Microservices

Khateeja Ambareen[1], Shridhar K[2]
[1, 2] Department of Computer Science and Engineering
[1] GSSS Institute of Engineering and Technology For Women Mysore, Karnataka, India
[2] VSS Division, Chennai, Tamil Nadu, India

Abstract- *The Microservice architectural style is an approach to developing a single application as a suite of small services, each running in its own process and communicating with lightweight mechanisms, often an HTTP resource API. These services are built around business capabilities and independently deployable by fully automated deployment machinery. There is a bare minimum of centralized management of these services, which may be written in different programming languages and use different data storage technologies.*

A microservice architecture is the natural consequence of applying the single responsibility principle at the architectural level. This results in a number of benefits over a traditional monolithic architecture such as independent deployability, language, platform and technology independence for different components, distinct axes of scalability and increased architectural flexibility. Microservices are often integrated using REST over HTTP.

Keywords- REST over HTTP,MONOLITHIC ARCHITECTURe, MICROSERVICES.

I. INTRODUCTION

Distributed computing has been constantly evolving in the past two decades. During the mid-90s, the industry evaluated component technology based on Corba, DCOM and J2EE. A component was regarded as a reusable unit of code with immutable interfaces that could be shared among separate applications. The component architecture represented a shift away from how applications were previously developed using dynamic-link libraries, among others. However, the communication protocol used by each component technology was proprietary – RMI for Java, IIOB for Corba and RPC for DCOM. This made interoperability and integration of applications built on different platforms using different languages a complex task.

Evolution of microservices: With the acceptance of XML and HTTP as standard protocols for cross-platform communication, service-oriented architecture (SOA) attempted to define a set of standards for interoperability. Initially based on Simple Object Access Protocol, the standards for web services interoperability were handed over to a committee called Oasis. Suppliers like IBM, Tibco, Microsoft and Oracle started to ship enterprise application integration products based on SOA principles. While these were gaining traction among the enterprises, young Web 2.0 companies started to adopt representational state transfer (Rest) as their preferred protocol for distributed computing. With JavaScript gaining ground, JavaScript Object Notation (JSON) and Rest quickly became the de facto standards for the web.

Microservices are not just code modules or libraries – they contain everything from the operating system, platform, framework, runtime and dependencies, packaged as one unit of execution. Each microservice is an independent, autonomous process with no dependency on other microservices. It doesn't even know or acknowledge the existence of other microservices. Microservices communicate with each other through language and platform-agnostic application programming interfaces (APIs). These APIs are typically exposed as Rest endpoints or can be invoked via lightweight messaging protocols such as RabbitMQ. They are loosely coupled with each other avoiding synchronous and blocking-calls whenever possible.

II. RELATED WORK

"Microservices" became the hot term in 2014, attracting lots of attention as a new way to think about structuring applications. I'd come across this style several years earlier, talking with my contacts both in ThoughtWorks and beyond. It's a style that many good people find is an effective way to work with a significant class of systems. But to gain any benefit from microservice thinking, you have to understand what it is, how to do it, and why you should usually do something else. The term "Micro-Web-Services" was first introduced during a presentation at CloudComputing Expo in 2005 by Dr. Peter Rodgers. On slide #4 of the conference presentation he states that "Software components are Micro-Web-Services. Juval Lowry also had similar precursor type of thinking about classes being granular services, as the next evolution of Microsoft architecture. Services are composed using Unix-like pipelines (the Web meets Unix true loose-coupling). Services can call services (multiple language run-times). Complex service-assemblies are abstracted behind simple URI interface. Any service, at

any granularity, can be exposed". He described how a well designed service platform "applies the underlying architectural principles of the Web and Web services together with Unix-like scheduling and pipelines to provide radical flexibility and improved simplicity by providing a platform to apply service-oriented architecture throughout your application environment." [4] The motivation behind this design, which originated in a research project at Hewlett Packard Labs, is to make code less brittle and large-scale, complex software systems robust to change.[5] To make "Micro-Web-Services" work one has to question and analyze the foundations of architectural styles such as SOA and the role of messaging between software components in order to arrive at a new general computing abstraction.[6] In this case, one can think of Resource-Oriented Computing (ROC) as a generalized form of the Web abstraction. If in the Unix abstraction "everything is a file", then in ROC everything is a "Micro-Web-Service". It can contain information, code or the results of computations so that a service can be either a consumer or producer in a symmetrical and evolving architecture.

III. EXISTING SYSTEM

MONOLITHIC ARCHITECTURE

Let's imagine that you are building an e-commerce application that takes orders from customers, verifies inventory and available credit, and ships them. The application consists of several components including the StoreFrontUI, which implements the user interface, along with some backend services for checking credit, maintaining inventory and shipping orders[8].

The application is deployed as a single monolithic application. For example, a Java web application consists of a single WAR file that runs on a web container such as Tomcat. A Rails application consists of a single directory hierarchy deployed using either, for example, Phusion Passenger on Apache/Nginx or JRuby on Tomcat. You can run multiple instances of the application behind a load balancer in order to scale and improve availability.

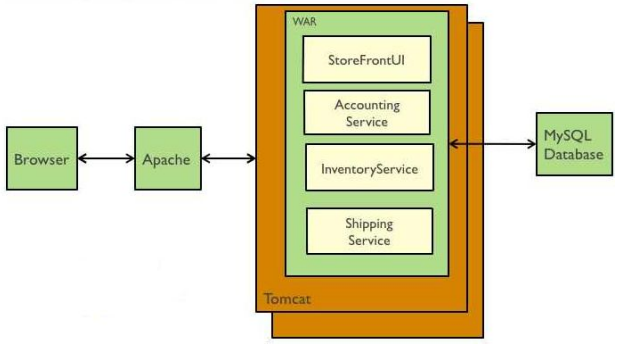

Fig. 1: Monolithic architecture

This solution has a number of benefits[2]:

- Simple to develop - the goal of current development tools and IDEs is to support the development of monolithic applications
- Simple to deploy - you simply need to deploy the WAR file (or directory hierarchy) on the appropriate runtime
- Simple to scale - you can scale the application by running multiple copies of the application behind a load balancer.

However, once the application becomes large and the team grows in size, this approach has a number of drawbacks[2] that become increasingly significant:

- The large monolithic code base intimidates developers, especially ones who are new to the team. The application can be difficult to understand and modify. As a result, development typically slows down. Also, because there are not hard module boundaries, modularity breaks down over time. Moreover, because it can be difficult to understand how to correctly implement a change the quality of the code declines over time. It's a downwards spiral.
- Overloaded IDE -the larger the code base the slower the IDE and the less productive developers are.
- Overloaded web container-the larger the application the longer it takes to startup.This had have a huge impact on developer productivity because of time wasted waiting for the container to start. It also impacts deployment too.
- Continuous deployment is difficult - a large monolithic application is also an obstacle to frequent deployments. In order to update one component you have to redeploy the entire application. This will interrupt background tasks (e.g. Quartz jobs in a Java application), regardless of whether they are impacted by the change, and possibly cause problems. There is also the chance that components that haven't been updated will fail to start correctly. As a result, the risk associated with redeployment increases, which discourages frequent updates. This is especially a problem for user interface developers, since they usually need to iterative rapidly and redeploy frequently.
- Scaling the application can be difficult - a monolithic architecture is that it can only scale in one dimension. On the one hand, it can scale with an increasing transaction volume by running more copies of the application. Some clouds can even adjust the number of instances dynamically based on load. But on the

other hand, this architecture can't scale with an increasing data volume. Each copy of application instance will access all of the data, which makes caching less effective and increases memory consumption and I/O traffic. Also, different application components have different resource requirements - one might be CPU intensive while another might memory intensive. With a monolithic architecture we cannot scale each component independently

- Obstacle to scaling development - A monolithic application is also an obstacle to scaling development. Once the application gets to a certain size its useful to divide up the engineering organization into teams that focus on specific functional areas. For example, we might want to have the UI team, accounting team, inventory team, etc. The trouble with a monolithic application is that it prevents the teams from working independently. The teams must coordinate their development efforts and redeployments. It is much more difficult for a team to make a change and update production.
- Requires a long-term commitment to a technology stack - a monolithic architecture forces you to be married to the technology stack (and in some cases, to a particular version of that technology) you chose at the start of development .With a monolithic application, can be difficult to incrementally adopt a newer technology. For example, let's imagine that you chose the JVM. You have some language choices since as well as Java you can use other JVM languages that inter-operate nicely with Java such as Groovy and Scala. But components written in non-JVM languages do not have a place within your monolithic architecture. Also, if your application uses a platform framework that subsequently becomes obsolete then it can be challenging to incrementally migrate the application to a newer and better framework. It's possible that in order to adopt a newer platform framework you have to rewrite the entire application, which is a risky undertaking.

IV. PROPOSED SYSTEM

The term "Microservice Architecture" has sprung up over the last few years to describe a particular way of designing software applications as suites of independently deployable services[1]. While there is no precise definition of this architectural style, there are certain common characteristics around organization around business capability, automated deployment, intelligence in the endpoints, and decentralized control of languages and data.

A microservice architecture is the natural consequence of applying the single responsibility principle at the architectural level[8]. This results in a number of benefits over a traditional monolithic architecture such as independent deployability, language, platform and technology independe nce for different components, distinct axes of scalability and increased architectural flexibility.

Microservices are often integrated using REST over HTTP. A microservice architecture promotes developing and deploying applications composed of independent, autonomous, modular, self-contained units.

A component was regarded as a reusable unit of code with immutable interfaces that could be shared among disparate applications.

To start explaining the microservice style it's useful to compare it to the monolithic style: a monolithic application built as a single unit. Enterprise Applications are often built in three main parts: a client-side user interface (consisting of HTML pages and javascript running in a browser on the user's machine) a database (consisting of many tables inserted into a common, and usually relational, database management system), and a server-side application. The server-side application will handle HTTP requests, execute domain logic, retrieve and update data from the database, and select and populate HTML views to be sent to the browser. This server-side application is a monolith - a single logical executable[2]. Any changes to the system involve building and deploying a new version of the server-side application.

Such a monolithic server is a natural way to approach building such a system. All your logic for handling a request runs in a single process, allowing you to use the basic features of your language to divide up the application into classes, functions, and namespaces. With some care, you can run and test the application on a developer's laptop, and use a deployment pipeline to ensure that changes are properly tested and deployed into production. You can horizontally scale the monolith by running many instances behind a load-balancer.
Monolithic applications can be successful, but increasingly people are feeling frustrations with them - especially as more applications are being deployed to the cloud . Change cycles are tied together - a change made to a small part of the application, requires the entire monolith to be rebuilt and deployed. Over time it's often hard to keep a good modular structure, making it harder to keep changes that ought to only affect one module within that module. Scaling requires scaling of the entire application rather than parts of it that require greater resource.

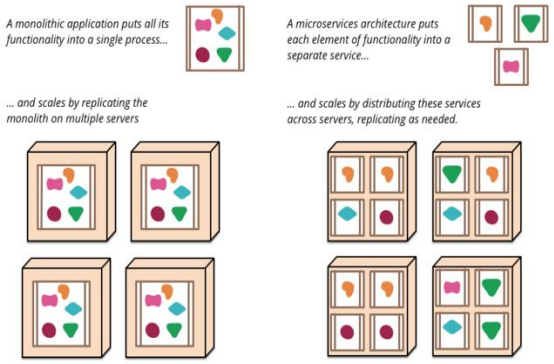

Fig. 2:Monolithics and Microservices

These frustrations have led to the microservice architectural style: building applications as suites of services. As well as the fact that services are independently deployable and scalable, each service also provides a firm module boundary, even allowing for different services to be written in different programming languages. They can also be managed by different teams .

4.1 Characteristics of a Microservice Architecture

We cannot say there is a formal definition of the microservices architectural style, but we can attempt to describe what we see as common characteristics for architectures that fit the label. As with any definition that outlines common characteristics, not all microservice architectures have all the characteristics, but we do expect that most microservice architectures exhibit most characteristics.

4.1.1 Componentization via Services

For as long as we've been involved in the software industry, there's been a desire to build systems by plugging together components, much in the way we see things are made in the physical world. During the last couple of decades we've seen considerable progress with large compendiums of common libraries that are part of most language platforms.

When talking about components we run into the difficult definition of what makes a component. Our definition is that a component is a unit of software that is independently replaceable and upgradeable.

Microservice architectures will use libraries, but their primary way of componentizing their own software is by breaking down into services. We define libraries as components that are linked into a program and called using in-memory function calls, while services are out-of-process components who communicate with a mechanism such as a web service request, or remote procedure call. (This is a different concept to that of a service object in many OO programs [3].)

One main reason for using services as components (rather than libraries) is that services are independently deployable. If you have an application [4] that consists of a multiple libraries in a single process, a change to any single component results in having to redeploy the entire application. But if that application is decomposed into multiple services, you can expect many single service changes to only require that service to be redeployed. That's not an absolute, some changes will change service interfaces resulting in some coordination, but the aim of a good microservice architecture is to minimize these through cohesive service boundaries and evolution mechanisms in the service contracts.

Another consequence of using services as components is a more explicit component interface. Most languages do not have a good mechanism for defining an explicit Published Interface. Often it's only documentation and discipline that prevents clients breaking a component's encapsulation, leading to overly-tight coupling between components. Services make it easier to avoid this by using explicit remote call mechanisms.

Using services like this does have downsides. Remote calls are more expensive than in-process calls, and thus remote APIs need to be coarser-grained, which is often more awkward to use. If you need to change the allocation of responsibilities between components, such movements of behavior are harder to do when you're crossing process boundaries.

At a first approximation, we can observe that services map to runtime processes, but that is only a first approximation. A service may consist of multiple processes that will always be developed and deployed together, such as an application process and a database that's only used by that service.

4.1.2 Organized around Business Capabilities

When looking to split a large application into parts, often management focuses on the technology layer, leading to UI teams, server-side logic teams, and database teams. When teams are separated along these lines, even simple changes can lead to a cross-team project taking time and budgetary approval. A smart team will optimise around this and plump for the lesser of two evils - just force the logic into whichever application they have access to. Logic everywhere in other words. This is an example of Conway's Law[5]in action.

Any organization that designs a system (defined broadly) will produce a design whose structure is a copy of the organization's communication structure.

-- Melvyn Conway, 1967

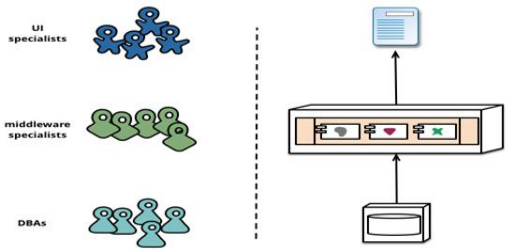

Fig. 3: Organized around Business Capabilities

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, persistant storage, and any external collaborations. Consequently the teams are cross-functional, including the full range of skills required for the development: user-experience, database, and project management.

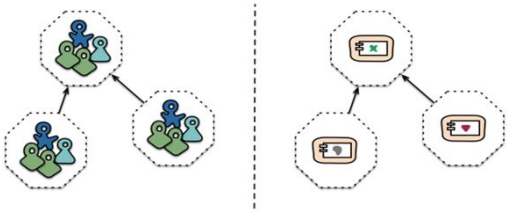

Fig.4:Service boundaries reinforced by team boundaries

Large monolithic applications can always be modularized around business capabilities too, although that's not the common case. Certainly we would urge a large team building a monolithic application to divide itself along business lines. The main issue we have seen here, is that they tend to be organised around too many contexts. If the monolith spans many of these modular boundaries it can be difficult for individual members of a team to fit them into their short-term memory. Additionally we see that the modular lines require a great deal of discipline to enforce. The necessarily more explicit separation required by service components makes it easier to keep the team boundaries clear.

4.1.3 Decentralized Data Management

Decentralization of data management presents in a number of different ways. At the most abstract level, it means that the conceptual model of the world will differ between systems. This is a common issue when integrating across a large enterprise, the sales view of a customer will differ from the support view. Some things that are called customers in the sales view may not appear at all in the support view. Those that do may have different attributes and (worse) common attributes with subtly different semantics. This issue is common between applications, but can also occur within applications, particular when that application is divided into separate components. A useful way of thinking about this is the Domain-Driven Design notion of Bounded Context. DDD divides a complex domain up into multiple bounded contexts and maps out the relationships between them. This process is useful for both monolithic and microservice architectures, but there is a natural correlation between service and context boundaries that helps clarify, and as we describe in the section on business capabilities, reinforce the separations.

As well as decentralizing decisions about conceptual models, microservices also decentralize data storage decisions. While monolithic applications prefer a single logical database for persistant data, enterprises often prefer a single database across a range of applications - many of these decisions driven through vendor's commercial models around licensing. Microservices prefer letting each service manage its own database, either different instances of the same database technology, or entirely different database systems - an approach called Polyglot Persistence. You can use polyglot persistence in a monolith, but it appears more frequently with microservices.

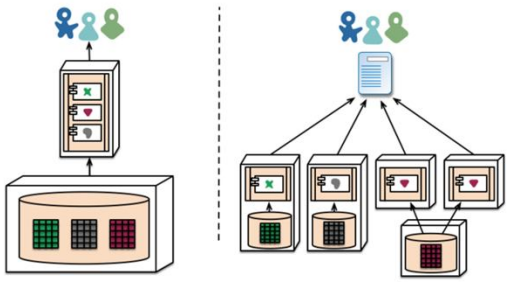

Fig.5: Decentralized Datamanagement

Decentralizing responsibility for data across microservices has implications for managing updates. The common approach to dealing with updates has been to use transactions to guarantee consistency when updating multiple resources. This approach is often used within monoliths.

Using transactions like this helps with consistency, but imposes significant temporal coupling, which is problematic across multiple services. Distributed transactions are notoriously difficult to implement and and as a consequence microservice architectures emphasize transactionless coordination between services, with explicit recognition that consistency may only be eventual consistency and problems are dealt with by compensating operations.

Choosing to manage inconsistencies in this way is a new challenge for many development teams, but it is one that often matches business practice. Often businesses handle a degree of inconsistency in order to respond quickly to demand, while having some kind of reversal process to deal with mistakes. The trade-off is worth it as long as the cost of fixing mistakes is less than the cost of lost business under greater consistency.

4.1.4 Infrastructure Automation

A monolithic application will be built, tested and pushed through these environments quite happlily. It turns out that once you have invested automating the path to production for a monolith, then deploying more applications doesn't seem so scary any more. Remember, one of the aims of CD is to make deployment boring, so whether its one or three applications, as long as its still boring it doesn't matter.

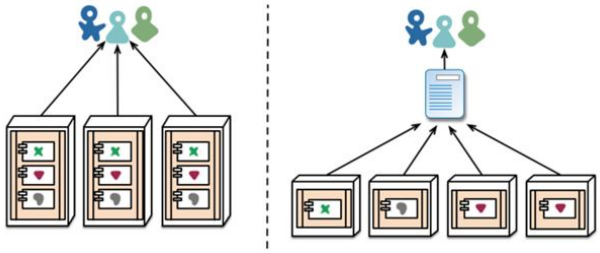

Fig. 6: Module deployment often differs

4.2 Internal layers of microservices :

Microservices can usually be split into similar kinds of modules. Often, microservices display similar internal structure consisting of some or all of the displayed layers.

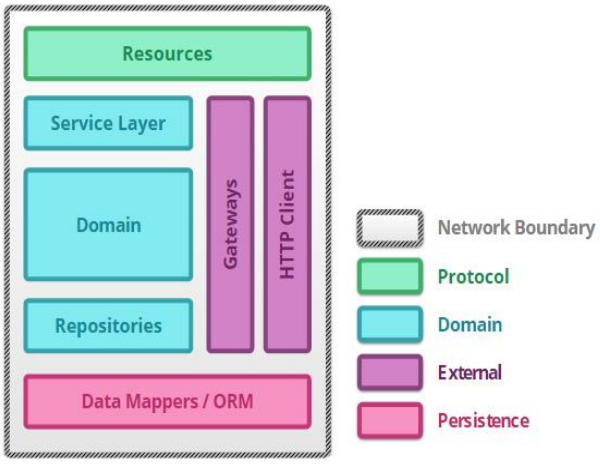

Fig.7: Internal layers of microservices

Resources act as mappers between the application protocol exposed by the service and messages to objects representing the domain. Typically, they are thin, with responsibility for sanity checking the request and providing a protocol specific response according to the outcome of the business transaction

Almost all of the service logic resides in a domain model representing the business domain. Of these objects, services coordinate across multiple domain activities, whilst repositories act on collections of domain entities and are often persistence backed.

If one service has another service as a collaborator, some logic is needed to communicate with the external service. A gateway encapsulates message passing with a remote service, marshalling requests and responses from and to domain objects. It will likely use a client that understands the underlying protocol to handle the request-response cycle.
Except in the most trivial cases or when a service acts as an aggregator across resources owned by other services, a micro-service will need to be able to persist objects from the domain between requests. Usually this is achieved using object relation mapping or more lightweight data mappers depending on the complexity of the persistence requirements.

Often, this logic is encapsulated in a set of dedicated objects utilised by repositories from the domain.

Microservices connect with each other over networks and make use of “external” datastores.

V. CONCLUSIONS

Breaking down your monolithic application into microservices also means breaking down your monitoring approach. For monolithic applications, traditional APM tools that provide code-level visibility are useful to understand performance bottlenecks inside the application.

For microservices applications, focus on understanding the performance of the individual microservices and the interactions between them, and use low overhead instrumentation techniques to gather application-specific metrics.

A Monolithic architecture only makes sense for simple, lightweight applications. You will end up in a world of pain if you use it for complex applications.

The Microservice architecture pattern is the better choice for complex, evolving applications despite the drawbacks and implementation challenges.

REFERENCES

[1] martinfowler.microservices (2016,November6) [Online]. Available: http://martinfowler.com/articles/microservices

[2] How-to-prepare-next-generation-cloud-applications , microservice .(2016,November 5). [Online] Available :http://www.computerweekly.com/feature/Microservices-How-to-prepare-next-generation-cloud-applications

[3] microservices .(2016,November 5). [Online]Available: https://en.wikipedia.org/wiki/Microservices

[4] Rodgers and Peter. "Service-Oriented Development on NetKernel- Patterns, Processes & Products to Reduce System Complexity". CloudComputingExpo. http://sys-contv.sys-con.com. Aug.19. 2015.

[5] Russell and Perry; Rodgers, Peter; Sellman, Royston (2004). "Architecture and Design of an XML Application Platform". HP Technical Reports. pp. 62. Retrieved 20 August 2015.

[6] Hitchens and Ron "Your Object Model Sucks" PragPub Magazine (Pragmatic Programmers):CA, Dec .2014, pp.15-25

[7] Swaine and Michael ed.. PragPub Magazine (Pragmatic Programmers):CA, Dec .2014, pp.129-140

[8] Namiot, Dmitry, and Manfred Sneps-Sneppe. "On micro-services architecture." International Journal of Open Information Technologies 2.9 (2014).