A Concise Guide Microservices

For Executives

RG Consulting

Executive Microservices Overview

Executive Summary

Microservices and microservice architecture are the latest hot enterprise level methods for producing high performance, scalable and flexible applications. Microservice has proven itself in the largest and most testing of environments and its success with Netflix and Amazon has made others sit up and take note.

However, what is microservice, and how can it be of utility in the enterprise data center? In this article, we will discuss and address the scalability problems that microservice addresses and how systems can be restructured and services distributed to enhance availability and performance. We will also describe the plethora of other features and advantages both great and small that microservices bring to the enterprise. Additionally we will look at its strengths and weaknesses, and where it should and should not be considered suitable for deployment.

What are Microservices?

Microservice is the term given to a style of software architecture that involves building and delivering systems as a set of small, granular, independent yet collaborating services. However, microservices are not necessarily a new concept. Instead, microservices have come to the fore recently perhaps due to their successful use at Netflix who have applied the software pattern with great effect.

The methodology behind microservices is to breakdown the entire application into individual modular packages of related functionality. By doing so, it enables the development of the application in a concurrent distributed fashion; however, it also loosely couples all the business functionality services. By doing so, it removes many of the service dependencies making it easier to work collaboratively and/or to rework or rewrite code independently.

By taking a microservices approach to application design the developer can ensure that:

- Services are focused on doing one thing but doing it well
- That services can be built using the best available technology
- That developer's can work in collaboration on different services in relative independence which means running concurrent development to shorten the development timeline.
- That applications designed and built in this manner are always loosely coupled which means a developer can rework, change and deploy individual components without affecting the entire system.
- Microservices are great enabler for frequent upgrade and new features as deployment and delivery does not affect the entire system only the specific components being upgraded.

Microservices concept is similar to SOA (Software Orientated Architecture) and the principles of OOP (Object Orientated Programming) where the focus is on loose coupling and the removal of dependencies. The concepts are similar but the scale is different. SOA provides separation through infrastructure design and hardware deployment of individual modular components. OOP is a class/methods based object orientated philosophy, which has a goal of providing granularity and focus by encouraging methods and functions to address only a single action. Microservices from a conceptual perspective also encourages the de-coupling of services and dependencies as well as the focus of services on individual pieces of business logic.

Decomposing the applications business logic into independent functional components has many advantages both from developmental and maintenance viewpoints. From a deployment perspective, it allows a development team to be small and highly focused. Additionally it gives them the freedom to choose whichever technology or language that best suits their specific use. Similarly, by having the application compartmentalized by function and deployed independently it means troubleshooting is easier and less disruptive to the applications availability.

A microservice approach to application development is also advantageous when it comes to deployment of a new or reworked application within a microservice architecture. It is within the architecture where we see the similarities with SOA.

What is Microservice Architecture?

The style of the microservice architecture, delivers a single application comprising a suite of small individual services modules. Each of these services runs its own business logic and communication processes through HTTP resource API or through messaging services. These microservices were built around individual business functionality and are designed to be independent of other functional services that make up the system. Being modular in design the application could then be deployable on different servers and infrastructure. This addresses many problems that IT has with scaling enterprise applications in the data center.

Monolithic Application strengths and flaws

Microservices contrasts with the traditional monolithic approach whereby a single enterprise application would be built as one single unit, comprising a user interface, a server-side application and a database. Monolithic application architecture is successful but has several shortcomings especially now that many applications are being deployed in the cloud. One such shortcoming is that the application developers build it as a single unit with all the logic for handling requests running in a single process. This is then delivered as a single software package and it therefore installed on a single server.

Scalability is achieved through replicating the application on other servers and load balancing requests. This monolithic approach is beneficial and is used very successfully in data centers as it divides the application into classes and functions under one deployable application. Typically, the application under development can be run on a consultant's laptop and a development timeline and pipeline can be used to monitor and ensure changes and version control. The problem though is that it is one single unit and therefore any change requires a rebuild of the entire application. The interdependencies of every service and process mean it can be difficult to contain changes meant for only one module from leaking and affecting other modules. Also, as we have seen scaling requires replicating the whole application rather than by releasing resources for the parts of the application that needs them.

Microservice architecture removes the constraints of having the entire application hosted on one server so that resources can be more effectively attributes to those parts of the application that needs them. However, the obvious penalty is that by separating the business functions and services there will need to be a communication system between the services and this requires additional integration overhead.

Integration is always a tricky business so having more of it to do is not particularly advantageous. However, if a communicating between services then either HTTP web services or subscribe/publish messaging protocols can be used to simplify the requirements of security, performance and reliability. On deciding which to use it is good advice to deploy HTTP web service when asking a question that is requiring a direct answer, and subscribe/publish when making a statement that requires no immediate feedback.

The microservice architecture provides other advantageous elements such as:

- Each service is typically small which makes it easier for a developer to understand and this will quicken deployment. Being small the service will start quicker within its web-container which again hastens development and deployment.
- Each service can be deployed separately from each other this facilitates the ability to make specific changes and feature updates on a continuous change basis without disrupting operation or availability of the entire application.
- Using microservice architecture also makes development and deployment more efficient as small teams are required to work on individual service's components and they work independently of the other teams using their own technologies and

tools to get the job done without external influence or constraints.

- Improved maintenance and troubleshooting as small services make fault-finding easier as a problem within one service will be isolated from the others.
- Microservice architecture provides the means to use whatever software or language that suits the task in hand which breaks away from the hard-set technology stack of the monolithic application. Many large modern web-sites such as Amazon and eBay started off as monolithic applications that evolved to using SOA and microservices.
- A microservice approach is also a favored way in which to introduce new and emerging technologies even in a monolithic environment as microservices can be built to proxy functionality and work alongside the monolithic application. In this way, it is easy to build new or updated services using emerging technologies without having to rewrite the entire application stack. This is particularly useful for websites that wish to introduce hot emerging features of a particular technology as they can adapt their monolithic website application to work alongside the microservice that will provide the new service.
- Microservices also provides an upgrade path by using the dissemination of the monolithic application into functional areas that can then be replicated and proxy by the microservice architecture. By taking one functional area at a time, to proxy, deploy, and switch off, the development team can safely convert the monolith into a microservice architecture one-step at a time. However, the problem there is being able to identify what is a suitable function to use as a microservice, and how do we identify one?

How to decompose a monolithic application into services?

One of the problems with trying to determine what pieces of code can be decoupled from another and be a suitable subject for microservice is that there is no real clear definition as to what microservice actually is. Not all the currently accepted consensus appears to agree though some suggest it should be less than 100 LOC (lines of code). Of course this is no sort of definition at all so we have to go on a more practical criteria, that of functionality. Therefore when breaking apart a large monolithic application or specification we have to identify areas that can be readily replaced or upgraded with no impact on other services. These techniques harp back to one of the principles of modular design, which is, do drive modularity through the requirement of change. The idea being that you want to keep things that will change at the same time to be in the same module. Secondly, services that rarely change should be kept away from services that are frequently updated they should be in separate modules. Similarly, modules whose services change often and as a result of one another's actions should be in the same module.

Another strategy is to target the functionality of the system using the verb or use case technique. These can be found and should be well documented in the latest software requirements and specifications (SRS). By comparing the use-case examples for functionality, the developer can divide the system using inbuilt functionality, which is beneficial, as the original developers designed them to be modular and independent. The functional modules will also have a known input and known outputs. Examples of these functional areas, which are encompassed into microservices, are commonly used website functions such as login/registration services and shipping orders.

These are some obvious functional operations with use-case examples however; some areas may not have verb or a detailed use-case. In that case, a noun or resources approach will be required. Noun/resource partitioning is done by identifying those services, which operate on resources of a given type. These are such entities as inventories and price lists, which will track the stock levels of products and hold their current and historical prices.

However, whichever method is used to segregate the code and provide the basis for the microservice a few key pointers can be helpful and these come from the world of OOP. In OOP, the Single Responsibility Principle defines a class as having only one reason to change. This defines its functionality, in so much as the class can support one and only one function. This is a good basis to work from when dividing operational functions in an enterprise application. By ensuring that microservices cover only one well-defined business logic function, which has well defined inputs and outputs, the designer will invariably find the process of decomposing a huge enterprise application a much less onerous task.

Once the developer has decided upon the method and the process to decompose an existing monolithic application or to design an application in a Greenfield project, there is the problem of integration. Integration is always a tricky and sometimes time-consuming business especially if third parties are involved. Microservice does nothing to ease those woes in fact it makes integration tasks far more numerous. Therefore, you need to have a well-defined integration policy for protocols and technologies.

What are Microservice integration protocols?

As we touched on earlier, HTTP and messaging services are typically the protocols used within microservice architectures to communicate between microservices. HTTP is a synchronous protocol and is suitable for communications that require an instant action. Messaging services are asynchronous and are suitable for subscribe/publish style command or information that can be processed later when perhaps a receiver (service/subscriber) comes online. A general rule of thumb can be ‘when asking a question’ that expects an immediate reply use an HTTP protocol service. Conversely ‘when publishing information’ that does not require an immediate response, if any, then use a subscribe/publish messaging queue. The reason for this is simply to try to minimize the errors that may arise during service interaction. Message queuing alleviates many of the timing issues that may arise when a service finds another unresponsive due to being busy or temporarily offline. By placing the message in a queue, the recipient service can pick up and process the message when it becomes available.

Client Interfacing

In a monolithic environment clients of the application communicate through many instances of the application via a load balancer. The load balancer in this situation receives and manages the client’s connection to a particular instance of the application. However, in a microservice architecture the monolith application has been replaced by a set of loosely coupled services distributed on an array of hardware. Calling these services directly is not really an effective method especially in today’s mobile device centric working environments where internet connections may have lower bandwidth and have higher latency.

A high speed LAN connected desktop may well be able to call hundreds of services directly when opening a website. An example being how Amazon describes some of their product sales pages require 100+ service calls. The direct call method may work in the case of a directly connected LAN desktop but a smartphone or tablet attempting that amount of service calls over a slow mobile or wireless connection is certainly going to result in long latency on page opening and refresh. The answer is to install an API gateway.

API Gateway

The purpose of the API gateway is to act as a mediator between the client and the services so that instead of the mobile client requesting 100+ services it will ask for only one or two. The API gateway will handle these service calls as being aggregates of the 100+ calls and handle them as a proxy for the client. The result is that far fewer actual service calls are traversing the connection between the client and the API gateway and consequently the applications services. This greatly enhances the websites performance.

A further advantage of using an API gateway is that it provides an additional layer of abstraction between the client and the services. This means services names can change; some may merge and only the API gateway will need to be informed and updated, the client remains oblivious of any backend changes.

However, there is still the inter-service communication to handle, as that too is different from the monolithic model. In the traditional monolithic model, services interact via internal method calls. This is a quick and easy method under a single process. However, that is not true with a microservice architecture whereby services will run in different processes on perhaps distributed systems. Consequently microservices needs to use a method of inter process communication between disparate services in order to effectively communicate.

Synchronous HTTP

Synchronous HTTP is a well-known method for inter-process communication and is commonly used in microservice architectures. Typically synchronous HTTP will use REST or SOAP service call mechanisms. The advantage of using these is that they are well known and firewall friendly so it will work across the internet. A further advantage is that both REST and SOAP are easy to understand and maintain as they use a standard request/reply style of communication. This as we saw earlier is the favored method for dealing with situations that require an immediate response.

The disadvantages to using synchronous HTTP services is that both parties must be simultaneously available and this is not always possible in a busy application as they could well reside on different hardware and communicate across the data centre network. The facts are that distributed systems are inherently prone to application and network induced latency, jitter, and packet loss which can result in partial failures. HTTP services are vulnerable to these temporary glitches and they will see the service as being down resulting in a failed service call.

Another problem is that clients require to address their calls to specific servers using an address or host name and a port. That maybe seem trivial in a small infrastructure, however when considering scaled or cloud deployments where auto-scaling will result in services being ephemeral. In such cases, different methods such as service discovery or service registrars are deployed to track the services, in others, a load balancer is used to register services and manage service calls.

Asynchronous messaging

An alternative to synchronous HTTP is too use asynchronous messaging which mitigates some of the problems by not requiring simultaneous availability. AMQP (Advanced Message Queuing Protocol) is a typical example. It is an open standard application level protocol for handling middleware intermediately message orientated communications. The advantages of AMQP or any other messaging queuing system is that they decouple the sender from the receiver or in messaging parlance the publisher from the subscriber by using a message broker. Messages are stored, buffered, by the message broker until the subscriber is able to consume the message. Publishers only know of the message broker, they have no awareness of subscribers or consumers of their service. As a result there is no need for service discovery as the publisher always talks directly with the message broker.

A particularly useful feature of message brokers is that by buffering messages for subscribers they can be effectively used to update data in distributed systems. Rather than use distributed transactions which again requires all participants to be available simultaneously a message broker based system of message publishing allows the subscribers to pick up the message when they become available. These event based asynchronous replications allows a service to update other services that some data has been changed. It doesn’t need to know who is interested the message broker will handle that as other services subscribe to those events.

The downside to asynchronous messaging is the obvious requirement of a message broker which is another component in the system which adds complexity and a possible point of failure. However in real world microservice architectures there is a tendency to deploy a mixture of synchronous and asynchronous communication techniques.

When to Use Microservices

One of the primary uses for utilizing a microservice style architecture is for scalability. Systems are typically scaled in monolithic enterprise data centers by taking one instance of the application and replicating it how many times as is necessary to provide redundancy, performance and capacity. Each instance of the application requires to be hosted on a separate server and they are connected via load balancer devices, which will proxy the incoming client connections and forward them to a particular instance of the application. Load balancers maintain a record of the sessions and make ensure a client session is always maintained and connected to the correct instance of the application. This form of scaling is termed x-axis scaling and is part of the 3-dimensional scale cube.

The 3-Dimennsional Scale Cube

As we have seen, the X-axis represents the horizontal length and is related to duplication or cloning applications or services. It is the traditional method for providing scaling in data centers. With cloning, each instance holds an identical copy of the application and handles the entire session with a client. X-axis provides redundancy and performance but does not address the problems with development and application complexity. A typical application for this model would be web server farms.

Another axis in the 3-dimensional scaling model is called the z-axis, and this represents the depth axis and this is also deployed in monolithic data centers. The Z-axis represents data partitioning and the splitting of similar things. Each instance will be an exact copy of the application; just like the X-axis, the difference being that in the Z-axis model each server is responsible for only a subset of the functionality. A typical application for this model is database scaling whereby each server is responsible for a subset of the data. An external device will redirect clients to the correct server holding the data they want perhaps determined by some criteria such as the primary key. Z-axis is also often used in applications especially search services which will send a query to each partition and aggregate the results for presentation to the client. Z-axis improves transaction efficiency as requests/queries are distributed across multiple servers.

The Y-axis is represented by the vertical axis and is related to functional decomposition and the splitting of different things. Unlike the X-axis and the Z-axis, which consist of complete copies of the applications, in the Y-axis model functionality is distributes across multiple different services. This requires that the application is decomposed into individual functional services and each server contains a copy and is responsible for a particular service. The advantage of Y-axis modeling is that server sizing and resources can be better managed, and applied to the unique requirements of each service. Y-axis modeling also provides a method for managing the scaling of application development and complexity.

To summarize – The 3 Dimensional Scale Model

- X-axis – horizontal duplication, scale by cloning systems
- Z-axis – Data partitioning defined by the splitting of similar things
- Y-axis – is defined by Functional decomposition and is the splitting of different

things

Microservices lends itself in particular to Y-axis scaling as it splits different services as part of its decomposition of the application into separate functional services. However, micro-services are not just about scaling it is also about technology versatility and using the best of breed techniques to do the task.

Database Scaling

The concept of CQRS is an example of database scaling, where reading and writing are designated as being separate objects. With CQRS, commands are differentiated from queries. A command results in a change (write), whereas a query returns a result (read). Systems can have many different read/write database requirements and ratios dependent of the service. Microservice allows the developer to design the optimal data access methods for that specific function.

Best of Breed Solutions

One of the biggest problems in building enterprise applications is deciding on the technology stack and what tools to use that will be best suited to perform all jobs. In a monolithic application architecture, this has to be determined up front and be compatible across the entire application development architecture. Microservices however is not dependent on other services technologies they are independent modules and such only require to interface with compatible input and outputs. This means they can be build on completely different languages and technologies.

Having the ability to use modern specialist tools to perform specific tasks within a service is a very compelling reason to use microservice architecture. Additionally due to a service, being a complete slice of an application covering each functional layer the best of breed choice can also be applied to the database layer. Therefore, a microservice is not constrained to using a relational SQL database it can use any type it wishes, one that is best suited to the data types the service works with. Similarly, a service can be designed to use Sinatra, when rendering HTML, or use Java dealing with complex domain rules. Event processing is best suited to functional programming so F# or NodeJS might be a better tool for high performance requirements. By using microservices, it provides a separation between functional services, which allows the developer the freedom to select the correct tool to solve a given problem rather than have to use a compromise.

Rates of Change

The separation of functionality also provides a method for continuous development. In most modern application new features and upgrades are necessary, to remain up-to-date and to support the latest technologies and content. Microservices provides a method for separation of key functional services that are likely to change often and those core components that will remain untouched. An example is the look and feel of websites, where technologies and techniques are rapidly changing. In order to facilitate the use of emerging specialist technologies and tools microservices can be designed to have the user interface logic and presentation decoupled from integration or business logic.

When introducing new technology

As we saw earlier, new technologies can be implemented and operated in parallel with a monolithic application. Some major websites use this technique to introduce innovative temporary features whilst still running the core code on the monolithic application.

Over-lapping Functionality

In many environments there are applications performing over-lapping or identical functions many times over. One example is authentication; every monolithic application uses its own form of user management and authentication. Microservices provides the opportunity for a service to support other applications and become a single-point of authentication.

These are some of the uses for a microservice architecture, which provide utility for the developer however, like everything it is not always the best solution in every situation.

When not to use Microservices

The problem of additional cost and complexity often rises when discussing Greenfield startup developments. It is true that creating an application using microservice architecture could prove to be both more expensive and take considerably longer; - both of these are major project performance indicators. Indeed a common philosophy is that microservice architectures only really come into their own and deliver benefit when an application reaches the stage where scalability is becoming a problem as only then are all the benefits realized. Instead, the consensus appears to be that startups use a monolithic architecture for fast and cheaper development to build production ready applications and address the problems of scalability at a later date – when funding is more accessible. While this may be wise advice it does leave the unsatisfactory situation whereby a startup is building an application on a limited budget and time constraint to an architecture model (monolithic) that is not the intended final deliverable architecture.

Unfortunately finance and time constraints will always dictate the viability of the technology alternatives. Therefore, if working to tight budgets, which most application development teams are not, then microservice architecture may not suit.

Costs – Significantly Higher Overheads

Whereas a monolithic application could be, build in a small server cluster a micro server approach is going to require more hardware and more load balancing and interfacing equipment such as message brokers. When that is budgeted for development, pre-production and production environments the costs can rise dramatically.

DevOps Skills

Application development teams build and deliver as a single application monolithic enterprise software that are presented or thrown over the wall to the operations team for deployment. This approach leads to troublesome deployments and releases due to the knowledge gap between the two teams. There is rarely a lot of operational knowledge resident within the development team and vice versa. With microservices, the problem is compounded to the point that it is a requirement that there is operation experience and knowledge of operational and networks within the development team. There is also in polyglot (the best technology for the job) environments specific database administration knowledge and proficiency in both teams if a new type of database technology is to be used within a service. These are the DevOps skills that will be required to build and run successful microservice architecture. These skills certainly do not come cheap but they are mandatory to successful deployment.

Distributed Systems

The development team is going to be dealing with interfacing distributed systems – this is a fact with microservices. As a result, it is vital to have strong knowledge of network layer technologies, protocols and how they are optimized and secured. Again, this is not normally inherent in a software development team as they typical work with internal method calls. With microservice, the shift is towards RPC, HTTP REST and AMQP and these have to be understood and tested for latency and optimized for performance.

Maintenance

One of the biggest challenges when developing a microservice architecture is to go from the preproduction proof of concept type models to the preproduction stage. It is here that development teams discover that monitoring and testing for availability does not come inbuilt or naturally to the microservice architecture. Unlike in the monolithic environment where one needs to check for server availability and service awareness/response within a microservice environment there is a necessity to check many more distributed servers and services to ensure availability and responsiveness. This requires inbuilt test software to be delivered and operational for the network operational team in the NOC.

Conclusion

Microservice provides the solutions for many modern day application and data center scaling issues. It also provides a way for the developer to use different technologies and tools to overcome specific problems freeing them from the traditional technology stack. This alone provides a way for established monolithic applications such as core websites to be updated by building new features as microservices. This provides a way for older monolithic applications to remain untouched but to appear to have the latest innovative tools and techniques.

However, microservices are not the solution for every application as they do have inherent drawbacks, such as increased reliance on interfacing, rising costs and expensive skill sets.

Despite these drawbacks, the microservice architecture has proven itself in the most testing of enterprise and cloud environments performing with distinguish at both Netflex and Amazon.