Leveraging Microservices as an Application Architecture

Damian Nolan

***Abstract*—Microservice Architecture is an approach in software engineering and design that aims to divide an application into small constituents that are highly cohesive and loosely coupled. The following paper is intended to give an overview and exploration into the architectural style. Microservice Architecture has gained extensive exposure in the last number of years - Throughout this paper we will introduce Microservice Architecture as a concept, discuss Microservices as a subset of Service Orientated Architecture (SOA), compare and contrast Microservices with traditional Monolithic Architecture and review Docker as a containerisation platform for building Microservice applications.**

I. Introduction

THE term "Microservice" does not hold a true or set definition it is merely a term introduced by a number of software architects at a workshop near Venice in May, 2011 to provide context to a number of reoccurring design principles and patterns that seemed to be emerging more frequently. Just under a year later, James Lewis presented at 33rd Degree, conference for Java Masters in Krakow where he spoke about building systems composed of systems and focusing on the Unix philosophy of minimalist and modular. [1] This gave rise to the hot new term "Microservices" and began to get more and more attention.

Martin Fowler describes the Microservice Architectural style as a approach to developing a single application as a suite of independently deployable services each running in its own process [2]. Each service should be independent in its our right and provide functionality based around a single responsibility. Services should be loosely coupled, provide high cohesion and adhere to the single responsibility principle. All of a sudden we can see that many of the traits we mentioned are key concepts in Object Orientated Programming which are also common in the design principles of Service Orientated Architecture. Stubbings and Polovina [3] explore and contrast the how Object Orientated expertise can be leveraged in the design of Service Orientated Architecture (SOA).

II. Are Microservices just SOA?

This immediately sparks the debate - Is Microservice Architecture really just SOA? Upon the rise of the term "Microservices" into the community, many SOA architects and engineers were coming forward, saying "We've been doing this for years!". And as Martin Fowler mentions in his keynote[4] - "Service Orientated Architecture is a very broad term". It encompasses a vast landscape of design concepts, principles, patterns and implementation standards. And it is often considered controlled by vendors who release commercial solutions Fowler goes on to mention that in essence Microservices can be really be considered a subset of SOA.

Indeed, this is a popular and well defined statement and is backed up by Adrian Cockcroft, the man responsible for pioneering the architectural style at Netflix as they moved towards cloud based and Microservice orientated design. He describes the architecture as "fine-grained SOA", in his keynote at Dockercon in 2014 [5].

Netflix are widely considered to be part of the pioneers of the Microservice Architectural style and have helped pave the way for it to flourish, introducing such libraries and tools as Hystrix[6] and Chaos Monkey[7] for testing and strengthening systems.

III. Monolith vs MicroServices

Josh Evans of Netflix provides the analogy of the human body to describe the architectural style in his keynote [8]. He describes Microservices as the organs which make up the human body and work together as one organism. Perhaps the best way to put Microservices into context is by comparing the differences in contrast to a traditional monolithic architecture. Martin Fowler uses a similar example in his Resource Guide to Microservices where he outlines a potential structure of a traditional enterprise application. [2] Take for example a simple application that exists in 3 tiers.

A. Presentation Layer

A Presentation Layer sits at the top of the application. This is the entry point of an application from a customer or user perspective, more commonly referred to as the front end. For example, this layer may consist of a web application composed of HTML, CSS and Javascript presenting a user interface to the client.

B. Logic Layer

A Logic Layer lies behind the user interface and provides functionality to carry out business operations. This could be a wide variety of functionality and responsibilities could be anything from user accounts to instant messaging to calendar events. For this purpose, lets take a Java server application as the example. Requests may be issues over HTTP via REST[9] or RPC and handled accordingly by a number of object orientated classes to perform the requested operation.

C. Data Layer

The Data Layer lives at base of the application stack and is responsible for persistence of the raw application data. All of

the data needed and in use by the application exists here. A data store such as a relational database could be employed to persist and manage data in this layer. The server application may employ a framework such as Hibernate[10] JPA for object relational mapping in order to perform actions on data.

The monolith in this example is the server application. The services it provides run on a single process and are not independently deployable. A monolith is a software application whose modules cannot be executed independently [11]. In the case of the example defined above we may need to make improvements, changes or simple bug fixes to any one of the services we described. Changes made to the source code of the application imply - rebuild and redeploy. But what if the changes are small and confined to one service? There is an overhead associated with this as it becomes expensive to build a new release for production and this may only be affordable every couple of weeks. Continuous integration and continuous deployment are considered hidden dividends of Microservices [12]. Although not exclusive to the world of Microservices they may in theory be more easily orchestrated as each service has a single responsibility, however require careful orchestration. This allows developers to focus on developing and can speed up productivity across teams.

IV. Microservices Characteristics

To migrate the example above to a Microservice based architecture the functionality described for handling user accounts, instant messaging and calendar events would be refactored into individual services that are independently deployable and each run in an isolated process. This provides a more loosely coupled and highly cohesive design, adhering more so to the single responsibility principle mentioned previously. However, that being said this does not mean it is an easy task to move from one architecture to another. Particularly in the case of Microservices there a wide range of aspects to be taken into consideration. Knoche writes of using modernization paths to predict performance impact on systems migrating from monolithic software applications towards microservices [13]. Microservices may not hold a definition set in stone however, it is clear that there is a number of definitive common characteristics that can be used to identify with architectural style.

1. Independent Deployment

Independent deployment is a key characteristic and indeed perhaps the most important principle of Microservice architecture. The idea of having the ability to deploy services as independent applications implies a number of benefits.

Language Neutrality: A collection of Microservices operating together for the purpose of a single application do not have to be confined to one programming language. Sam Newmann uses the term - Technology Heterogeneity [14]. A service can use the best tools for the job and there may be various services written in completely different languages using different technologies.

Specialised Teams: Teams working on their respective services can become experts in the area and obtain a great understanding of how to carry out their jobs to the best of their ability. Having teams focus on their respective services may speed up productivity [14]. Why be a jack of all trades and master of none?

Appropriate Scaling: Independent Deployment lends itself to scaling in the right places. In terms of a traditional monolithic piece of software, the monolith may be replicated numerous times to achieve desired scaling. This infers the question - is every aspect of the system being used enough to justify increased scaling of the entire system. Independent deployment offers scaling of individual services where appropriate, the term horizontal scaling is used [8].

2. Bounded Context

The term ”Bounded Context” is used when describing the characteristics of Microservice Architecture. This pattern is key in Domain Driven Design [15]. Bounded context in Microservices refers to organisation and creation of services based upon their business logic. A service should have a single responsibility and that responsibility should be entirely encapsulated within that service. This also lends itself to the idea of specialised teams as previously mentioned. Fowler speaks in his keynote about how Amazon divided themselves into teams responsible for a section of business logic all the way through to the end-user experience [4].

3. Communication

In a cloud or distributed environment service interactions with one another can be carried out via IPC (inter-process communication) or protocols such as HTTP are commonly applied. Many Microservices communicating together in a chain through HTTP may suggest a certain amount of latency being involved, depending on what operations are being carried out. To combat situations like this, Google introduced gRPC[16]. gRPC is as open source RPC framework that employs Protocol buffers[17] as a method of serializing structured data. It is highly efficient and is designed to target speed and performance. Binary data is sent through the wire at high speeds, alongside rapid serialisation/deserialization [18].

In order to provide communication via services or processes it is standard to use light-weight mechanisms [11]. This also lends itself to the idea of smart endpoints and dumb pipes[2]. The services themselves become the endpoints where business logic lives and pipes refer to the mechanisms used for them to communicate which are kept simple.

4. Decentralisation

Decentralisation of services infers a substantial amount of additional complexities within an application ecosystem. How does a client know what service it needs to contact in a cloud or distributed network? API Gateways[19] and Service Discovery[20] are two critical concepts in Microservice Architecture. An API Gateway uses a Facade pattern[21] from object orientated design to encapsulate how requests are made

to a number a services. This minimises code complexity on the client side and eliminates direct calls to services made by the client. The API Gateway is used in conjunction with a Service Discovery pattern. This may be either client side or server side. Service Discovery employs the use of a service registry, a database of network locations of available service instances[20].

V. Containerisation with Docker

Docker [22] is a containisation platform that utilises the host operating system using lightweight images at the application layer to build and ship applications effectively and quickly. Containers can be considered as instances of images running in isolated processes. They are often compared to virtual machines however they are more lightweight and can share the host OS kernal with many other containers. Virtual machines often take a couple of minutes to start whereas containers launch in a number of seconds. Container images are substantially smaller than virtual machines and are typically tens of MBs in size [22]. In the last number of years Docker has become an industry leader in the world of Microservice Architecture. Sinnott and Korzhirbayev carried out a comparison of container-based technologies for the cloud[23] where they analyse Docker and evaluate pros and cons of container-based technologies based on in depth research of a number of performance metrics such as CPU, Disk I/O and Network I/O performance.

Containers are built with Docker using Dockerfiles[24]. A Dockerfile is a declarative file in which instructions for images containing applications or services are defined. For example, a basic Dockerfile may include:

- A base image
- Installation of Dependencies
- Adding of source code
- Exposing ports to the host OS
- Commands for starting an application

Example Dockerfile

```
# Base Image
FROM node:alpine

# Specify a working directory
WORKDIR /app

# Add package.json and install dependencies
ADD package.json /app/package.json
RUN npm install

# Add application source code
ADD . /app

# Expose port 3000 on host machine
EXPOSE 3000

# Start the server
CMD ["node", "server.js"]
```

More detailed and complex configurations can also be defined such as environment variables and volumes. Volumes provide a way in which containers can persist data. Smart, Nguyen and Jaramillo provide excellent analysis of how Docker technology can be leveraged in Microservice Architecture [25].

Docker Compose

Docker compose is a higher level abstraction from Dockerfiles. Multiple images can be defined using a docker compose file with their build context referencing the path of a Dockerfile for the service container. Similar to Dockerfiles a number of configuration settings can be defined here. John Zaccone provides an excellent example using Docker Compose in his keynote at Dockercon 2017, the source code of his demo application can be found on his Github [26].

Container Orchestration

Container Orchestration refers to management of containerised software with regard to automation tools. Docker Swarm [27] is a feature provided by Docker for container orchestration in a cloud or distributed environment. The goal of Swarm is to supply a level of transparency for container management across a number of hosts. In terms of Microservice Architecture, Docker Swarm could be considered to be in direct competition with Google's Kubernetes [28] and Amazon's AWS in the domination for Infrastructure as a Service (IaaS) and particularly container orchestration.

VI. Conclusion

While this review may suggest that Microservice Architecture is a fantastic solution for many reoccurring problems in an application design, it should be considered by no means a silver a bullet [14]. Microservice Architecture certainly adds a number of complexities and is often considered very risky to begin with as adopting this style can be very difficult to get right. Every application has different requirements and perhaps a simpler approach should be taken in the beginning. It is not uncommon to start with a monolith and later refactor to a Microservice based design. CTO of NPM Laurie Voss explains in his keynote how NPM began as a monolith and later refactored to a Microservice Architecture and is now the world's largest software registry[29]. So while Microservices and indeed Docker may not be the cure to address every aspect of software design and architecture, they both definitely have a lot promising and helpful tools to offer.

References

[1] James Lewis. "Microservices, Java - the Unix Way". In: Mar. 2012. URL: https://vimeo.com/74452550.

[2] Martin Fowler James Lewis. *Microservices Resource Guide*. 2014. URL: https://martinfowler.com/microservices/.

[3] Gary Stubbings and Simon Polovina. "Levering object-oriented knowledge for service-oriented proficiency." In: *Computing* 95.9 (2013), pp. 817–835. ISSN: 0010485X. URL: http://search.ebscohost.com/login.aspx?direct=true&db=bth&AN=90053029&site=eds-live.

[4] Martin Fowler. "GOTO Conference Berlin 2014, Microservices". In: 2014. URL: https://www.youtube.com/watch?v=wgdBVIX9ifA.

[5] Adrian Cockcroft. "State of the Art in Microservices". In: 2014. URL: https://youtu.be/nMTaS07i3jk.

[6] Netflix. *Hystrix*. URL: https://github.com/Netflix/Hystrix.

[7] Netflix. *Chaos Monkey*. URL: https://github.com/Netflix/chaosmonkey.

[8] Josh Evans. "Mastering Chaos - A Netflix Guide to Microservices". In: QCon San Francisco, 2016. URL: https://www.infoq.com/presentations/netflix-chaos-microservices.

[9] Roy Thomas Fielding. "Architectural Styles and the Design of Network-based Software Architectures". AAI9980887. PhD thesis. 2000. ISBN: 0-599-87118-0.

[10] RedHat JBoss Middleware. *Hibernate ORM*. URL: http://hibernate.org/orm/.

[11] Nicola Dragoni et al. "Microservices: yesterday, today, and tomorrow". In: (June 2016).

[12] TOM KILLALEA. "The Hidden Dividends of Microservices." In: *Communications of the ACM* 59.8 (2016), pp. 42–45. ISSN: 00010782. URL: http://search.ebscohost.com/login.aspx?direct=true&db=bth&AN=117173592&site=eds-live.

[13] Holger Knoche. "Sustaining Runtime Performance While Incrementally Modernizing Transactional Monolithic Software Towards Microservices". In: *Proceedings of the 7th ACM/SPEC on International Conference on Performance Engineering*. ICPE '16. Delft, The Netherlands: ACM, 2016, pp. 121–124. ISBN: 978-1-4503-4080-9. DOI: 10.1145/2851553.2892039. URL: http://doi.acm.org/10.1145/2851553.2892039.

[14] Sam Newmann. *Building Microservices*. O'Reilly, 2014. ISBN: 9781491950357.

[15] Martin Fowler. *Bounded Context*. URL: https://martinfowler.com/bliki/BoundedContext.html.

[16] Google. *gRPC*. URL: https://grpc.io/.

[17] Google. *Protocol Buffers*. URL: https://developers.google.com/protocol-buffers/.

[18] Joey Clover. *Microservices are hard - an invaluable guide to microservices*. URL: https://hackernoon.com/microservices-are-hard-an-invaluable-guide-to-microservices-2d06bd7bcf5d.

[19] Nginx. *Building Microservices: Using an API Gateway*. URL: https://www.nginx.com/blog/building-microservices-using-an-api-gateway/.

[20] Nginx. *Service Discovery in Microservices Architecture*. URL: https://www.nginx.com/blog/service-discovery-in-a-microservices-architecture/.

[21] Erich Gamma et al. *Design Patterns: Elements of Reusable Object-oriented Software*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1995, p. 175. ISBN: 0-201-63361-2.

[22] *Docker*. URL: https://www.docker.com/what-docker.

[23] Zhanibek Kozhirbayev and Richard O. Sinnott. "A performance comparison of container-based technologies for the Cloud". In: *Future Generation Computer Systems* 68.Supplement C (2017), pp. 175–182. ISSN: 0167-739X. DOI: https://doi.org/10.1016/j.future.2016.08.025. URL: http://www.sciencedirect.com/science/article/pii/S0167739X16303041.

[24] Docker. *Dockerfile reference documentation*. URL: https://docs.docker.com/engine/reference/builder/.

[25] Robert Smart David Jaramillo Duy V Nguyen. "Leveraging Microservices Architecture by using Docker technology". In: (Mar. 2016).

[26] John Zaccone. *Office Space Dockercon Demo*. URL: https://github.com/jzaccone/office-space-dockercon2017.

[27] Docker. *Docker Swarm*. URL: https://docs.docker.com/engine/swarm/.

[28] Akshai Parthasarathy. *Kubernetes vs Docker Swarm*. URL: https://platform9.com/blog/kubernetes-docker-swarm-compared/.

[29] Laurie Voss. "How npm split a monolith and lived to tell the tale". In: 2016. URL: https://www.oreilly.com/ideas/how-npm-split-a-monolith-and-lived-to-tell-the-tale.