# Some Hints:

1. You should **delete** all content in this page;
2. **Rename** this document using such format:

”Your student ID---your name---you topic title”, for example:

“201600000000---Ding---Amazon cloud architecture”

<https://www.martinfowler.com/articles/microservices.html>

# Topics of interests (not limited to the following topics)

1. Some research direction about software architecture
2. some new software architecture styles;
3. Amazon cloud architecture
4. Google distributed system
5. Big data management
6. Micro-software Architecture
7. …

**Course Report of Software Architecture and Design Pattern**



|  |  |
| --- | --- |
| **Title：** | **Microservices architecture** |
|  |  |
| **Student Number：** | **201736900125** |
| **Student Name：** | **伍泰炜** |
| **Final Score：** |  |
| **Review Date：** |  |
| **Supervisor Signature：** |  |

**Microservices architecture**

**Abstract.** The main aim in writing this article is to explain the major ideas and principles of microservices. By taking the time to do this we clearly think that the microservices architectural style is an important idea - one worth serious consideration for enterprise applications. We have recently built several systems using the style and know of others who have used and favor this approach.

**Keywords: Microservice Future components**

**1 What is** **Microservice 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, often an HTTP resource API. 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.

**1 Microservices**

"Microservices" - yet another new term on the crowded streets of software architecture. Although our natural inclination is to pass such things by with a contemptuous glance, this bit of terminology describes a style of software systems that we are finding more and more appealing.

To start explaining the microservice style it's useful to compare it to the monolithic style: a monolithic application built as a single unit. Enterprise Applications are often built in three main parts: a client-side user interface (consisting of HTML pages and javascript running in a browser on the user's machine) a database (consisting of many tables inserted into a common, and usually relational, database management system), and a server-side application. The server-side application will handle HTTP requests, execute domain logic, retrieve and update data from the database, and select and populate HTML views to be sent to the browser. This server-side application is a *monolith* - a single logical executable. Any changes to the system involve building and deploying a new version of the server-side application.

Such a monolithic server is a natural way to approach building such a system. All your logic for handling a request runs in a single process, allowing you to use the basic features of your language to divide up the application into classes, functions, and namespaces. With some care, you can run and test the application on a developer's laptop, and use a deployment pipeline to ensure that changes are properly tested and deployed into production. You can horizontally scale the monolith by running many instances behind a load-balancer.

Monolithic applications can be successful, but increasingly people are feeling frustrations with them - especially as more applications are being deployed to the cloud . Change cycles are tied together - a change made to a small part of the application, requires the entire monolith to be rebuilt and deployed. Over time it's often hard to keep a good modular structure, making it harder to keep changes that ought to only affect one module within that module. Scaling requires scaling of the entire application rather than parts of it that require greater resource.

**2 Characteristics of a Microservice Architecture**

We cannot say there is a formal definition of the microservices architectural style, but we can attempt to describe what we see as common characteristics for architectures that fit the label. As with any definition that outlines common characteristics, not all microservice architectures have all the characteristics, but we do expect that most microservice architectures exhibit most characteristics. While we authors have been active members of this rather loose community, our intention is to attempt a description of what we see in our own work and in similar efforts by teams we know of. In particular we are not laying down some definition to conform to.

1. **Componentization via Services**

Microservice architectures will use libraries, but their primary way of componentizing their own software is by breaking down into services. We define **libraries** as components that are linked into a program and called using in-memory function calls, while **services** are out-of-process components who communicate with a mechanism such as a web service request, or remote procedure call.

One main reason for using services as components (rather than libraries) is that services are independently deployable. If you have an application that consists of a multiple libraries in a single process, a change to any single component results in having to redeploy the entire application. But if that application is decomposed into multiple services, you can expect many single service changes to only require that service to be redeployed. That's not an absolute, some changes will change service interfaces resulting in some coordination, but the aim of a good microservice architecture is to minimize these through cohesive service boundaries and evolution mechanisms in the service contracts.

2. **Organized around Business Capabilities**

When looking to split a large application into parts, often management focuses on the technology layer, leading to UI teams, server-side logic teams, and database teams. When teams are separated along these lines, even simple changes can lead to a cross-team project taking time and budgetary approval. A smart team will optimise around this and plump for the lesser of two evils - just force the logic into whichever application they have access to. Logic everywhere in other words. This is an example of Conway's Law in action.

The microservice approach to division is different, splitting up into services organized around **business capability**. Such services take a broad-stack implementation of software for that business area, including user-interface, persistant storage, and any external collaborations. Consequently the teams are cross-functional, including the full range of skills required for the development: user-experience, database, and project management.

3. **Products not Projects**

Most application development efforts that we see use a project model: where the aim is to deliver some piece of software which is then considered to be completed. On completion the software is handed over to a maintenance organization and the project team that built it is disbanded.

Microservice proponents tend to avoid this model, preferring instead the notion that a team should own a product over its full lifetime. A common inspiration for this is Amazon's notion of "you build, you run it" where a development team takes full responsibility for the software in production. This brings developers into day-to-day contact with how their software behaves in production and increases contact with their users, as they have to take on at least some of the support burden.

4. **Smart endpoints and dumb pipes**

The microservice community favours one approach: *smart endpoints and dumb pipes*. Applications built from microservices aim to be as decoupled and as cohesive as possible - they own their own domain logic and act more as filters in the classical Unix sense - receiving a request, applying logic as appropriate and producing a response. These are choreographed using simple RESTish protocols rather than complex protocols such as WS-Choreography or BPEL or orchestration by a central tool.

The two protocols used most commonly are HTTP request-response with resource API's and lightweight messaging.

5. **Decentralized Governance**

Teams building microservices prefer a different approach to standards too. Rather than use a set of defined standards written down somewhere on paper they prefer the idea of producing useful tools that other developers can use to solve similar problems to the ones they are facing. These tools are usually harvested from implementations and shared with a wider group, sometimes, but not exclusively using an internal open source model. Now that git and github have become the de facto version control system of choice, open source practices are becoming more and more common in-house .

Netflix is a good example of an organisation that follows this philosophy. Sharing useful and, above all, battle-tested code as libraries encourages other developers to solve similar problems in similar ways yet leaves the door open to picking a different approach if required. Shared libraries tend to be focused on common problems of data storage, inter-process communication and as we discuss further below, infrastructure automation.

6. **Decentralized Data Management**

Decentralization of data management presents in a number of different ways. At the most abstract level, it means that the conceptual model of the world will differ between systems. This is a common issue when integrating across a large enterprise, the sales view of a customer will differ from the support view. Some things that are called customers in the sales view may not appear at all in the support view. Those that do may have different attributes and (worse) common attributes with subtly different semantics.

This issue is common between applications, but can also occur *within* applications, particular when that application is divided into separate components. A useful way of thinking about this is the Domain-Driven Design notion of Bounded Context. DDD divides a complex domain up into multiple bounded contexts and maps out the relationships between them. This process is useful for both monolithic and microservice architectures, but there is a natural correlation between service and context boundaries that helps clarify, and as we describe in the section on business capabilities, reinforce the separations.

7. **Infrastructure Automation**

Infrastructure automation techniques have evolved enormously over the last few years - the evolution of the cloud and AWS in particular has reduced the operational complexity of building, deploying and operating microservices.

It turns out that once you have invested in automating the path to production for a monolith, then deploying *more* applications doesn't seem so scary any more. Remember, one of the aims of CD is to make deployment boring, so whether its one or three applications, as long as its still boring it doesn't matter.

8. **Design for failure**

Since services can fail at any time, it's important to be able to detect the failures quickly and, if possible, automatically restore service. Microservice applications put a lot of emphasis on real-time monitoring of the application, checking both architectural elements (how many requests per second is the database getting) and business relevant metrics (such as how many orders per minute are received). Semantic monitoring can provide an early warning system of something going wrong that triggers development teams to follow up and investigate.

This is particularly important to a microservices architecture because the microservice preference towards choreography and event collaboration leads to emergent behavior. While many pundits praise the value of serendipitous emergence, the truth is that emergent behavior can sometimes be a bad thing. Monitoring is vital to spot bad emergent behavior quickly so it can be fixed.

Microservice teams would expect to see sophisticated monitoring and logging setups for each individual service such as dashboards showing up/down status and a variety of operational and business relevant metrics. Details on circuit breaker status, current throughput and latency are other examples we often encounter in the wild.

9. **Evolutionary Design**

Whenever you try to break a software system into components, you're faced with the decision of how to divide up the pieces - what are the principles on which we decide to slice up our application? The key property of a component is the notion of independent replacement and upgradeability - which implies we look for points where we can imagine rewriting a component without affecting its collaborators. Indeed many microservice groups take this further by explicitly expecting many services to be scrapped rather than evolved in the longer term.

The Guardian website is a good example of an application that was designed and built as a monolith, but has been evolving in a microservice direction. The monolith still is the core of the website, but they prefer to add new features by building microservices that use the monolith's API. This approach is particularly handy for features that are inherently temporary, such as specialized pages to handle a sporting event. Such a part of the website can quickly be put together using rapid development languages, and removed once the event is over. We've seen similar approaches at a financial institution where new services are added for a market opportunity and discarded after a few months or even weeks.

Putting components into services adds an opportunity for more granular release planning. With a monolith any changes require a full build and deployment of the entire application. With microservices, however, you only need to redeploy the service(s) you modified. This can simplify and speed up the release process. The downside is that you have to worry about changes to one service breaking its consumers. The traditional integration approach is to try to deal with this problem using versioning, but the preference in the microservice world is to only use versioning as a last resort. We can avoid a lot of versioning by designing services to be as tolerant as possible to changes in their suppliers.

**2 Are Microservices the Future?**

Often the true consequences of your architectural decisions are only evident several years after you made them. We have seen projects where a good team, with a strong desire for modularity, has built a monolithic architecture that has decayed over the years. Many people believe that such decay is less likely with microservices, since the service boundaries are explicit and hard to patch around. Yet until we see enough systems with enough age, we can't truly assess how microservice architectures mature.

There are certainly reasons why one might expect microservices to mature poorly. In any effort at componentization, success depends on how well the software fits into components. It's hard to figure out exactly where the component boundaries should lie. Evolutionary design recognizes the difficulties of getting boundaries right and thus the importance of it being easy to refactor them. But when your components are services with remote communications, then refactoring is much harder than with in-process libraries. Moving code is difficult across service boundaries, any interface changes need to be coordinated between participants, layers of backwards compatibility need to be added, and testing is made more complicated.

Another issue is If the components do not compose cleanly, then all you are doing is shifting complexity from inside a component to the connections between components. Not just does this just move complexity around, it moves it to a place that's less explicit and harder to control. It's easy to think things are better when you are looking at the inside of a small, simple component, while missing messy connections between services.

Finally, there is the factor of team skill. New techniques tend to be adopted by more skillful teams. But a technique that is more effective for a more skillful team isn't necessarily going to work for less skillful teams. We've seen plenty of cases of less skillful teams building messy monolithic architectures, but it takes time to see what happens when this kind of mess occurs with microservices. A poor team will always create a poor system - it's very hard to tell if microservices reduce the mess in this case or make it worse.

One reasonable argument we've heard is that you shouldn't start with a microservices architecture. Instead begin with a monolith, keep it modular, and split it into microservices once the monolith becomes a problem.

**References**

1. The term monolith has been in use by the Unix community for some time. It appears in The Art of Unix Programming to describe systems that get too big.
2. Many object-oriented designers, including ourselves, use the term service object in the Domain-Driven Design sense for an object that carries out a significant process that isn't tied to an entity. This is a different concept to how we're using "service" in this article. Sadly the term service has both meanings and we have to live with the polyseme.
3. We consider an application to be a social construction that binds together a code base, group of functionality, and body of funding.
4. The original paper can be found on Melvyn Conway's website here
5. We can't resist mentioning Jim Webber's statement that ESB stands for "Egregious Spaghetti Box".
6. Netflix makes the link explicit - until recently referring to their architectural style as fine-grained SOA.
7. At extremes of scale, organisations often move to binary protocols - protobufs for example. Systems using these still exhibit the characteristic of smart endpoints, dumb pipes - and trade off *transparency* for scale. Most web properties and certainly the vast majority of enterprises don't need to make this tradeoff - transparency can be a big win.