Exame de Arquitecturas Aplicacionais

MEI

22/06/2022
Duração máxima: 2h

Grupo I / Parte Teórica

1. Considere o artigo *Monolith First*, de Martin Fowler, que está anexo a este enunciado. Enuncie qual é a sua opinião acerca da mensagem do artigo e, em função da sua análise, identifique aquelas que lhe parecem ser as vantagens da utilização de microsserviços. Em que situações é que vê riscos da utilização desta proposta aquitectural?

Grupo II / Parte Prática

2. Considere que se pretende elaborar uma solução para um sistema de informação para aeroportos (YAAIS - YetAnotherAirportInformationSystem). O objectivo primordial desse sistema de informação é a disponibilização de informação das chegadas e partidas, bem assim como possibilitar que os passageiros façam check-in de forma mais facilitada.

   São actores primordiais deste novo sistema de informação as companhias de aviação que promovem os voos (que partem e que chegam), os passageiros que se inscrevem no sistema para acederem à informação e todos os sites noticiosos que queiram disponibilizar esta informação nas suas páginas. Cada pessoa (utilizador) que se regista no sistema pode-se associar-se à página de cada uma das companhias aéreas, possibilitando desta forma que estas lhe possam enviar informação. Cada utilizador poderá consultar na página do seu perfil estas mensagens das companhias aéreas.

   A aplicação Web (YAAIS) deve funcionar com clientes web browser, web móvel e clientes nativos para iOS e Android. Deve estar preparada para lidar com grandes volumes de dados, de utilizadores, mensagens das companhias aéreas, etc.. Em termos de interoperabilidade com outros sistemas, é importante dizer que:

   (a) a empresa que vai criar esta aplicação fez um acordo com os principais fornecedores de aplicações de PIM (contactos, calendário e tarefas) para que estas aplicações (ex: Outlook, GoogleCalendar, etc.) possam alimentar os calendários com informação dos voos e do registo de check-in dos utilizadores nesses voos.

   A aplicação deverá permitir:

   - registar um novo utilizador
   - registar uma nova companhia de aviação
   - permitir a uma companhia enviar uma mensagem

- permitir a um utilizador verificar que mensagens é que lhe foram enviadas
- permitir a um utilizador ligar o seu PIM para que possa ser alimentado pelas companhias aéreas com informação sobre os voos

Responda às seguintes questões:

(a) desenvolva o diagrama de classe PSM, e eventualmente caso julgue útil o PIM, da solução que propõe. Seja o mais descritivo possível por forma a identificar os diferentes papéis das diferentes classes. Justifique a arquitectura computacional que apresenta e detalhe as decisões que tomou.

(b) como faria o deployment da solução? Quais são os componentes aplicacionais que utilizaria, o que está instalado em cada componente e como lidaria com o aumento da carga aplicacional?

MonolithFirst

3 June 2015

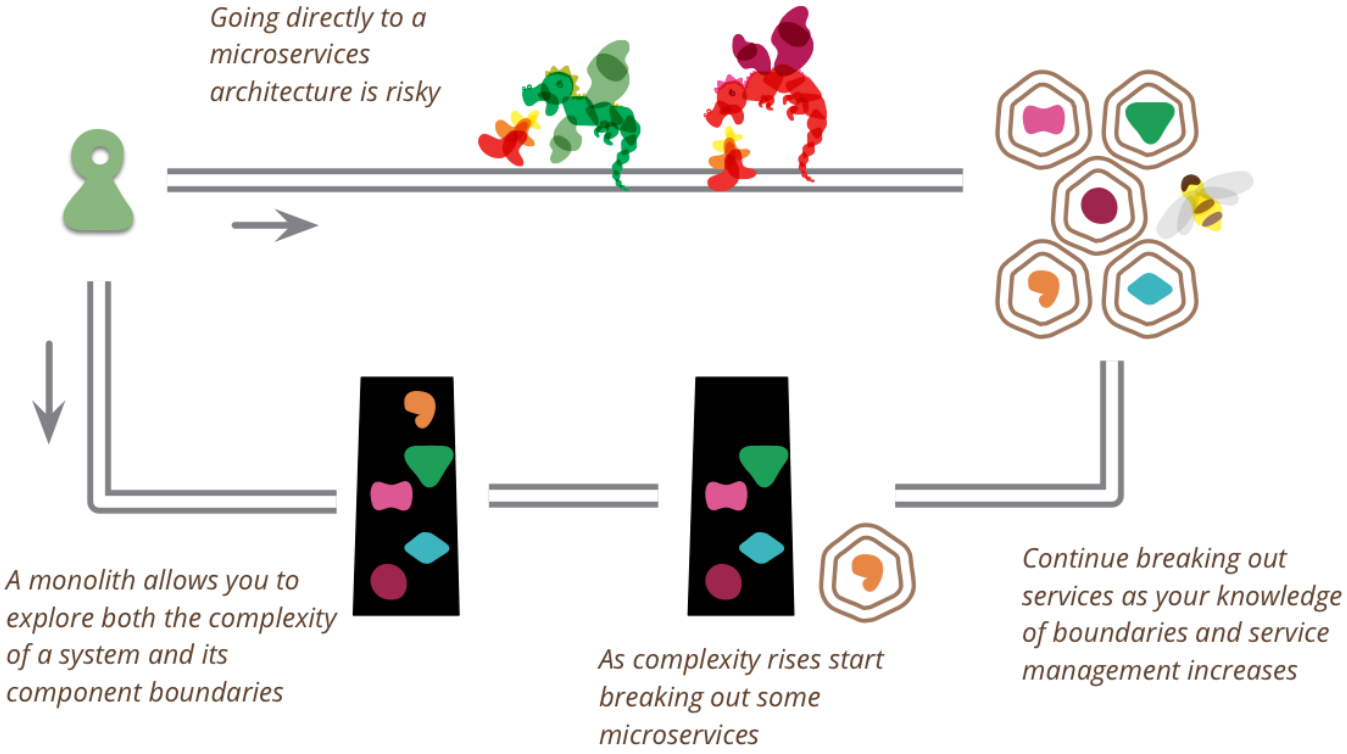
Martin Fowler

EVOLUTIONARY DESIGN
MICROSERVICES

As I hear stories about teams using a microservices architecture, I've noticed a common pattern.

1. Almost all the successful microservice stories have started with a monolith that got too big and was broken up
2. Almost all the cases where I've heard of a system that was built as a microservice system from scratch, it has ended up in serious trouble.

This pattern has led many of my colleagues to argue that **you shouldn't start a new project with microservices, even if you're sure your application will be big enough to make it worthwhile.** .



Microservices are a useful architecture, but even their advocates say that using them incurs a significant MicroservicePremium, which means they are only useful with more complex systems. This premium, essentially the cost of managing a suite of services, will slow down a team, favoring a monolith for simpler applications. This leads to a powerful argument for a monolith-first strategy, where you should build a new application as a monolith initially, even if you think it's likely that it will benefit from a microservices architecture later on.

The first reason for this is classic Yagni. When you begin a new application, how sure are you that it will be useful to your users? It may be hard to scale a poorly designed but successful software system, but that's still a better place to be than its inverse. As we're now recognizing, often the best way to find out if a software idea is useful is to build a simplistic version of it and see how well it works out. During this first phase you need to prioritize speed (and thus cycle time for feedback), so the premium of microservices is a drag you should do without.

The second issue with starting with microservices is that they only work well if you come up with good, stable boundaries between the services -

which is essentially the task of drawing up the right set of BoundedContexts. Any refactoring of functionality between services is much harder than it is in a monolith. But even experienced architects working in familiar domains have great difficulty getting boundaries right at the beginning. By building a monolith first, you can figure out what the right boundaries are, before a microservices design brushes a layer of treacle over them. It also gives you time to develop the MicroservicePrerequisites you need for finer-grained services.

I've heard different ways to execute a monolith-first strategy. The logical way is to design a monolith carefully, paying attention to modularity within the software, both at the API boundaries and how the data is stored. Do this well, and it's a relatively simple matter to make the shift to microservices. However I'd feel much more comfortable with this

approach if I'd heard a decent number of stories where it worked out that way. [1]

A more common approach is to start with a monolith and gradually peel off microservices at the edges. Such an approach can leave a substantial monolith at the heart of the microservices architecture, but with most new development occurring in the microservices while the monolith is relatively quiescent.

Another common approach is to just replace the monolith entirely. Few people look at this as an approach to be proud of, yet there are advantages to building a monolith as a SacrificialArchitecture. Don't be afraid of building a monolith that you will discard, particularly if a monolith can get you to market quickly.

Another route I've run into is to start with just a couple of coarse-grained services, larger than those you expect to end up with. Use these coarse-grained services to get used to working with multiple services, while enjoying the fact that such coarse granularity reduces the amount of inter-service refactoring you have to do. Then as boundaries stabilize, break down into finer-grained services. [2]

While the bulk of my contacts lean toward the monolith-first approach, it is by no means unanimous. The counter argument says that starting with microservices allows you to get used to the rhythm of developing in a microservice environment. It takes a lot, perhaps too much, discipline to build a monolith in a sufficiently modular way that it can be broken down into microservices easily. By starting with microservices you get everyone used to developing in separate small teams from the beginning, and having teams separated by service boundaries makes it much easier to scale up the development effort when you need to. This is especially viable for system replacements where you have a better chance of coming up with stable-enough boundaries early. Although the evidence is sparse, I feel that you shouldn't start with microservices unless you have reasonable experience of building a microservices system in the team.

I don't feel I have enough anecdotes yet to get a firm handle on how to decide whether to use a monolith-first strategy. These are early days in microservices, and there are relatively few anecdotes to learn from. So anybody's advice on these topics must be seen as tentative, however confidently they argue.

Further Reading

Sam Newman describes a case study of a team considering using microservices on a greenfield project.

Notes

1: You cannot assume that you can take an arbitrary system and break it into microservices. Most systems acquire too many dependencies between their modules, and thus can't be sensibly broken apart. I've heard of plenty of cases where an attempt to decompose a monolith has quickly ended up in a mess. I've also heard of a few cases where a gradual route to microservices has been successful - but these cases required a relatively good modular design to start with.

2: I suppose that strictly you should call this a "duolith", but I think the approach follows the essence of monolith-first strategy: start with coarse-granularity to gain knowledge and split later.

Acknowledgements

I stole much of this thinking from my coleagues: James Lewis, Sam Newman, Thiyagu Palanisamy, and Evan Bottcher. Stefan Tilkov's comments on an earlier draft played a pivotal role in clarifying my thoughts. Chad Currie created the lovely glyphy dragons. Steven Lowe, Patrick Kua, Jean Robert D'amore, Chelsea Komlo, Ashok Subramanian,

Dan Siwiec, Prasanna Pendse, Kief Morris, Chris Ford, and Florian Sellmayr discussed drafts on our internal mailing list.