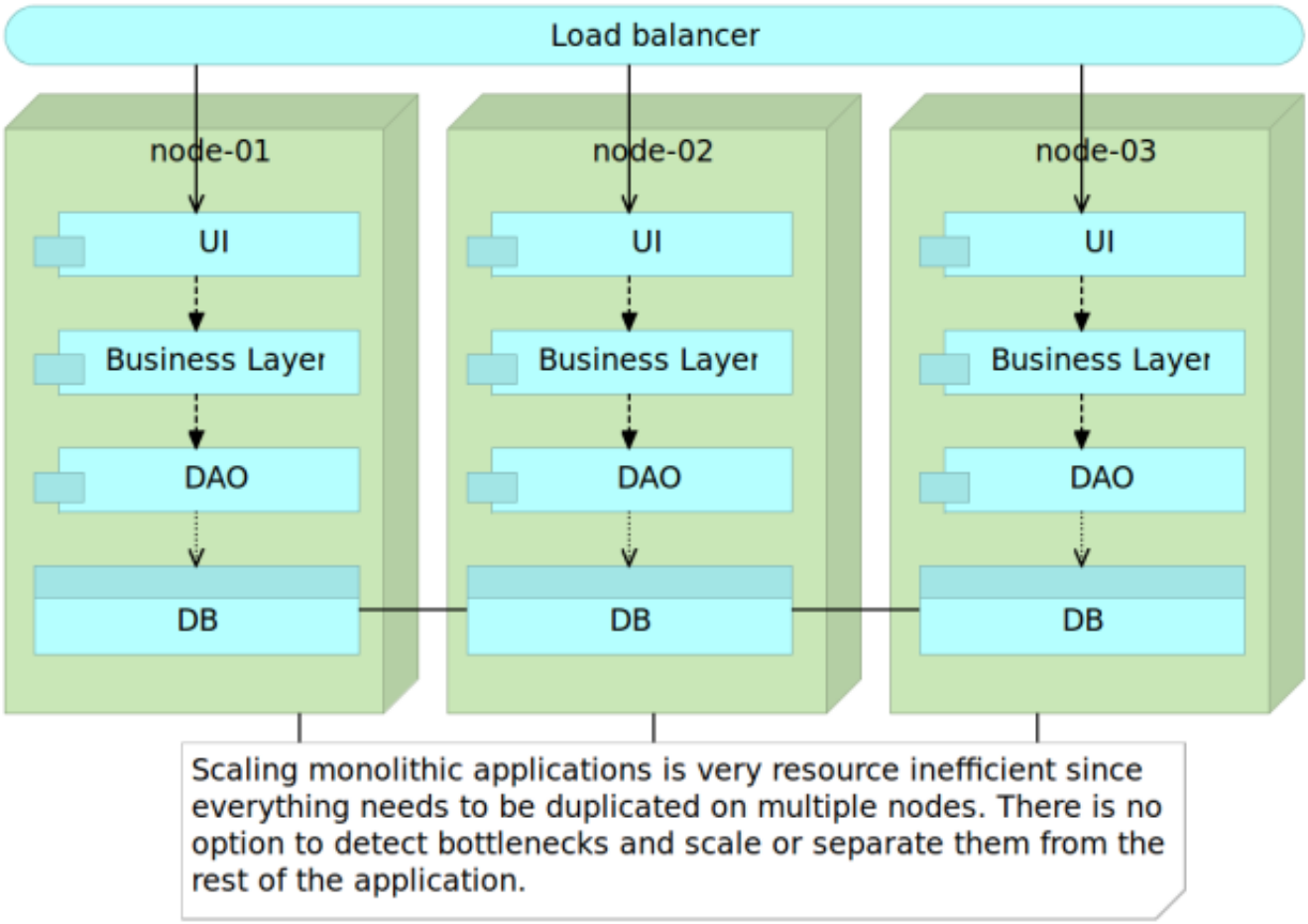
More and more features are developed that way but with time things stop being linear. The direction soon stops being purely vertical and we start having diagonal relations. A single API class starts calling multiple BL classes which in turn might need to invoke several DAL classes and deal with multiple database entities. Communication starts being not only vertical but also diagonal. More and more methods from multiple classes are invoked making the code more and more coupled.

Many of you will say at this point that this happens because there is no well established governance and that it can all be fixed with better code reviews. Sometimes it can be fixed. More often it can be made less coupled. People start taking shortcuts. It is in our nature to do that especially when faced with tight deadlines.



With each new feature, complexity of the monolithic application increases and with it development, testing and deployment speed decreases. It is inevitable. The bigger the code base, the more time one needs to spend in development since the number of dependencies increases as well as the number of things we need to take into account during the development. Similar statement is valid for testing and deployment. The bigger the application, the more time we need to spend testing it every time there is a change. That is one of the reasons why some companies struggle with Agile methodologies. The cost of continuously testing the application is too high so testing is left for the end (with often disastrous results). If testing is manual, the bigger the application, the more man-hours need to be invested. On the other hand, if testing is automated, the bigger the application the bigger the chance that there will be flaky tests that fail due to reasons others than bugs in the code (too many dependencies, too much data to be set, and so on). Architecture is the key to implementation of most eXtreme programming practices like continuous integration, test-driven development, short development cycles, and so on.

The result is the increase in difficulties we are facing when trying apply changes. Developing a new feature takes more and more time. Refactoring starts being less and less practiced due to potential risks that things will get broken, hence the famous and often repeated phrase; if it works, do not touch it. Something we could develop in a day, takes a week. Business starts being more and more affected because it needs scaling. It needs new features, it needs bigger teams and it needs to be able to increase the usage of the system. However, the result is opposite. It takes more time to develop a feature, bigger teams are less efficient than smaller ones and scaling the application is not efficient if the only option we have is to multiply everything. Multiplying the number of servers big application runs on is equivalent to failing high school. Even though you flanked one or two subjects and would benefit by spending more time learning them, you have to repeat the whole year[1]. As a result, you spend two years attending classes of all subjects even though your problem is only math. The correct way to handle this situation would be to take extra classes in math. Scale classes in which your performance is not adequate.



Microservices, Practices and Tools

Microservices are all about having small and specialized applications. They should be autonomous with only a single entrance through the API. They should be decoupled from each others (with the exception of APIs). This decoupling is accomplished through a physical separation. One cannot invoke classes and methods or functions from other services if they are physically separated. Finally, they are organized vertically. They represent a business feature or entity and not a technical one (as is the case of horizontal layers). For more information about advantages and disadvantages, please consult the Monolithic Servers vs Microservices article. The purpose of this post is to explore practices and tools that are essential for a successful application of microservices architecture.

Continuous Delivery or Deployment (CD)

With microservices comes decreased development cycle resulting is shorter time-to-market. It takes less time to develop, test and deploy a microservice than equivalent feature(s) inside a monolithic application. If you want to fully profit from this increase in speed, it is essential that there are no bottlenecks in the process and CD is probably the best path we can take so that we work in a way that there is nothing stopping us from releasing to production as soon as possible. I intentionally skipped putting continuous integration (CI) in this group since, even though it does increase the speed, it requires a manual phase at the end. Besides, CD will prove that you are worthy the challenge since all other practices will require high level of automation as well.

Containers

Without containers, your environments will quickly become a mess. Switching from one single (huge) application to deployments of many small services means that the amount of things that needs to be handled increases exponentially. While you might have been fine with a single applications server and few runtime dependencies, microservices, when done with the freedom they enable, will increase those dependencies drastically. One service might use JDK 7 while the other could benefit from streaming features introduced in JDK 8. One might have only static files handled with nginx while the other could require an application server that should serve dynamic content. One could benefit from relational database while the other could be better of with ElasticSearch. This is the type of freedom we want to accomplish with microservices. The freedom to choose the best tool, framework, programming language and what-so-not, for the task at hand. It's a huge advantage that comes with the price in form of increased operation costs. That price, until recently, was too big for most organizations. With the emergence of Docker, problem has been (mostly) solved. Don't install anything on your servers (besides Docker daemon). Instead, package and run your fully self-sufficient services as containers. Inside those containers should be everything. Applications server, dependencies and runtime libraries, compiled code, configurations and so on. All we have to do is run services packed as containers and make sure that they can communicate with each other (more on that subject later on).

Configuration Management (CM)

Some claim that there is no need for configuration management if all deployments are done with Docker. I beg to differ. CM is still an important and indispensable tool. However, the scope of tasks that CM should do is much smaller than it was before. For example, deployments should not be in the CM hands any more. We have better tools for that ranging from simple ones like Docker Compose to more complex ones like Kubernetes, Docker Swarm and Mesos. No matter the choice, CMs can, at best, only run the command that will execute one of those tools. The real scope of CMs should be making sure that very low-level things are working properly. Is your OS updated? Are all system users created? Do we have firewalls set up correctly? Due to this change in scope, different CM tools are needed. Do we need client-server pull model used by Chef and Puppet? No we don't. Do we need complicated Ruby syntax? No we don't. Actually, I don't think that those tools are good even without Docker. They were great before but not any more. So what is the alternative? Ansible is. It is as good in some areas as Chef and Puppet and better in others. SSH Push system is a much better principle than pull. It makes me wonder how come that no one thought of it before since SSH is with us from the dawn of time. Long story short, CM is a must and Ansible is the best tool of that kind we can find today. With the recent acquisition by RedHat, Ansible is bound to be adopted even by traditional enterprise organizations.

I almost forgot to mention that the task of managing application configurations should also be out of the scope of CM tools. How should we manage them? The key words are: service discovery.

Service Discovery

Managing application configurations was always a though thing to do. Even with the appearance of CM tools and their promise to solve that problem, we are still struggling. Now more than ever since we are not expected any more to have (more or less) static configurations. Configuration is becoming more and more dynamic. Take scaling as an example. If the traffic increases we should increase the number of nodes running our services. Once the traffic decreases, we should scale down and let nodes do something else. This type of dynamism requires equally flexible re-configurations which are hardly possible and certainly too complicated and time demanding with CM tools. That's where service discovery comes in. Instead of defining things in advance (service X will run on servers Y and Z), we just need to make sure that each service announces its existence whenever deployed. That announcement is stored in some kind of registry and available to any other service that might need it. It's a relatively simple process supported by ever-increasing number of tools. Register yourself or have a process that will detect your existence, store that data in highly available and distributed registry that can be queried by anybody. Registrator is a great solution that can easily be combined with registries like etcd or Consul. Throw into the mix tools like confd or Consul Template and suddenly all the configurations are stored, available to whomever needs them and proper processes and configuration files are created. The best part is that everything is automatic and works in any size, be it one, hundred or thousand servers with services being constantly moved around them.

Finally, the service most affected by such a dynamic setting is proxy.

Proxy Service

Do not even try to make microservices speak with each others directly. If you're already familiar with Docker, you might think that using links is a great idea for all cases. In some it is and in most it isn't. With the networking improvements Docker introduced in version 1.9, linking is more attractive than before but that still does not remove the need for a proxy. Setup an nginx or HAProxy service and make sure that configurations are updated every time a new service is deployed. Service discovery tools will do that for you and the proxy will always redirect all requests to any number of instances of a destination service. All that's left is to develop services in the way that they (almost) always make requests to the proxy.

With that out-of-the-way, you can start aiming for zero-downtime deployments.

Blue-Green Deployments

Blue-green deployment is another one of those things that are in use for a long time but only recently become truly easy to do. The gist of the idea is to deploy a new release in parallel with the current one, test it until you're sure that everything works as expected and simply change the proxy to point to the new release. At no point there will be down-time (at least not caused by deployments). Why was this hard before and now it's easy? Because before we had monolithic applications occupying whole servers so we needed to double the capacity. Because before we had to pre-configure everything. With microservices that tend to have low usage of resources and service

discovery that ends up updating the proxy service, blue-green deployment is easy to accomplish and won't break your budget since both releases of microservices can briefly run on the same server.

Speaking of a budget, clustering and scaling is also becoming cheaper than ever.

Clustering and Scaling

Most organizations I worked for had a lot of wasted resources and yet were in constant need for more and more servers. For example, they could have five servers dedicated to one huge application and two more to batch tasks run at night. That separation might seem convenient but one could easily discover that during day-time application servers were (more or less) fully occupied while batch servers were mostly idle. At night-time the situation would be reversed. On top of that, application servers would, when faced with heavy traffic, have almost fully occupied memory but CPU would never reach more than 50%. Batch process, on the other hand could be a heavy user of CPU and not require much memory. This is a very simplified image but the point is that if each server is dedicated to a pre-defined purpose, its resources are partly wasted. The reason behind that is that most of us are used to having associations that relate servers with their purposes. We even give them names. Oh, yes, that application runs on Gandalf (name of the server) while that one is installed on Mordor (another name of the server). However, curious thing is that we do not do the same with, let's say, CPU. We do not specify that certain process should run on the third CPU. Another example is Java. We might specify minimum and maximum memory for a program running on JDK but (special cases excluded) we do not say what is the exact amount of memory it should use at any given moment.

We should change the way we manage servers and treat all of them as a single data center or a server farm. We should think of the sum of everything and depending on that information decide where to deploy. Luckily for us, there are tools that do just that. Kubernetes, Mesosphere DCOS and Docker Swarm are only a few of the tools we have at our disposal. With them we can deploy our services and applications not to a particular server but somewhere inside the data center where there is enough resources to run it. The decision is made based on the number of containers, available memory, type of hard disk and many other combinations. Those decisions should be reevaluated constantly so that depending on the current or predicted necessities, services are continuously scaled and de-scaled. Once such a system is in place, not only that we gain flexibility that we haven't had before but our infrastructure costs can drop drastically.

One of the elements missing in order to get there are health checks and self-healing systems.

Self-Healing Systems

Most of use have some form of health checks. On the application level we tend to catch exceptions and react to the problems as they happen. You might even have applications that apply some of the newer concepts like, for example, actors with Akka that tend to provide new and better ways to handle failures and, more importantly, recuperate from them. Second level of checks is normally on

a system level where we tend to continuously verify whether applications are up and running. Finally, there is a hardware level checking that allows us to monitor resources like memory, CPU, hard disks and so on. One of the common tools for the second and third level of checking is Nagios. There are many others like it and most of them share two major problems. First of them is that they are big and cumbersome and can be replaced with something much lighter that, at the same time, utilizes information from the service registry. Second, and much more important problem, is that tools like Nagios tend to simply monitor and notify about potential problems. Below them we have a set of operators that react to those notifications. A server goes down and they start running around and executing some procedures that will put the system back into the correct state. Problem with human operators is that they are slow. It takes time to react and fix the problem. On the other hand, if previous advices are implemented, all the tools that should fix the problem are already in place. If for whatever reason a service goes down (be it because the process died or the whole server is down), the monitoring system should not only detect the failure but also re-run the deployment through, for example, Docker Swarm which, in turn, will place it somewhere inside the data center. In most cases, operators should simply receive a notification saying something like "Something went wrong and was fixed. Continue playing Solitaire." Sounds like science fiction? Well, it isn't. If you reconsider all the tools we spoke about, there's nothing that would prevent us from doing this. Ping a service every 5 seconds and if there is no response or the response is not 200, initiate the deployment. Monitor response time through those pings and if it is longer than 500 milliseconds, scale up. Tools are there and we just need to let them do the work. Monitoring tool that does those pings can, for example, easily be Consul itself (we already spoke about it). We could employ its health checks and run deployments when some status is critical. You could have a self-healing system that is both reactive (acts after a failure) and preventive (acts when some thresholds are reached). Tools like Mesos and Kubernetes have failover strategies already included in the package.

Summary

The short version of this article is as follows.

- Package your microservices into containers with Docker or, once it's production ready, rkt.
- Don't provision environments manually. Employ CM tools like Ansible to do that for you.
- Don't configure applications manually nor with CM tools. Use service discovery with combinations like Consul, Registrator and Consul Template or etcd, Registrator and confd.
- Use proxy services like nginx or HAProxy as the (almost) only way to make requests (be it from outside or from one of your services to another).
- Avoid downtime produced by deployments by employing blue-green procedure.
- Treat all your servers as one big entity. Use tools like Kubernetes, Mesosphere DCOS and Docker Swarm to deploy services.
- Don't spend your career watching dashboard of some monitoring tool. Set up a self-healing system that will re-initiate deployments when things go wrong or some threshold is reached.

I know that all of those things might sound very intimidating but once they are up and running you can spend your time on the things that bring real value to your organization. You can dedicate time

and money to development of features that bring business value and let the system take over everything that follows after you commit your code.

1. Schools might not work in the same way in your country. If that is the case, just imagine that failing a single subject leads to a repetition of the whole year. ↩

This entry was posted in Architecture, Docker and tagged Ansible, CM, confd, Configuration Management, Consul, consul-template, Container, DCOS, Docker, Docker Swarm, etcd, HAProxy, Kubernetes, Mesosphere, nginx, Registrator, Service Discovery on November 10, 2015 [http://technologyconversations.com/2015/11/10/microservices-the-essential-practices/] by Viktor Farcic.

3 thoughts on “Microservices: The Essential Practices”

Pingback: Web Operations Weekly No.40 | ENUE Blog

Pingback: Google Cardboard pour les développeurs en vidéo

Pingback: 12-11-2015 - Links - Magnus Udbjørg