Evolution of Software Architecture

The concept of software development architecture originated in the 1980s. The advent of Object-Oriented programming from the 1980s and in particular in the 1990s, led to a significant contribution to the field of software development (Dragoni et al., 2017). Various architectural design patterns emerged explaining and evaluating the important and recurring object design in object-oriented systems (Gamma et al., 1995).

Over a period of time, distributed systems started gaining substantial importance for the benefits it offered with respect to geography, speed, resource sharing and fault tolerance (Ghosh, 2014). As we moved through the 2000s, a new architecture style emerged focusing on services in contrast to the application as a whole, which helped to harness the complexity of distributed systems and integrate different software applications. SOA is a "paradigm for organizing and utilizing distributed capabilities that may be under the control of different ownership domains" (MacKenzie et al., 2006). SOA offered benefits like dynamism, reusability, and modularity, distributed development and integration of heterogeneous and legacy systems. Microservices are the second iteration of the concept of SOA (Dragoni et al., 2017).

After SOA, microservice architecture was introduced in early 2005 by Peter Rodgers that offered improved simplicity over SOA. Today microservices architecture is very popular and widely adopted as demonstrated in surveys conducted by different open-source integration providers and technology companies (WSO2, CA technogies, Runscope, 2018, p. 2). Microservices manages complexity by making services completely independent in development and deployment. The unique feature of microservices over SOA is its capability to independently deploy the services (Newman, 2015).

Microservices architecture

Microservices architecture (MSA) is a modern architectural style for software development. In this architecture style, the small and autonomous services work together to achieve goals. [Newman] in his book defines microservices, as independent services with each service defined on the business boundaries, communicating with each other to achieve a specific goal. The fact that these services are independent entities brings many improvements and flexibility in testing, deployment, development and error handling [Micr00] which are otherwise difficult to achieve in the monolith style of architecture. This also enhances scalability and allows each microservice to be independently scaled. However, these benefits introduce some challenges as well.

In addition to these three characteristics, you can identify two more fundamental attributes of microservices:

Each microservice can be deployed independently. Without this, a microservice application would still be monolithic at the point of deployment.

A microservice is replaceable. Having a single capability places natural bounds on size; likewise, it makes the individual responsibility, or role, of a service easy to comprehend [Microservices in action].

**Why is microservice a good choice?**

Microservices architecture started with the goal to be able to deploy smaller parts of software independently without affecting the rest of the application. However, this has evolved and started to influence the way software is architected from the outset. Microservices therefore suit evolutionary design, where the business anticipates that certain functions may fail in the future. Business models that are scalable need applications that can be reconfigured and augmented as scenarios evolve. Since each microservice is a small business process, and because it represents a small aspect of business functionality, it is easy to replace or change the workflow. A web service based approach is more challenging in this respect, as the focus on object reuse means that changes can often affect many disparate parts of the application. Technical heterogeneity leads to microservice. Development friction increases as complex systems grow and Microservices reduce friction and risk.

**Challenges**

Having these granular independent services means that the execution of multiple services in specific orders can derive business value, and therefore, the way they interact has a pivotal role [InSi18]. In real-world scenarios, end-to-end business processes can be long-running, parallel, and also sometimes might require human intervention [InSi18]. This research focuses on this critical aspect of communication within the microservices and the technology used for the communication between the services.

Integration Technology

Getting integration right is the single most important aspect of the technology associated with microservices [Newman]. When well done microservices retain their autonomy. There are an array of integration technologies available such as SOAP, REST and RPC. Integration technologies need to provide security, recoverability, interruptibility or transactions while communicating with other services, API's or resources.

There are a few factors to keep in mind before choosing the integration technologies. According to [Martin Fowler], the way two services communicate with each other has to be simple, light weight and technology agnostic. Bearing this factors, the technology best suited for microservices architecture are -

1. **REST over HTTP**

Representational State Transfer (ReST) is an architectural style inspired by the Web. REST itself doesn’t really talk about underlying protocols, although it is most commonly used over HTTP. ReST can be used with other protocols but some of the features that HTTP gives as part of the specification, such as verbs, make implementing REST over HTTP easier, whereas with other protocols you’ll have to handle these features yourself.

HTTP itself defines some useful capabilities that play very well with the REST style. For example, the HTTP verbs (e.g., GET, POST, and PUT) already have well understood meanings in the HTTP specification as to how they should work with resources. The REST architectural style actually tells that methods should behave the same way on all resources, and the HTTP specification happens to define a bunch of methods we can use. GET retrieves a resource in an idempotent way, and POST creates a new resource. This means we can avoid a lot of different createCustomer or editCustomer methods. Instead, we can simply POST a customer representation to request that the server create a new resource, and initiate a GET request to retrieve a representation of a resource. Conceptually, there is one endpoint in the form of a Customer resource in these cases, and the operations we can carry out upon it are baked into the HTTP protocol. HTTP also brings a large ecosystem of supporting tools and technology. We get to use HTTP caching proxies like Varnish and load balancers like mod\_proxy, and many monitoring tools already have lots of support for HTTP out of the box. These building blocks allow us to handle large volumes of HTTP traffic and route them smartly, in a fairly transparent way. We also get to use all the available security controls with HTTP to secure our communications. From basic auth to client certs, the HTTP ecosystem gives us lots of tools to make the security process easier, and we’ll explore that topic more in Chapter 9. That said, to get these benefits, you have to use HTTP well. Use it badly, and it can be as insecure and hard to scale as any other technology out there. Use it right, though, and you get a lot of help. Note that HTTP can be used to implement RPC too. SOAP, for example, gets routed over HTTP, but unfortunately uses very little of the specification. Verbs are ignored, as are simple things like HTTP error codes. All too often, it seems, the existing standards and technology are ignored in favor of new standards that can only be implemented using brand-new technology—conveniently provided by the same companies that help design the new standards in the first place!

1. **Messaging Queue(MQ)**

The communication style that ReST uses is called the Request-Response model. There are also other ways of communication in which the communication is not synchronous unlike the Request-Response model. One of the ways to achieve the asynchronous communication is the Publisher subscriber model or the point to point model. Both the models use the Message Queue.

The message provides the ability to store the messages on the messaging platform.The messaging platform is also called as the Messaging Broker. Queue as the name suggests is the First In First Out(FIFO) data structure, which means that the first message sent to the queue will be retrieved first from the queue. The queues can be shared between different applications or each application can have a dedicated queue.  Number of attributes can be defined for a queue like the name of the queue, its size, algorithm to sort the messages. The different style of messaging communication includes Publish/subscribe messaging and Point-to-point messaging.

Like ReST messaging is a lightweight communication protocol. The other advantages it offers is the asynchronous communication and transaction capability.

1. **gRPC**

gRPC is a language-neutral, platform neutral, open source RPC mechanism initially developed at Google (Google, 2015b). It employs protocol buffers (Protobuf) (Google, 2015a), which is Google’s mature open source mechanism, for structured data serialization. gRPC aims to minimize data transfer overhead while providing several calling patterns. [A Lightweight and High Performance Remote Procedure Call Framework for Cross Platform Communication](https://www.scitepress.org/papers/2016/59312/59312.pdf)]

gRPC is also a framework for cross platform remote procedure calls just like Thrift. In gRPC Protocol Buffers are used both as a serialisation format for messages but also as an interface definition language that makes the cross platforms calls feasible [[Efficient communication with microservices](http://www8.cs.umu.se/education/examina/Rapporter/PetterJohansson2017.pdf)]

Programming language used for the development of microservices

Java is both a programming language and platform. Java language is a high-level object-oriented programming language. Java platform is an environment on which Java programming language applications run (Differences between Java EE and Java SE - Your First Cup: An Introduction to the Java EE Platform, n.d.). There are four different platforms of Java programming language and all the platforms contain Java Virtual Machine and an application programming interface (API).

1.      Java Platform, Standard Edition (Java SE),

2.      Java Platform, Enterprise Edition (Java EE)

3.      Java Platform, Micro Edition (Java ME)

4.      JavaFX

According to Oracle, microservices can be built using Java EE(Enterprise Edition) (Krill, 2018). Java Platform, Enterprise Edition is the standard in the community driven enterprise software (<https://www.oracle.com/java/technologies/java-ee-glance.html>). All the programs for comparison will be built on and run on the latest available version of Java EE as of today which is Java EE 8.

Prerequisites for Java -

1. Java Platform (Java 8) runtime
2. Java EE 8 libraries (1.8 JDK)
3. Glassfish server 4.0

**Ballerina**

Like Java, Ballerina is a language and a platform. However, unlike Java, Ballerina is not an object-oriented programming language. It is a traditional declarative language like C, as most things that pass over the network are not objects. Ballerina also supports object oriented style of coding. As Ballerina was designed specially to solve the integration related issues, WSO2 designed this language as an API first programming language.  Ballerina platform runs on java and the runtime for the ballerina application is called jBallerina. jBallerina internally transforms the Ballerina sources to Java bytecode and run on a JVM<https://ballerina.io/learn/installing-ballerina/#installing-via-the-ballerina-language-zip-file> . Ballerina requires Java version 1.8 or above to run the ballerina programs. All the programs in Ballerina will be build using the version 1.2.6 which is the latest stable version of ballerina (<https://ballerina.io/downloads/release-notes/>).

Prerequisites for Ballerina -

1. Java 8 runtime or above
2. Ballerina 1.2.6

**Jolie**

Jolie is the first microservice-oriented programming language. It is Java Orchestration Language Interpreter Engine (JOLIE). Jolie is a service-oriented programming language with a syntax like C language. Unlike Java and Ballerina, Jolie does not have its own platform. It requires Java to run. Latest version of Jolie runs on Java 8 or later versions of Java. As of today, the latest version of Jolie is 1.8.<https://www.jolie-lang.org/downloads.html>.

Prerequisites for Jolie-

1. Java 8 runtime or above
2. Jolie 1.8

Montesi, Guidi, and  Zavattaro (2014) believe that microservice designers have to deal effectively with both behavioural and architecture aspects of the microservices architecture. These technical problems arise due to the fragmentation of monolith architecture and during integration. Ideally, a language should exist specially created for designers, which lets them focus on single concepts instead of the different tools and technology (Montesi et al., 2014). This would make it easy for designers to use and reduce the complexity and maintenance of the application.

Jolie and Ballerina are full-fledged programming languages designed for microservices architecture (Newman, 2015). Jolie enables the user to translate the graphical design into code, and Ballerina has graphical and textual support for programming, this helps different stakeholders with different technical backgrounds to work and collaborate more effectively (Weerawarana et al., 2018).

Unlike Ballerina and Jolie, other programming languages lack native constructs for workflows thus adding the burden of middleware, ESB and API gateways that help to scale the application, reduce its complexity with a growing number of services (Weerawarana et al., 2018).  Additionally, Ballerina and Jolie, unlike mainstream languages, support visual programming and also sequence diagrams to reduce the complexity that arises due to the fragmentation. Furthermore, they are also capable of providing features like reliable delivery of messages and interruptibility, which the mainstream languages fail to offer (Weerawarana et al., 2018).

This research uses the experiment design approach to answer the research question. According to Jackson (2014), the experiment design approach “allows a researcher not only to describe and predict but also to establish a cause and effect relationship through manipulation of a variable and control of the situation.” The main foundation of the experimental approach is that the researcher gains the control of the environment as much as possible. This is to identify if there is a cause and effect relationship between the variables being studied. In the experimental process, certain factors are selected and deliberately varied in a controlled manner to understand their effects on the response of interest (Wang & Wan, 2009).  These variables are called independent variables. The variables that are measured are the dependent variables. These variables are derived from the critical factors for a programming language and integration and these variables will be used to measure the programming language in my research.