Leveraging microservices architecture by using Docker technology

David Jaramillo
Cloud Engineering and Services
IBM
Boca Raton, FL, USA
djaramil@us.ibm.com

Duy V Nguyen
Cloud Engineering and Services
IBM
Boca Raton, FL, USA
dnguyenv@us.ibm.com

Robert Smart
Emerging Technologies
IBM
Hursley, United Kingdom
smartrob@uk.ibm.com

***Abstract*— Microservices architecture is not a hype and for awhile, started getting attention from organizations who want to shorten time to market of a software product by improving productivity effect through maximizing the automation in all life circle of the product. However, microservices architecture approach also introduces a lot of new complexity and requires application developers a certain level of maturity in order to confidently apply the architectural style. Docker has been a disruptive technology which changes the way applications are being developed and distributed. With a lot of advantages, Docker is a very good fit to implementing microservices architecture. In this paper we will discuss about how Docker can effectively help in leveraging mircoservices architecture with a real working model as a case study.**

Keywords—micoservices; docker; devops; automation

I. INTRODUCTION

A common way to build software applications until now is the monolithic approach where one deployment unit has several responsibilities and in some cases it does mostly everything. Monolithic approach is still good for small scale teams and projects, but when scalability, flexibility and other requirements like fast development, short time to market, wider team collaboration, and so on become more and more critical to achieve business competitiveness, monolithic starts becoming a big barrier.

It's much harder to make changes to the application in responding to radical change requirements from users or business model frequently, as the code base becomes bigger, more complicated with more people making changes to it. The tightly coupled model of monolithic approach natually requires incremental amount of effort to coordinate to make any must-have updates which consequently slows down the release cycle of the application as well as contributes further to its fragility

Microservices architectural approach was introduced as a solution to solve the monolithic problem. Even though it's in theory supposed to address most of the problems occured in mononithic approach, but Microservices also has a long downsides list which emphasizes the requirement to have a certain level of maturity in automation and agility for the developers to be able to make use of it's advantages. Containerization technologies where Docker is leading is really accelerating the application of microservices architecture in a lot of use cases.

II. MICROSERVICES ARCHITECTURE

Microservices are small autonomous services that work together to fulfill a business requirement. This section will discuss about some basic concepts and characteristics in micoservice architecture.

A. Small and focussed

Even though "small" is not a very good measure to describe microservices but we can use it as an attempt to emphasize one of the most important characteristics of microservices that is each service is fine-grained, high cohesion to focus on fulfilling a granular responsibility.

In the enterprise context, microservices should be designed with business oriented driver in mind. That means the resulting services should not mimic organization, technological or communication boundaries. In stead, they should be modelled around specific business domain.

From development perspective, each service should be treated as an independant application with its own source code repository and delivery pipeline.

B. Loosly coupled

Loose coupling is an essential characteristic of microservices. Each microservice needs to be deployed as

978-1-5090-2246-5/16/$31.00 ©2016 IEEE

needed without the necessary of coordination with other services' owners. If you have two services and you are always releasing those two things together in one big deployment, that might be a sign two of them should be only one thing and there are further works to be done against the current services decomposition. The loose coupling enables more frequent and rapid deployments that eventually improves the responsiness of the application in response to its users' needs.

C. *Language-neutral*

Microservices need to be built using technology that's the developers are most comfortable with. Development teams should not be dictated by any programming languages being used, meaning that micoservices architecture leverages freedom in using technologies that make the most sense for the task and people performing the task. This makes it easier to take full advantages of the most optimal technologies and skills the teams have.

Because microservices are language-neutral, the communication among them also through a language-neutral application programming interface (API), typically an HTTP-based resource API like REST. Figure 1 below describes an example where variety of languages, technologies could be used to build a typical online shopping system with microservices approach. The services are developed using different programming languages and development frameworks like Java, PHP or Node.js. Each service even has its own type of data storage in which Catalog service uses Cloudant, Order service uses SQL Database and so on.

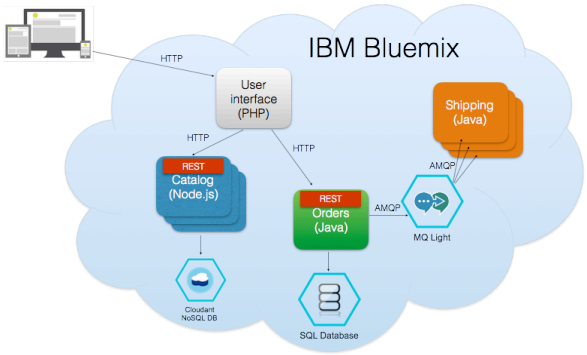

Figure 1: *Language-neutral in building microservices*

D. *Bounded context*

A bounded context encapsulates details of a single domain like data model, domain model, and so on. It also defines the integration points with other bounded ones. In microservices architecture, it's critical to have well defined bound context. This means the more boundaries between domains are explicitly well-defined, the more we're able to reason about the services designing, sizing efficiently. There are times we have models that are shared between those boundaries, as well as models that really only need to exist inside of each boundary area.

E. *Challenges in build a microservices architecture*

Besides a long list of advantages, microservices architecture style, on the other side, introduces several challenges that need to be addressed before being able to produce benefit.

1) Failure isolation

Failures will happen, it's just a matter of time and when they do happen with particular services, they better fail fast. Quick failures lead to better understanding and problem solving. Also we need to have a divide to concur model where we can break things into smaller chunks and with the fast tooling being used or created towards aiming to support continuous delivery of many tiny changes so that developer can change one thing at a time and if it breaks, we know that's the only thing that broke.

Design for failure is an important requirement in successfully building a microservices based system.

2) Observability

Building a microservices architecture needs a way to visualize the health status of every services in the system to quickly locate and respond to any problem occur. This also includes a comprehensive logging mechanism to record, store, make it searchable for better analysing the logs. It's much more challenging than just a large number of items to be added beyond the look out. With all the fast moving parts, the observation design needs to be somewhat that makes sense the data being visualized to provide helpful information as input for analysing.

3) Automation requirement

A culture of leveraging automation is upmost important and the most challenged item in the list. The explosion of the number of services and relationships between them will happen at some point, probably very soon that can not be handled without a way to automate things.

4) High independancy

One of major principles of a microservices based system is making services highly decoupled. That means it's critical to keep the independancy between services so that each of them can be developed, deployed independantly without effecting each other.

5) Testing

Testing is in the list of the most challenging items while building microservices architecture. More moving parts means more chances for failures to occur. How to make sure the test comprehensive enough to cover all aspects is not easy. Automated testing has advanced significantly over time as new tools and techniques keep being introduced. However, challenges remain as for how to effectively and efficiently test both functional and non-functional aspects of the system, especially in a distributed model. With more independent components and patterns of collaborations among them, microservices architecture even add a new level of complexity into the tests [1].

6) Scalibility

Even though one of the important motivations of building a microservices architecture is to solve the scalability problem, but scaliblity is also a very tough challenge of microservices architecture itself. At some point there will be explosion in the number of micoservices being created in your system, not to

mention different versions for each one, resulting the explosion of the number of connections among them which adds to the complexibility of a microservices based system and requires special solutions to be put in place like service discovery to know which service is running where, a routing mechanism to route the traffic through services' APIs, a better configuration management to dynamically do the configuration and apply changes to the system and so on.

III. DOCKER

Docker is an open platform for quickly developing, shipping and running software applications. It's designed to help delivering applications faster by using a lightweight container virtualization platform surrounded by a set of tools and workflows that help developers in deploying and managing application easier [2].

We can have a quick idea about Docker containers by associating it with a container ship where its containers are loaded on to the ship and then shipped and unloaded to different locations. Docker container is basically where you can create applications, place them together with all required dependency components for the apps to be able to run inside a hardened box and then put it through any verification or rigorous testing as needed for quality ensurance purpose. The creation of the box can be done on almost any operating system and infrastructure, including cloud environment and it's espeically very fast thanks to the leveraging of unified systems and other techniques which will be discussed later.

Below are some more details information about Docker technologies.

A. *Docker architecture*

Docker uses a client-server architecture. Figure 2 below illustrates architecture overview of Docker with major components and how they interact with each other.

Docker users use a set of Docker commands from the Client component to interact with the Docker host through the Docker deamon which runs on the host. Docker deamon in turn does the building, running, and distributing Docker containers as well as publishing Docker images to particular Docker registry. Docker client can either run on the same system with Docker deamon or on a diffrent one then remotely intereacts with the deamon. The Docker client component communicates with the Docker deamon through sockets or via a set of RESTful API. Details about each component can be found in the following section.

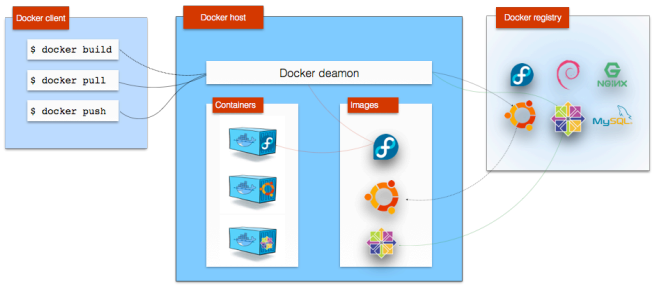

***Figure 2**: Docker architecture overview*

B. *Docker major components and concepts*

1) *Docker image*

An image is a read-only template, a build component of Docker from which Docker containers are launched. Every image starts with a base image. Other parts are then added on top of the base one as needed in order to fulfill business responsibility of the image. For example, an image could contain a minimum base CentOS operating system and then a middleware like Node.js engine would be added, and finally your web application installed on top of that.

With Docker, images can be newly built locally or downloaded, updated from an existing remote source repositories where the images were created by other people. If we dive deeper, each image consists of several layers and Docker uses union file systems to combine those layers into a single image. Union file systems (UFS) is a filesystem service for Linux, FreeBSD and NetBSD OS which implements a union mount for other file systems. It allows files and directories of separate file systems, known as branches, to be transparently overlaid, forming a single coherent file system [2]

The use of UFS in Docker helps to accelerate the portability and lightweight characteristics of the containerization technology. For instance, when an image is changed in the event like the middleware or application is updated with new version, in the making of new image, instead of replacing the whole image or rebuilding from scratch, only associated new layer gets added or updated. Once the build is done, you only need to distribute just the particular updated layers, making distributing Docker images more lightweight and consequently faster and simpler.

The building of Docker images is through a simple, descriptive set of steps which are called *instructions*. It starts with a Docker specific file called Dockerfile where the instructions are declaratively stored. When a build request is triggered, Docker reads the Dockerfile for the instructions to execute and then returns final image. Each instruction creates a new layer in the resulting image. Figure 3 below describes an example of how the instructions look like in a typical Dockerfile

```
FROM debian:jessie

MAINTAINER NGINX Docker Maintainers "docker-maint@nginx.com"

RUN apt-key adv --keyserver hkp://pgp.mit.edu:80 --recv-keys 573BFD6B3D8FBC641079A6ABABF5BD827BD9BF62
RUN echo "deb http://nginx.org/packages/mainline/debian/ jessie nginx" >> /etc/apt/sources.list

ENV NGINX_VERSION 1.9.9-1~jessie

RUN apt-get update && \
    apt-get install -y ca-certificates nginx=${NGINX_VERSION} && \
    rm -rf /var/lib/apt/lists/*
# forward request and error logs to docker log collector
RUN ln -sf /dev/stdout /var/log/nginx/access.log
RUN ln -sf /dev/stderr /var/log/nginx/error.log

VOLUME ["/var/cache/nginx"]

EXPOSE 80 443

CMD ["nginx", "-g", "daemon off;"]
```

***Figure 3**: An example of a Dockerfile*

2) *Docker registry*

Docker images are usually stored on shared repositories called Docker registries. The registries are where you can download images from or upload images to using Docker

client set of commands, which make the distribution of the images easy and together with other parts of Docker technology, they contribute to creating a whole new way of distributing applications the developers can have. Just like working with source code management.

Docker has a public registry named Docker Hub which is a big collection of existing images ready for use. People can freely create new images and add them to the collection or share them on a privately held repository running behind an interprise firewall as needed.

3) Docker container

A Docker container is an isolated and secure environment which contains everything necessary for an application to run, typically including a shared operating system (Docker container does not include the guest operating system but shares it with other containers), necessary middleware, application specific user-added files and other metadata. A Docker container is created from a Docker image that can then be run, started, stopped, or removed at will by a very little effort. The image instructs Docker what process to run, and a variety of other necessary configuration meta data at the time the container is launched. Docker image is read-only but when Docker runs a container from an image, it adds a read-write layer on top of the image using UFS in which the application can then run.

IV. DOCKER IS A GOOD FIT FOR MICROSERVICES

As mentioned in previous session about microservices architecture, the approach does come with a list of challenges need to be addressed like the higher level of complexity from development to going on production, the critical requirement of automation in every aspects, failure isolation, testing challenges, and so on. Fortunately Docker has been introduced with key features and tooling around which can help addressing those challenges. Lets discuss some highlighted ones

A. Accelerate automation.

Docker containers are naturally very well fit for microservices architecture since each of them can be used as a deployment unit to granularly contain a service. Because the creation and launching of every container is scriptable by design, and with plenty of tools available over time, Docker container is accelerating automation culture in every step of software development lifecycle.

B. Accelerate the independency.

Each Docker container is an isolated box which can contain run time environment for a particular service. With the wide range of platforms being supported by and adopting Docker, the service's development team can independantly work on implementing the service with whatever technology or language, process, tools they're comfortable with the most.

C. Accelerate portability.

Docker puts application and all of its dependencies into a container which is portable among different platforms including Linux distributions and clouds. Different stakeholders of the application like developers, testers, administrators, and so on can quickly run same application on virtual machines, local laptops, bare-metal servers or in the cloud at will. This especially helps to do independant isolated testing of a particular service in microservices architecture.

D. Accelerate resource utilization.

Even though it's not explicitly written down as a principle but being lightweight, portable are implication requirements towards being a micoservices architecture friendly environment.

In Docker, each container is constituted by just the application and the dependencies which the app needs to be able to run, ideally neither more nor less. The container then runs as an isolated process on the host operating system, sharing the kernel with other containers. Thus, if the container is placed in a VM environment, besides making use of the resource utilization benefits from using virtual machines the containerization technique is even making it more portable and efficient.

In a bare-metal environment, the lightweight nature of Docker containers helps to create and run more instances than virtual machines resulting higher resource utilization.

E. Secured.

A lot of what Docker is offering are allowing developers to flexibly maximize the security of the code at different levels. When building the code, developers can freely use penetration test tools to stress test any part of build cycle. Since the source for building Docker images are explicitely, declaratively described in the Docker build, distribution components (eg Dockerfile, docker-compose file), developers can handle the image supply chain easier and be able to force, mandate security policies as needed. Also the ability to easily hardened immutable services by putting them into Docker containers adds strong security insurance for the services.

V. CASE STUDY - A WORKING MODEL

As an example use case, our team had been using the traditional monolithic model for building applications for a long time. We understand the incremental effort we have to spend in development, deployment as well as the overhead in each release of the app where a lot of meetings and coordination were required between dev and ops teams.

We wanted to make things better after each project and decided to evolutionaly approach microservices architecture principles in recent application releases. Docker is really a rescue in our microservices architecture adoption steps. Lets discuss about how it's been done so far in our roadmap.

A. Solution architecture overview.

The case study we will discuss is basically a solution including an updated working flow, process and an application that can help to innovate the way internal charge back activities between departments being done which is essentially a typical internal cost coverage in an enterprise environment.

Figure 4 below describes the architecture overview of the evolving system where microservices based approach has been backed by maximizing the use of Docker technologies including the tooling of the Docker ecosystem.

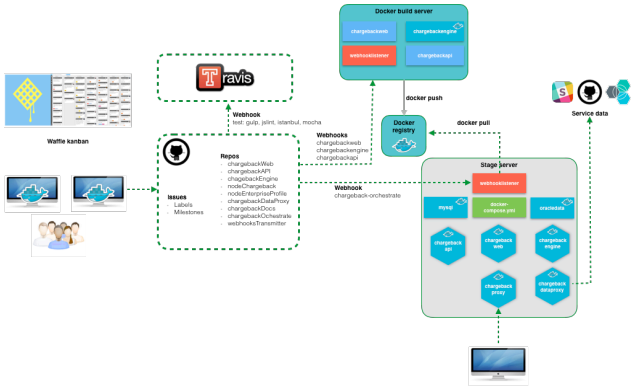

***Figure 4**: Architecture overview of enterprise charge back system which leverages microservices archiecture and Docker*

B. *How it works.*

The solution being discussed in the case study is not only leveraging Docker technology, but also strictly following agile methodology in software development and enjoying surrounding tools' benefits like Waffle, Github, Slack and so on.

The system was setup up so that everytime a developer pushes a set of commits to a shared source code repository which is dedicated to a particular service, notifications will be sent to group of people who are major stakeholders of the system including system owner, other developers, testers, etc and whoever subscribed to the group. This visualization helps to promote the engagement, colaboration between different stakeholders and maximize personal responsibility awareness of the developer in every line of code and somewhat isolate potential bugs.

We use the webhooks principles in Github to setup alerts out of Github so that it's able to say when a certain event has occured. Thus, once the changes are reviewed (including peer reviews, discussion through different channels) verified and merged to main code stream, a comprehensive set of automated testing will be triggered to run against the changes. That's achieved by using Travis as a testing framework so that whenever a pushing to the mainstream of a repository is received, Travis will run any tests contained within the associated git project and say whether they passed or failed. We use community-proved test tools like jslint, istanbul, mocha for automating the test activities. To further ensure the quality of code, a high coverage testing was set to make sure mostly every line of code developed will be scanned and tested.

Once all the tests are passed, a build will be triggered for the repository. We built up a special component named *webhooklistener* which is an integration, event-based utility that listens for configured webhooks from github and then performs a series of scripted actions like a demand to build a docker image, push the image to a registry, and so on.

A docker image will then be created for the service of that particular repository including appropriate versioning mechanism so that we can keep track of the level of code in the images and eventual Docker containers later on. The image will then be pushed to a dedicated private Docker registry of the project to be ready for any docker pull requests for launching Docker containers as needed. So the *webhooklistener* component is doing more than what implicated in its name, but mostly concentrating on building and delivering Docker images to appropriate image registry.

The special chageback-orchestrate project is a place to store deployment scripts for different environments. We use Docker compose tool of the Docker eco-system to instrument the containers declaratively so that we can quickly replicate or setup a new running environment as needed. There is still some manual activities involved for this part which is whenever there is a need to deploy a new version of the application to a particular server, we need to manually update the docker compose file with corresponding containers' versions and then push it to the dedicated Github repository. That pushing will trigger a webhook payload that in turn kicks off a new deployment through the *webhooklistener* component.

With the current model, we can also rollback to any version of the application as needed just by mapping the corresponding version of docker containers were built in that particular code level in the docker compose file and push it back to the mainstream dedicated chageback-orchestrate github repository.

C. *Improvement needs .*

Even though the current model in the case study is working well with high satisfaction from different stakeholders of the project, it's still evolving towards a more automated, scalable design, especially the auto scaling and observability aspect (specifically logging, mornitoring) which are big items in our backlog list that need to be done next to over come scalability challenge. But it is also very typical for any intention to apply microservices architecture where improvements can only be achieved through constantly evolving.

VI. Conclusion

In this paper we have discussed some concepts about microservices architecture and how Docker can help to successfully apply the beneficial architectural style with a well working case study. Even though Docker itself is not a silver bullet which can address every challenges of building a microservices architecture, but together with surrounding tools, it helps a lot in improving efficiency, automation and other necessary fundamentals in order to achieve the most important principles of building a microservices architecture.

References

[1] "Microservices from theory to practices" IBM Redbooks. http://www.redbooks.ibm.com/redbooks/pdfs/sg248275.pdf

[2] "About Docker" Docker documentation. https://docs.docker.com/engine/misc/

[3] "Understand the architecture" Docker documentation. https://docs.docker.com/engine/introduction/understanding-docker/