Microservice Architectural Style

Introduction

Today, there are several trends that are forcing application architectures to evolve. Users expect a rich, interactive and dynamic user experience on a wide variety of clients including mobile devices. Applications must be highly scalable, highly available and run on cloud environments. Organizations often want to frequently roll out updates, even multiple times a day. Consequently, it's no longer adequate to develop simple, monolithic web applications that serve up HTML to desktop browsers.

It is important to realize that this architectural style will allow us to change our product strategy much faster, and create business strategy that will give us some money from it.

What is microservice architectural style?

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 (RESTful). 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.

Who is this document for?

For you, who want to use web as a platform for building distributed systems.
To understand what business strategy is, and how it is related to a product strategy.
To recognize different types of product strategy.
To make an opportunity for your partners and your company.

Characteristics of Microservice architecture

Services as components

One main reason for using services as components 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 understand.

Organized around Business Capabilities

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, persistent 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 teams.

Product vs Project

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.

The product mentality, ties in with the linkage to business capabilities. Rather than looking at the software as a set of functionality to be completed, there is an on-going relationship where the question is how can software assist its users to enhance the business capability.

There's no reason why this same approach can't be taken with monolithic applications, but the smaller granularity of services can make it easier to create the personal relationships between service developers and their users**.**

Communication mechanisms in a microservice architecture

In a microservice architecture, the patterns of communication between clients and the application, as well as between micro services, are different than in a monolithic application. Let’s have a look at the communication mechanisms within the microservices and how clients interact with the microservices.

Inter-service communication mechanisms

Another major difference with the microservice architecture is how the different components of the application interact. In a monolithic application, components call one another via regular method calls. But in a microservice architecture, different services run in different processes. Consequently, services must use an inter-process communication (IPC) to communicate.

Synchronous HTTP

There are two main approaches to inter-process communication in a microservice architecture. One option is to a synchronous HTTP-based mechanism such as REST or SOAP. This is a simple and familiar IPC mechanism. It’s firewall friendly so it works across the Internet and implementing the request/reply style of communication is easy. The downside of HTTP is that it doesn't support other patterns of communication such as publish-subscribe.

Another limitation is that both the client and the server must be simultaneously available, which is not always the case since distributed systems are prone to partial failures. Also, an HTTP client needs to know the host and the port of the server. While this sounds simple, it's not entirely straightforward, especially in a cloud deployment that uses auto-scaling where service instances are ephemeral. Applications need to use a service discovery mechanism. Some applications use a service registry such as Apache ZooKeeper or Netflix Eureka. In other applications, services must register with a load balancer, such as an internal ELB in an Amazon VPC.

Asynchronous messaging

An alternative to synchronous HTTP is an asynchronous message-based mechanism such as an AMQP-based message broker. This approach has a number of benefits. It decouples message producers from message consumers. The message broker will buffer messages until the consumer is able to process them. Producers are completely unaware of the consumers. The producer simply talks to the message broker and does not need to use a service discovery mechanism. Message-based communication also supports a variety of communication patterns including one-way requests and publish-subscribe. One downside of using messaging is needing a message broker, which is yet another moving part that adds to the complexity of the system. Another downside is that request/reply-style communication is not a natural fit.

Do not put significant smarts into the communication mechanism itself. A good example of this is the Enterprise Service Bus (ESB), where ESB products often include sophisticated facilities for message routing, choreography, transformation, and applying business rules. The microservice community favours an alternative approach: *smart endpoints and dumb pipes*. Implement lightweight message bus that acts as a message router only.

API gateway pattern

In a monolithic architecture, clients of the application, such as web browsers and native applications, make HTTP requests via a load balancer to one of N identical instances of the application. But in a microservice architecture, the monolith has been replaced by a collection of services. Consequently, a key question we need to answer is what do the clients interact with?

An application client, such as a native mobile application, could make RESTful HTTP requests to the individual services as shown in figure 1.

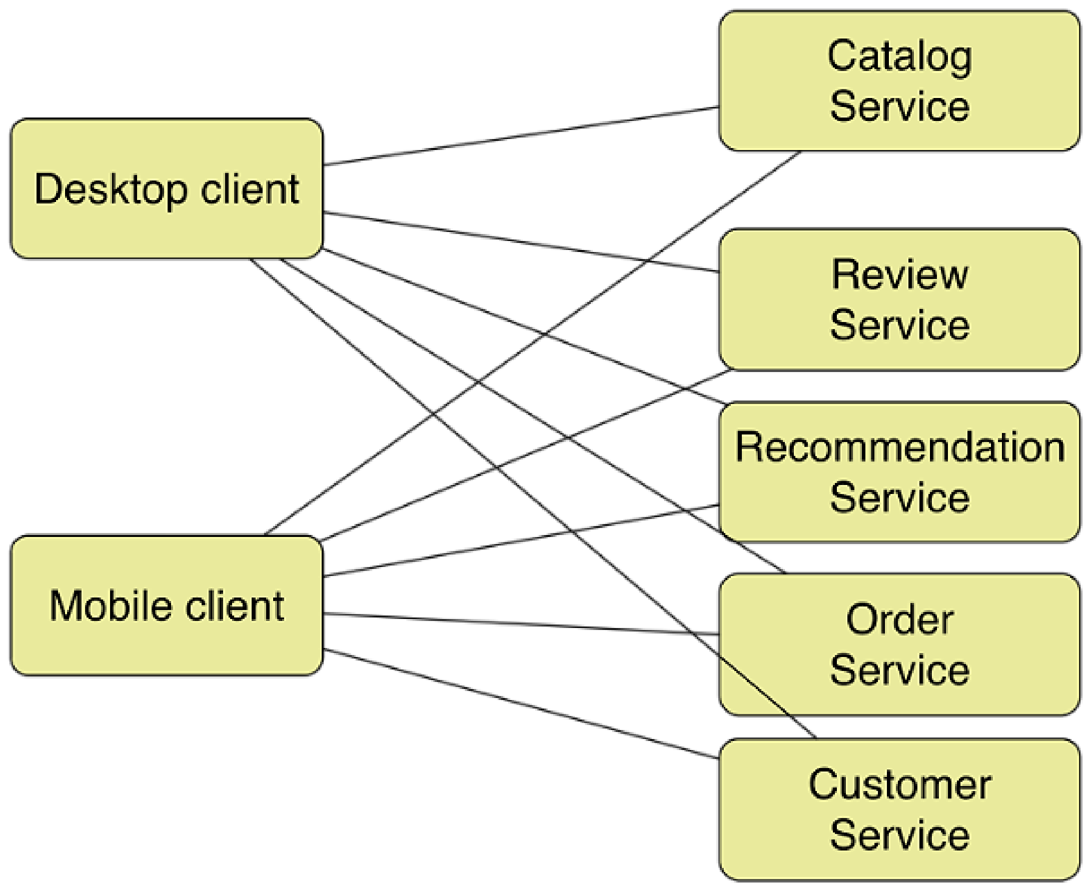


Figure 1 - Calling services directly

On the surface this might seem attractive. However, there is likely to be a significant mismatch in granularity between the APIs of the individual services and data required by the clients. For example, displaying one web page could potentially require calls to large numbers of services. Amazon.com, for example, describes how some pages require calls to 100+ services. Making that many requests, even over a high-speed internet connection, let alone a lower-bandwidth, higher-latency mobile network, would be very inefficient and result in a poor user experience.

A much better approach is for clients to make a small number of requests per-page, perhaps as few as one, over the Internet to a front-end server known as an API gateway, which is shown in Figure 2.

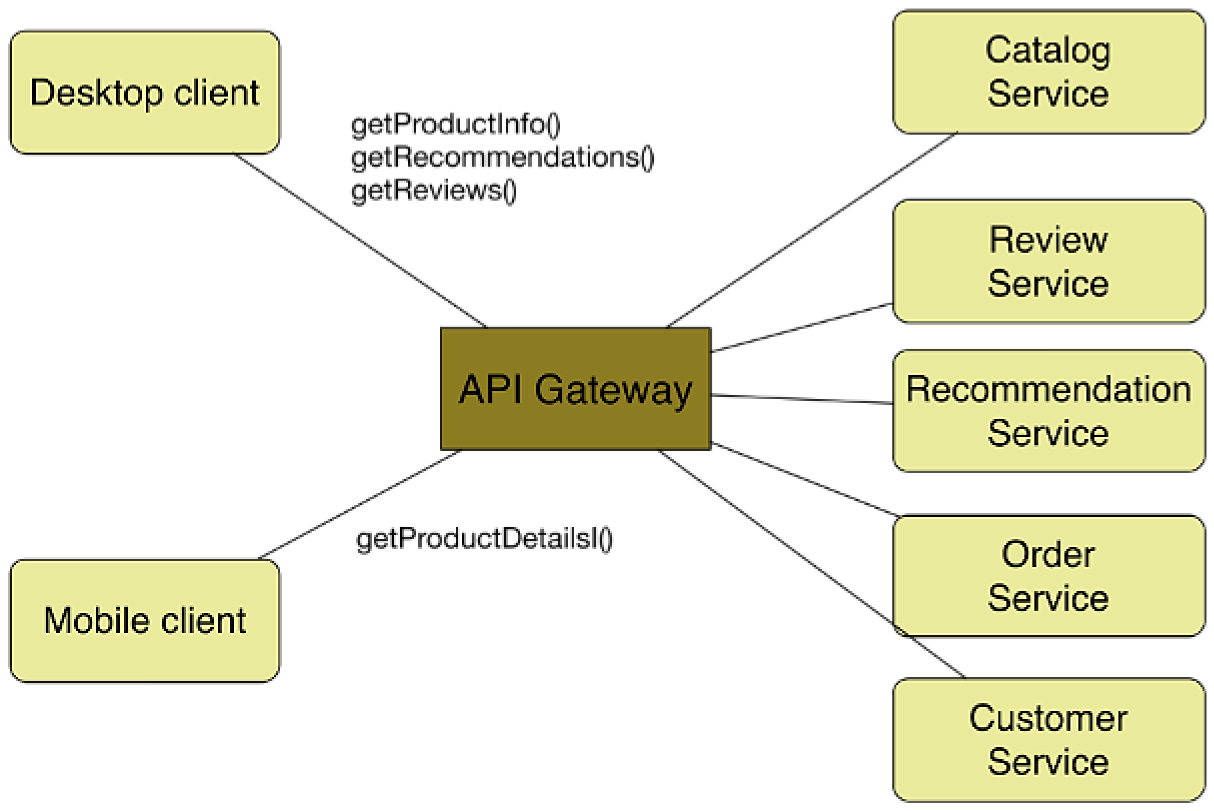


Figure 2 - API gateway

The API gateway sits between the application's clients and the microservices. It provides APIs that are tailored to the client. The API gateway provides a coarse-grained API to mobile clients and a finer-grained API to desktop clients that use a high-performance network. In this example, the desktop clients makes multiple requests to retrieve information about a product, where as a mobile client makes a single request.

The API gateway handles incoming requests by making requests to some number of microservices over the high-performance LAN. Netflix, for example, describes how each request fans out to on average six backend services. In this example, fine-grained requests from a desktop client are simply proxied to the corresponding service, whereas each coarse-grained request from a mobile client is handled by aggregating the results of calling multiple services. This aggregation can be performed in parallel (asynchronously) and increase performance more.

Not only does the API gateway optimize communication between clients and the application, but it also encapsulates the details of the microservices. This enables the microservices to evolve without impacting the clients. For examples, two microservices might be merged. Another microservice might be partitioned into two or more services. Only the API gateway needs to be updated to reflect these changes. The clients are unaffected.

You can consider this as separate component (service) in your archicteture that will be responsible for using 'policies' on the API gateway to add functionality to a service without having to make any changes to the backend service. For example, you can add policies to your gateway to perform data transformations and filtering, add security, analyze, monitor, monetize, execute conditional logic or custom code, and to perform many other actions. This is API management (apigee, 3scale,...).

API (Product) Strategy

The data and services made available via your APIs (gateway) are consumed by your employees, your customers, or other businesses (B2E, B2C, and B2B).

Client app developers face challenges when trying to consume services from different providers. There are many technologies available today for use by a service provider to expose its services. The same client app might have to use one mechanism to consume a service from one provider, and a different mechanism to consume a service from a different provider. App developers can even face the situation where they have to use different mechanisms to consume services from the same provider.

How does a business go about building an API strategy? Is your business seeking to connect with customers, or digitize and streamline internal business processes, or create new channels to work with partners? An understanding of the different kinds of API models, and which one is right for your business, is critical.

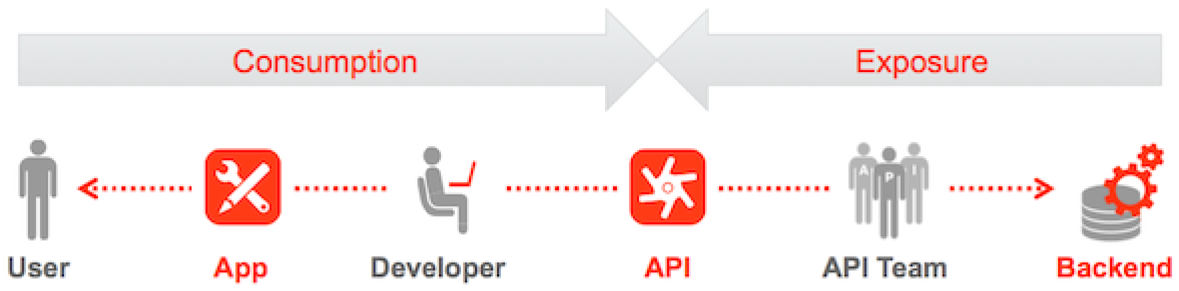
This understanding is aided by exploring the “digital value chain” that exists within a digital business and links enterprise data all the way through to the apps and consumers that benefit from it.

The digital value chain: from back-end to app-end

Much like a physical value chain, where a series of actions takes place to deliver a product to market, the digital value chain connects users to apps to developers to APIs (and API teams) to enterprise data and services in the backend.

The API team is like an internal “partner team,” working with an enterprise’s distributors and resellers, and managing which products are available. In the digital value chain, your distributors and resellers are developers, who build your digital presence (your “storefronts”) in the form of apps.

These developers might be within your company, work at a partner organization, or might operate independently in the outside world. Regardless of where they are innovating, developers represent a new channel, and the better their product (the app), the better the engagement with your end users.



http://apigee.com/docs/api-services/content/apigee-edge-organization-structure

The digital value chain

Traditionally, enterprise APIs have been associated with “exposure,” which describes the transformation of existing backend capabilities, resources, and data into APIs. But the focus of APIs in the modern enterprise has been expanded to include “consumption,” which grants developers access to those same resources so they can build and deploy apps. Imagine developers being able to explore API, to subscribe, to read statistics data.

Data connects the value chain end to end—in both directions. APIs don’t just enable an enterprise to expose or project data. They are no longer just the sockets through which transactions pass; they create a conduit for data to flow back to the enterprise and today are the primary tools for data collection and analysis.

Internal, partner, customer, and open strategies

Several flavors of API initiative support the digital economy. Many successful API initiatives are done in stages. With each stage, businesses can build on previous projects, assume more risk, and invest in larger projects more easily. The type of initiative you launch—internal, partner, or open—depends upon whether the app developer resides within your business, within a partner’s company, or works independently (“in the wild”).

Companies might be inclined to start with an open API, using Twitter, Foursquare, or Facebook as an archetype. This approach isn’t necessarily the right way to get started. Rather, an open strategy should generally be undertaken after a business has learned lessons and mitigated risks by executing an internal or partner model first.

https://blog.apigee.com/sites/blog/files/Agility-Collaboration_Innovation_Final_0.jpg

Internal

A partner API initiative is one that focuses on building an app-enabled business: As companies become larger and more complex, many are deferring to API ecosystems to minimize the amount of development effort to support multi-channel enablement. Developers are within the company, building the apps by consuming the APIs, and giving the feedback to

API teams. For example developers could be asked to build android app for HR department by consuming corporate API.

https://blog.apigee.com/detail/why_apis_anatomy_of_an_internal_api_initiative

Partner

A partner API initiative is one that focuses first on collaboration with strategic partners. Those partners create applications, add-ons, or integrations with the API. At this stage the API gets hardened and because the API is used across organizational boundaries, the API team will learn a new set of lessons including support, documentation, authentication schemes and so on. How many times we have lost potential partners because of this missing chanel?

https://blog.apigee.com/detail/why_apis_anatomy_of_a_partner_api_initiative/

Customer

The Customer API initiative is mostly used in one of two scenarios. The first is when offering software as a service (SaaS). (Look at Salesforce as an example where customers demand an API.) The second is when the customer of your business is another business (a B2B scenario).

https://blog.apigee.com/detail/why_apis_anatomy_of_a_customer_api_initiative/

Open

This is the case for innovation by leveraging the creativity and know-how of 100s of 1000s of developers around the world using your API to create cool apps and make big breakthroughs.

If you target an Open API initiative, chances are that your internal or partner strategies will go well too. It is important to be careful about setting expectations. A lot of things need to go right to get huge numbers of developers successfully creating apps on your APIs.

Business Strategy

Business strategy is about identifying your business objectives and deciding where to invest to best achieve those objectives. For example, moving from a direct sales model (your own sales force selling directly to customers) to an online sales model (your customers buy from your site) is a business strategy. Deciding whether to charge for your services (api) with subscriptions or transactions fees or whether you have an advertising-based revenue model is a business strategy. Deciding to move into an adjacent market is a business strategy. Deciding to charge your services (API), with free option of 2500 API calls per month.

Moreover, while the business may believe something is a great business opportunity, you don't yet know if your company can successfully deliver on this opportunity. Maybe it will cost too much to build. Maybe customers won't value it enough to pay for it. Maybe it'll be too complicated for users to deal with. This is where api(product) strategy and especially product discovery come into play to find out answers to this questions.

The product strategy speaks to how you hope to deliver on the business strategy.

Not A Free Lunch

Microservices one of these ideas that are nice in practice, but all manner of complexity comes out when it meets reality.

Significant Operational Overhead

A Microservices architecture brings a lot of operations overhead.

Where a monolithic application might have been deployed to a small application server cluster, you now have tens of separate services to build, test, deploy and run, potentially in polyglot languages and environments.

All of these services potentially need clustering for failover and resilience, turning your single monolithic system into, say, 20 services consisting of 40-60 processes after we've added resilience.

Throw in load balancers and messaging layers for plumbing between the services and the estate starts to become pretty large when compared to that single monolithic application that delivered the equivalent business functionality!

Productionising all of this needs high quality monitoring and operations infrastructure. Keeping an application server running can be a full time job, but we now have to ensure that tens or even hundreds of processes stay up, don't run out of disk space, don't deadlock, stay performant. It's a daunting task.

Physically shipping this plethora of Microservices through your pipeline and into production also needs a very high degree of robust and release and deployment automation.

Currently, there is not much in terms of frameworks and open source tooling to support this from an operational perspective. It's likely therefore that a team rolling out Microservices will need to make significant investment in custom scripting or development to manage these processes before they write a line of code that delivers business value.

Operations is the most obvious and commonly held objection towards the model, though it is too easily brushed aside by proponents of this architecture.

Substantial DevOps Skills Required

Where a development team might have been able to bring up, say, a Tomcat cluster and keep it available, the operations challenges of keeping Microservices up and available mean you definitely need high quality DevOps and release automation skills embedded within your *development* team.

You simply can't throw applications built in this style over the wall to an operations team. The development team need to be very operationally focussed and production aware, as a Microservices based application is very tightly integrated into it's environmental context.

Idiomatic use of this architecture also means that many of the services will also need their own data stores. Of course, these could also be polyglot (the right tool for the job!), which means that the venerable DBA now needs to be replaced with developers who have a good understanding of how to deploy, run, optimise and support a handful of NoSQL products.

Developers with a strong DevOps profile like this are hard to find, so your hiring challenge just became an order of magnitude more difficult if you go down this path.

Implicit Interfaces

As soon as you break a system into collaborating components, you are introducing interfaces between them. Interfaces act as contracts, with both sides needing to exchange the same message formats and having the same semantic understand of those messages.

Change syntax or semantics on one side of the contract and all other services need to understand that change. In a Microservices environment, this might mean that simple cross cutting changes end up requiring changes to many different components, all needing to be released in co-ordinated ways.

Sure, we can avoid some of these changes with backwards compatibility approaches, but you often find that a business driven requirements prohibit staged releases anyway. Releasing a new product line or an externally mandated regulatory change for instance can force our hand to release lots of services together. This represents additional release risk over the alternative monolithic application due to the integration points.

If we let collaborating services move ahead and become out of sync, perhaps in a canary releasing style, the effects of changing message formats can become very hard to visualise.

Again, backwards compatibility is not a panacea here to the degree that Microservices evangelists claim.

Distributed System Complexity

Microservices imply a distributed system. Where before we might have had a method call acting as a subsystem boundary, we now introduce lots of remote procedure calls, REST APIs or messaging to glue components together across different processes and servers.

Once we have distributed a system, we have to consider a whole host of concerns that we didn't before. Network latency, fault tolerance, message serialisation, unreliable networks, asynchronicity, versioning, varying loads within our application tiers etc.

Coding for some of these is a good thing. Backwards compatibility and graceful degradation are nice properties to have that we might not have implemented within the monolithic alternative, helping keep the system up and more highly available than the monolithic application would be.

The cost of this however is that the application developer has to think about all of these things that they didn't have to before. Distributed systems are an order of magnitude more difficult to develop and test against, so again the bar is raised vs building that unsexy monolithic application.

Asynchronicity Is Difficult!

Related to the above point, systems built in the Microservices style are likely to be much more asynchronous than monolithic applications, leaning on messaging and parallelism to deliver their functionality.

Asynchronous systems are great when we can decompose work into genuinely separate independent tasks which can happen out of order at different times.

However, when things have to happen synchronously or transactionally in an inherently Asynchronous architecture, things get complex with us needing to manage correlation IDs and distributed transactions to tie various actions together.

Testability Challenges

With so many services all evolving at different paces and different services rolling out canary releases internally, it can be difficult to recreate environments in a consistent way for either manual or automated testing. When we add in asynchronicity and dynamic message loads, it becomes much harder to test systems built in this style and gain confidence in the set of services that we are about to release into production.

We can test the individual service, but in this dynamic environment, very subtle behaviours can emerge from the interactions of the services which are hard to visualise and speculate on, let alone comprehensively test for.

Idiomatic Microservices involves placing less emphasis on testing and more on monitoring so we can spot anomalies in production and quickly roll back or take appropriate action. I am a big believer in this approach - lowing the barriers to release and leaning continuous delivery in order to speed up lean delivery. However, as someone who has also spent years applying test automation to gain confidence prior to release, anything that reduces this capability feels like a high price to pay, especially in risk averse regulated environments where bugs can have significant repercussions.

The Hype

However, when considering Microservice like architectures, it's really important to not be attracted to the hype on this one as the challenges and costs are as real as the benefits.

I am still a fan of the approach and believe that on the right project with the right team it is a wonderful architecture to adopt where the benefits outweigh the costs above.

Mutualistic Symbiotic Relationship (Microservices and PAAS)

If you look at the concerns typically expressed about microservices, you'll find that they are exactly the challenges that a PaaS is intended to address. So while microservices do not necessarily imply cloud (and vice versa), there is in fact a symbiotic relationship between the two, with each approach somehow compensating for the limitations of the other, much like the practices of eXtreme Programming do the same.

Platform as a Service (PaaS) offerings like Cloud Foundry have raised the level of abstraction to a focus on an ecosystem of applications and services.

It does not have to be a public cloud. Cloud Foundry is open source and it can be deployed on private or public (IaaS) infrastructure.

Technology

You can choose many different frameworks, platforms and tools to help you in achieving this with lower costs. I am choosing some of them that I believe you should consider at least. As you already know, every service can be constructed with different set of tools or frameworks (whatever you find fit), and this is great advantage. If you are fan of Java, it is hard to avoid Spring platform. It is always a good idea to look how others have done it, Netflix for example.

Spring

The Spring Framework is looking ahead into this micro-service future, with both direct and foundational support. REST is a 1st class citizen across Spring, reflected in the core 4.0 framework with Spring MVC as well as Spring IO platform components like *Spring Boot*, *Spring HATEOAS*, *Spring Integration*, Spring Reactor, Spring Security, Spring Social, *Spring Cloud* and Spring Data Rest. With Spring Framework 4.0, developers can create more reactive event-driven REST services, using the new non-blocking AsyncRestTemplate together with Java language features like Futures.

Spring Boot, which builds on Spring Framework 4.0’s @Conditional bean definition infrastructure, offers a “containerless” (embedded) runtime for REST based micro services, the ability to package as a single executable JAR, in addition to other lightweight capabilities - around exposing metrics for itself, for example.

Spring HATEOAS provides some APIs to ease creating REST representations that follow the HATEOAS principle when working with Spring and especially Spring MVC. The core problem it tries to address is link creation and representation assembly.
Spring Cloud provides tools for developers to quickly build some of the common patterns in distributed systems (e.g. configuration management, service discovery, circuit breakers, intelligent routing, micro-proxy, control bus, one-time tokens, global locks, leadership election, distributed sessions, cluster state). Coordination of distributed systems leads to boiler plate patterns, and using Spring Cloud developers can quickly stand up services and applications that implement those patterns. Some of the nice features are:

- config-server for centralized configuration
- refresh-able configuration (you can update configuration in runtime)
- service registration and discovery with Netflix Eureka
- managing failures with Netflix Histrix
- dynamic routing, monitoring, resiliency and security with Netflix Zool

Spring Integration extends the Spring programming model to support the well-known Enterprise Integration Patterns. Spring Integration enables lightweight messaging within Spring-based applications and supports integration with external systems via declarative adapters. Those adapters provide a higher-level of abstraction over Spring's support for remoting, messaging, and scheduling. Spring Integration's primary goal is to provide a simple model for building enterprise integration solutions while maintaining the separation of concerns that is essential for producing maintainable, testable code.

Decomposing applications, using RESTful principles is much faster with Spring this days.

NoSQL
ToDo

Netflix products
ToDo

References and further reading

Articles/Blogs/Presentations
http://martinfowler.com/articles/microservices.html
http://microservices.io/patterns/microservices.html
http://klangism.tumblr.com/post/80087171446/microservices
http://www.infoq.com/articles/microservices-intro
http://plainoldobjects.com/2014/04/01/building-microservices-with-spring-boot-part1/
http://apigee.com/docs/api-services/content/what-apigee-edge
https://spring.io/blog/2014/04/29/springone2gx-2013-replay-futures-and-rx-observables-powerful-abstractions-for-consuming-web-services-asynchronously

http://www.slideshare.net/mobile/SpringCentral/futures-andrxobservables-springone2013
http://continuousdeliveryexperience.com/conference/fort_lauderdale/2014/12/session?id=31942
http://blog.pivotal.io/cloud-foundry-pivotal/p-o-v/field-report-the-3-reasons-why-the-cfsummit-rocked
http://highscalability.com/blog/2014/4/8/microservices-not-a-free-lunch.html?printerFriendly=true
http://nirmata.com/2014/08/getting-started-microservices-using-netflix-oss-docker/
http://martinfowler.com/bliki/MicroservicePrerequisites.html
https://speakerdeck.com/joshlong/building-bootiful-microservices-with-spring-boot

Books

http://www.amazon.com/REST-Practice-Hypermedia-Systems-Architecture/dp/0596805829/ref=sr_1_1?s=books&ie=UTF8&qid=1407097230&sr=1-1&keywords=rest+in+practice

http://www.amazon.com/Release-Production-Ready-Software-Pragmatic-Programmers/dp/0978739213/ref=sr_1_1?s=books&ie=UTF8&qid=1407097123&sr=1-1&keywords=release+it

http://www.amazon.com/Enterprise-Integration-Patterns-Designing-Deploying/dp/0321200683/ref=la_B001KDEH2S_1_1?s=books&ie=UTF8&qid=1407097536&sr=1-1

https://pages.apigee.com/web-api-design-ebook.html

http://12factor.net/

Spring Platform

http://projects.spring.io/spring-boot/
http://projects.spring.io/spring-integration
https://github.com/reactor
http://projects.spring.io/spring-data-rest/
http://projects.spring.io/spring-hateoas/

Source code

https://github.com/olivergierke/spring-restbucks
https://github.com/joshlong/spring-doge-microservice
https://github.com/spring-cloud-samples/customers-stores
https://github.com/spring-projects/spring-integration-samples/tree/master/applications/cafe
https://github.com/cloudfoundry/uaa