The Evolution of Java Based Software Architectures

Sebastian Łaskawiec*

Received: 16 November 2015; Published online: 25 June 2016

© The author(s) 2016. Published with open access at www.uscip.us

Abstract

This paper analyses the evolution of Java based Software Architectures starting from 1980’s till 2015. Each architecture model was described in the context of Development Model, Deployment model and Maintenance Model with proper background information.

Keywords: Software Architecture; Layered Architecture; Service Oriented Architecture; Event Driven Architecture; Microservices; Reactive Architecture; Cloud, Containers

1. Introduction

Software architecture has been treated as a research topic since 1980’s The Golden Age of Software Architecture: A Comprehensive Survey. At first, the researchers concentrated on the physical structure of large scale systems but nowadays Software Architecture it is something more – it is a guidance for designing large, complex systems starting from physical connections between components, going through the development concerns and addressing issues in deployment and production maintenance. The amount of work related to Software Architecture is often concentrated on where the system complexity is located. In 1980’s it was the physical layer but nowadays, with Visualization on-demand and Cloud Computing, the application layer needs to be designed properly to use this advantage. Over the years hardware standardization efforts and dynamic growth of Java ecosystem significantly influenced this transition by providing a runtime platform as necessary libraries and which were able to utilize underlying Operation System.

Each architecture approach is presented with background information and main practical challenges of that time. This explains main motivation and puts Software Architecture approach into proper context. A formal architecture description consists from three perspectives. The first one is the Development Model, which describes how Development Teams create Software. The second one is the Application Deployment perspective, which describes how system components

*Corresponding e-mail: sebastian.laskawiec@gmail.com

1 Institute of Telecommunications University of Technology and Life Sciences in Bydgoszcz, Poland
can be deployed. The last one is the Application Maintenance and Scaling perspective which often overlaps with the Application Deployment model and shows how the system can be maintained in the production environment. The last part describes the biggest advantages and challenges of the given approach.

The value added of this paper is an analysis of each Architectural approach, it’s challenges and the way they were solved forwarded with a conclusion about future architectural challenges and directions.

2. Software Architecture prior to Java

Java emerged in 1995 Java History and was widely adopted by C/C++ developers because of syntactic similarities of those two languages. An evolution of the architecture prior to Java sets later Architectural approaches in the proper context.

2.1 Layered Architectures

In 1960’s along with the invention of the first data network Business Data Communications and Networking, Eleventh Edition, a first application was invented. It consisted of a mainframe server and a terminal. This approach is often called 1-Tier Application (Fig. 1).

![Fig. 1. 1-Tier Application Architecture](image)

The biggest advantage of this architecture is simplicity - both application logic and all necessary data reside on the same server. Since most of the software libraries were statically linked, every new software release had to be redeployed (or re-installed) to the server. Scaling with this approach can be done only in one dimension – vertical (adding more hardware resources to the server).

In 1970s, with increasing popularity of network equipment, many companies were facing integration problems between different hardware vendors. The standardization efforts resulting in creating new layer in System Architecture called Middleware, which exposed standard API from hardware to software developers.

In the 1980’s standardized network connections and increasing popularity of Personal Computers Options for departament networking was the main factor for moving processing towards to the clients (Fig. 2).
This shift changed also the programming model from server based to client based. However because creating mainframe software was complex and time-consuming many companies had planned backlog for several years and this architectural shift was problematic for them. With standardized APIs and centralized storage application Development was much simpler and took less time. Since the application was deployed on client computers, software distribution started to be problematic but on the other hand system scaling was improved.

In 1995 Wayne W. Eckerson published an article Three Tier Client/Server Architecture: Achieving Scalability, Performance, and Efficiency in Client Server Applications about 3-tier architecture, which separates presentation layer from application logic and accessing the data (Fig. 3). This change was needed from maintenance point of view because application logic and data access logic layer requires different skill set from people who maintain them.

A Three Tier application has been widely used since 1995 and it often reflects communication structure in companies (application developers, database administrators and UX designers). Clearly defined separation of concerns allows applying different scaling strategies to all layers and improves maintainability. The Development model was also changed and separate groups of developers was created (backend and frontend developers).

2.1 The Internet applications and raise of Java

In 1989 Timothy John introduced term “World Wide Web” and successfully transferred information
between client and server using Hypertext Transfer Protocol Weaving the Web: The Original Design and Ultimate Destiny of the World Wide Web by its inventor. It is estimated that 6 years later the Internet had more than 20 million users The Internet Index and in 1999 the number increased to 147 million The Internet Index. Increasing number of active users added additional challenges to a standard 3-tier systems such as serving data for many users at the same time, security or system administration.

In 1995 Sun introduced the Java programming language which was also introduced to Netscape Navigator Java History. Concentrating on the Web resulted in starting Java Professional Edition project, which was transformed into Java 2 Enterprise Edition and released in 1999 JavaTM 2 Enterprise Edition Developer's Guide (v. 1.2.1). Java itself addressed many fundamental concerns out of the box by providing multithreading, sandboxing application code using Applets or provides monitoring by Java Management Extensions.

3. Monolithic Applications

In 1998 Sun released a product called Java Professional Edition which was targeted to the enterprise customers. It's main purpose was to use the advantage of server programming model Professional Java for Web Applications which fits exactly into 3-Tier Application Architecture. A year later Sun released first J2EE specification which was the starting point of an Application Server era in software architecture JavaTM 2 Enterprise Edition Developer's Guide (v. 1.2.1). At the moment of writing this article JEE 6th edition is a standard widely used in the production environment and JEE 7 will have production support this year. During this time a lot of changes and improvements were introduced in infrastructure layer as well as in programming model.

Application servers are designed to handle more than one deployment at a time by isolating Class Loaders one from another (Fig. 4).

![Fig. 4. Many deployed application on one Application Server](image-url)

In parallel to the Application Servers development, companies like VMware started working on system virtualization Virtualization system including a virtual machine monitor for a computer with a segmented architectur which resulted in creating the first open-source x86 hypervisor - Xen
in 2003 Xen Virtualization. Operation System virtualization allowed creating Application Server clusters very easily. Such Application Server clusters are often called “domains” and frequently many Application Server are grouped together to perform certain function like batch processing or serving web resources (Fig. 5).

Typically, a server cluster is maintained by a team responsible for infrastructure and is tuned for maximum hardware utilization. This approach separates application development from production maintenance and deployment and makes whole organization less Agile.

JEE standard allowed developers to concentrate more on structuring application internals rather than implementing customized integration layer. There are 2 main streams of Software Development model - Layered and Hexagonal Architecture. A typical layered architecture Software Architecture in Practice, Third Edition (Fig. 6) consists of a Data Access Layer (often called DAO), Business Logic Layer and Presentation Layer.
This architecture pattern is very easy to learn by developers because they focus on technical side of the application. However, it often shadows business goals of the application and introduces a lot of technical details into Presentation Layer, making the application less intuitive for Business Partners. This problem is partially solved by applying Hexagonal architecture (often called Onion model) Hexagonal Architecture proposed by Alistair Cockburn in 2012, which was inspired by
Domain Driven Design Domain-Driven Design: Tackling Complexity in the Heart of Software. It’s main goal is to create a Domain which is focused on solving business problems and dedicating an adapter for each client (Fig. 7).

Applying Hexagonal architecture requires focusing on application's business purpose rather than technical structures and layering whereas additional adapters act as a facades to the system making it harder to understand and debug.

4. Pluggable Architecture

One of the key features which distinguishes Java from C/C++ is Dynamic Class Loading. Lack of static linking made upgrading libraries much easier possible even in runtime. In 2000 an initial version of OSGi (The Open Service Gateway Services Initiative) was released and established a framework for dynamic application deployment for small-memory devices OSGi Service Gateway Specification Release 1.0. First releases of the specification were targeted for embedded home devices like home routers or STB boxes but 3 years later, in 2003, a new project was created which was addressing Eclipse editor runtime related issues - Equinox. Since then OSGi platform has been used as a runtime environment for many code editors (like Eclipse or Netbeans) as well as by many application servers (for example Glassfish).

Enterprise applications use pluggable architecture at many different levels. One of the oldest mechanisms used is JNDI (Java Naming and Directory Interface Specification) which was introduced in JEE 1.2. JNDI allowing to register an object instance by one application and consume it by another (Fig. 8).

![Fig. 8. JNDI bean registration](image)

Very similar concept is used by the OSGi service registry OSGi Service Platform Core Specification Release 4.1 with some additional features which concentrate on component lifecycle. Controlling component’s lifecycle and dynamic nature is one of the most challenging tasks for Pluggable Architectures. On the other hand there are some use cases (for example Code editors with plugins) where this kind of solution fits perfectly.
5. Service Oriented Architectures

Monolithic systems grow rapidly when adding new features. Bigger code base makes systems more difficult to understand and developing new features - more time-consuming. In 1996, a first report about SOA was published Service Oriented Architectures, Part however it became widely used with popularization of Web Services. Although Web Services are not essential for SOA architecture, there is a relationship between those two technologies Service-Oriented Architecture Scenario.

A Service Oriented Architecture was built on two major technical fundamentals – Services and Enterprise Service Bus (Fig. 9). Those two components are essential to achieve loose coupling between components.

![Fig. 9. SOA system architecture](image)

Services are considered as course grained, self-contained, stateless components implementing business functionality with a well defined standard interface (often achieved by using Web Services). They are often classified as Data Services or Logic Services depending on their role in the system SOA in Practice.

In order to ensure communication between services there is an Enterprise Service Bus in the center of SOA architecture which acts a communication, mediation and translation layer between services. ESB is also responsible for intelligent message routing and separating physically Services. Many vendors add additional advanced features like load balancing or service name registration.

An architecture which consists only of Services and ESB is often called Fundamental SOA. Advanced SOA architectures use additional Orchestration and are often called Federated [3] or Network SOA. Enterprise SOA: Service-Oriented Architecture Best Practices. Orchestration layer is called Composed Services because they use basic functions and Services to perform more sophisticated business logic.

Some SOA systems also require integrating with Business Processes. This is the highest possible abstraction layer in SOA systems and is called Process-enabled SOA (Fig. 10)
From a user perspective SOA systems can be viewed as a Composite or Flow Systems. The former are mostly synchronous in nature. The end user sends a request to the system and getting back a response synchronously. Often some asynchronous processing takes place but it is not visible to the end user. The latter systems are asynchronous by its nature. The user can be notified when processing is finished but it will usually happen after some period of time Event-Driven Architecture Overview.

SOA architectures are aimed to decouple business functionality into smaller, easier to maintain pieces. Most ESB vendors add additional tools to make Web Services communication even easier and allow implementing features without writing the code. A centralized messaging system is also one of the weaknesses of SOA architectures. It may be considered as a single point of failure. Additionally today’s ESB software allows to embed business logic into message routing process. This might be treated as a potential danger for large systems, because business logic should be implemented either in Composed Services or in Processes and not to message routing.

5.1 Event Driven Architecture

In 2003 Roy Shulte published a paper which introduced Event Driven Architecture targeted for event based business processes. EDA and SOA have many similar concepts like modularity, encapsulation and clear interface boundaries but the communication model between Services is different [1].

Coordinating multiple services in order to achieve common goal might be realized using two different strategies - choreography or orchestration. W3C’s Web Services Choreography Working Group defines choreography as performing tasks by multiple independent agent to achieve common goal. An orchestration is a controlled sequence of Services invocations to achieve common goal. This emphasizes the difference - in orchestration there is a controlling entity whereas in choreography each Services performs actions independently [2].
An Event is the most important entity in Event Driven Architecture and it describes a notable thing that happened inside or outside of the system. An event can be also considered on purely technical level - it is a message sent from an Event Producer to an Event Listener using a message system. Event Listeners together with Event Processors need to filter out irrelevant or invalid data from message stream. Finally, Event Processors pass valid events into Event Reactors which are responsible for applying relevant actions based on event [3] (Fig. 11).

EDA messaging system architecture might be explicit or implicit. The former assumes there is a dedicated connection from Event Producer To Listener whereas the latter assumes no such connection.

Event Driven Architecture is an evolution of SOA where granularity is based on a type of Event, it is oriented on asynchronous processing using publish-subscribe model and Services are more loosely coupled than in SOA. Another important distinction is that Composed Services are built using choreography model rather than orchestration. Many ESB vendors offer EDA building blocks like JMS broker implementation along with traditional message routing components which allows to easily embedding EDA parts into existing SOA ecosystem. Since EDA is focused on asynchronous processing, this kind of architectures does not fit well into traditional web application which needs synchronous response to the client. One of the biggest problems in EDA architectures is maintaining data consistency, because each Service maintains its state based on event, which for some reasons might be delivered out of order or with significant delays.

6. Distributed Systems

Today Application Architectures have more challenges than they have had ever before. Peta-byte storage, thousands active users, hundreds of thousands transactions per hour and Time To Market less than a month. However there are tools which makes all this possible.

6.1 Infrastructure as a Service

Adjusting hardware resources for an application has always been one of the most challenging
things for the IT. Scaling out applications requires purchasing additional hardware and in large organizations it usually takes months. Fortunately in 2003 an open source hypervisor Xen was founded Xen Virtualization and it standardized an approach for Operating System virtualization. 3 years later Amazon introduced Elastic Compute Cloud Amazon Elastic Compute Cloud Beta announcement and started new era in software deployment - Infrastructure as a Service (IaaS), charging customers only for utilized resources and offering out of the box scaling. Many companies like Netflix decided to move from a standard Datacenter approach to a Cloud hosting solution. A few years later in 2010 another open source project made IaaS solutions even easier to adopt by the Enterprise - Openstack Openstack Austin Release Notes. New project allowed easily installing and maintaining Cloud solution within an enterprise Datacenter.

One of the most popular cloud offering is Public Cloud hosting, where whole infrastructure is provided and maintained by Cloud Provider and all servers are publically available. This is often the most cost effective strategy for customers. The second type of offering is Private Cloud where customer can host resources in his own Data Center (on-premise) or the resources might be hosted by Cloud Provider (off-premise). Servers in Private Cloud are usually accessible from dedicated subnetwork or via a secure connection. Often the Enterprise needs a dedicated hybrid Cloud which to some extends combines the above two approaches together The Enterprise Cloud.

Although Cloud itself didn’t change much in the application development model, a deployment model was significantly simplified. Out of the box monitoring and alerting solutions, which are typically built-in Cloud Offering allowed to run software on production by Development Teams. Infrastructure and Operation Teams can be reduced in numbers, or even in case of a public cloud, completely removed. In case of a larger organization’s Development and Operational Teams are often merged together into a DevOps Team. Such cross functional team can support full application lifecycle (from design to maintenance phase). Cross cutting concerns like central logging system or failure detection are sometimes solved by the Cloud Provider, but often need to be addressed by developers.

6.2 Containers

Although system virtualization pushed Software Architecture to another level, it came with the price of additional physical resources needed to support guest Operation Systems. Agile teams who maintained production software were very often also responsible for Virtual Machine’s Operation System which wasn’t very effective from operational point of view. In 2008 the Linux Containers Project (LXC) was founded which incorporated concepts like Chroots and CGroups Linux Containers (LXC) together allowing lightweight Operation System virtualization. Lightweight and full Operation System virtualization concepts are similar to some point but a full Operating System Virtualization might be considered as vertical resources partitioning whereas Linux Containers are orthogonal to this concept - they offer multiple applications within a single Virtual Machine (Fig. 12).
In 2013 Docker Project was founded and added additional layer of maintainability to the Linux Containers, Docker: Up & Running. One of the most important concepts is layering container images one onto another. The basic Operation System image is usually provided by its vendor but is designed to be customized. In some companies there are specialized image version dedicated per execution environment (UAT, Staging, Production) with different settings. Image recipes (called Dockerfiles) might be easily stored in Source Code Management system and built images might be pushed into the Docker Repository. This concept is inspired by Java ecosystem where every library is built from the source and exported to a Maven Repository.

At the moment of writing this article Docker creators do not advise using it into the Production environment considering it as not matured enough. However many companies ignores this and have successfully used this solution in Production. Docker naturally fits into the Cloud environment, moreover, many Cloud Providers, such as Red Hat, use Docker as a part of their Cloud Infrastructure Openstack Austin Release Notes. Since the solution is still very young there are many issues which still require a lot of research like inter-container communication or container lifecycle management.

6.3 Microservices Architecture

One of the IT’s biggest challenges is providing an application architecture which allows introducing new features quickly to the market and allows scaling it out easily at the time. There are many
initiatives in the enterprise world which try to address these concerns at different levels. Infrastructure as a Service and tools like Puppet or Chef address application scalability and configuration management problem. From Project Management perspective, Agile methods help building teams focused on Products rather than Projects. However from Application Architecture point of view, SOA, which has been widely used is often too complicated or simply does not fit well into companies business model. In this circumstance, Microservices Architecture naturally emerged to address those concerns. The first publications appeared in 2014 and in the same year Netflix presented their approach for software architecture Adopting Microservices at Netflix: Lessons for team and process design.

Microservices architecture is mainly inspired by SOA Services and Single Responsibility Principle. Each Service is focused on single Business function, is self-contained, fine-grained and independent application connected to other services using lightweight mechanisms (like REST interfaces or Message Queues) (Fig 13). This approach intentionally leads to decentralized data management. Often Microservices are created and maintained in the production by a single team which shows also strong DevOps influence.

There are many different strategies for scaling out applications The Art of Scalability: Scalable Web Architecture, Processes, and Organizations for the Modern Enterprise, Second Edition (Fig. 12). Scaling by cloning (Fig. 14 - X axis) can be interpreted by creating multiple Application Servers and sharing load among them using a Load Balancer. Decomposing into a set of separate services represents SOA-like approach (Fig. 14 - Y axis). Microservices introduced third dimension - data partitioning. It’s allowed to clone individual Services which are responsible for the same set of data (Fig. 14 - Z axis).
Despite the scaling out aspect, the Microservices architecture also decreases complexity of individual services and makes them independent from each other. This allows each Service to have its own deployment and release cycle and this, in turn, lowers Time to Market. However large number of connections between components often results in high system complexity and increased network latency. Components can no longer be tested in isolation and final system behavior is known after the production deployment. Finally, transaction management in Microservices is extremely complicated and many systems favor compensation strategy over transactions.

6.4 Reactive Architecture

Microservices architecture gives a lot of freedom in defining how Services communicate with each other and how they process the data. Spreading the system among many containers also requires different approach to failures, which in Microservices world are obvious. Virtualization on demand allows system to dynamically scale out but in order to reduce operations costs applications need to scale in when additional resources are no longer needed. Many organizations who are new to the Microservices Architecture need more concrete guidance to work with.

In order to address those issues in 2014 a Reactive Manifesto was created and signed by lots of people Reactive Manifesto. According to the manifest today's applications should be loosely coupled, scalable, easy to develop and fault-tolerant. This can be achieved by making system Responsive - it responds and detects errors quickly, Resilient - it responds even when facing failures, Elastic - can scale out and in dynamically, Message Driven - based on messages sent through lightweight messaging systems. Reactive Architecture Manifest is heavily inspired by Event Driven Architecture with few differences which are able to leverage current tools and technologies. Along with Reactive Architecture a Reactive Stream specification is developed providing an intuitive tool for inter Services communication and processing. Together with Actor Model
programming (tools like Akka) they provide solid fundamental for implementing system based on Reactive Architecture.

Reactive Architecture is still very young (Google Trends shows increasing popularity starting from 2014) and needs a lot of research. It’s mainly targeted for asynchronous systems and together with Reactive Streams provides good guidance for creating new systems. At the moment of writing this article there are few resources about adaptation of Reactive Architecture and its bottlenecks.

7. Conclusion

Software Architecture is still a very young discipline and requires a lot of research. In the past new ideas were carefully evaluated by scientists and proper research papers were published. Today, thanks to many technical conferences and tools like Twitter, new ideas are spread among many people very quickly. Modern programming languages based on JVM platform allow to prototype and evaluate new Architecture ideas very quickly without doing a long-term scientific research. Companies are keen on taking the risk of evaluating new technologies and approaches because if the evaluation is successful, it may give them an advantage over competitors. This trend shows that new Software Architecture ideas need to solve practical problems that many Developers and System Engineers are facing.

Melvin Conway’s Law says that system design always reflects organization communication structure. This explains why in the past Layered and Monolithic architectures were so popular. Hardware was very expensive at the time and having a separate maintenance team who tuned its performance had a lot of sense. On the other hand every team member was concentrated on a very narrow part of the system and the context of a change as well as business justification was often lost. Today new software features need to be delivered faster and the system as a whole need to be more stable. In order to fulfill those requirements all companies departments need to cooperate closely together. New software developments methods were invented to meet this criteria including Agile methods and DevOps (cooperation between Development and Operation teams). New Software Architectures like Microservices or Reactive Architecture reflect new communication model in the companies. Today, teams are cross-functional, flat-structured and focused on the business purpose rather than technical domain. Solving every day production problems by the Development Team makes new Architectures fragmented into smaller pieces, so that a small team could maintain it. Finally, thanks to the Cloud infrastructure each each small component can be released separately. However there are some additional cross cutting concerns which need to be kept in mind like application monitoring, logging or application scaling (X axis scaling). Those concerns might be addressed by Linux Containers and Docker by creating a standardized container properly installed and configured software in it. According to Forrester Report about containers Red Hat Containers TLP: Storyline & Data Review, this is the direction that many organizations would like to take in the near future.

From Development perspective new Architecture propositions like Microservices or Reactive Architecture are mostly influenced by Service Oriented Architecture and Event Driven Architecture. The biggest changes were introduced to communication layer. Current architecture approaches favor lightweight protocols and messaging systems over enterprise solutions like ESB. Each service might be compared to a small Monolithic application because it has similar problems, like creating
proper Domain Model and adapter with Hexagonal approach, or how to create proper layers using Layered Architecture. Finally, aligning smaller Services to the business goals rather than technical structure makes a lot of sense from Business point of view. After all, the product created by Engineers need to be sold to the customers.

It is very likely that future Software Architectures will be specialized to solve only a subset of problems and a large enterprise system will require a hybrid approach many smaller subsystems embedded into it. Despite the hybrid approach, some sort of standardization will have to be done especially in Containers and Docker area allowing communicating Software and underlying Infrastructure.

References

DUR001: Alexandra Durcikova, Alan Dennis, Jerry Fitzgerald, Business Data Communications and Networking, Eleventh Edition, 2011,
COC001: Alistair Cockburn, Hexagonal Architecture, http://alistair.cockburn.us/Hexagonal+architecture
FOR001: Conor Donegan, David Cannon, Dave Bartoletti, Kurt Bittner, Red Hat Containers TLP: Storyline & Data Review, 2015,
EVA001: Eric Evans, Domain-Driven Design: Tackling Complexity in the Heart of Software, 2003,
BON001: James Bond, The Enterprise Cloud, 2015,
MAT001: Karl Matthias, Sean P. Kane, Docker: Up & Running, 2015,
MIC001: Netflix, Adopting Microservices at Netflix: Lessons for team and process design, 2014
WIL001: Nicholas S. Williams, Professional Java for Web Applications, 2014,
JOS001: Nicolai M. Josuttis, SOA in Practice, 2007,
SEY001: Patricia Seybold Group, Event-Driven Architecture Overview, 2006,
CHA001: Prabhakar Chagan, Xen Virtualization, 2007,
OSG001: The Open Services Gateway Initiative, OSGi Service Gateway Specification Release 1.0, 2000,
OSG002: The OSGi Alliance, OSGi Service Platform Core Specification Release 4.1, 2007,
BER001: Tim Berners-Lee, Weaving the Web: The Original Design and Ultimate Destiny of the World Wide Web by its inventor, 2000,
ROY001: W. Roy Schulte, Yefim V. Natis, Service Oriented Architectures, Part, 1996,
NAT001: Yefim V. Natis, Service-Oriented Architecture Scenario, 2003,
MIC002: Google Trend: Microservices Architecture,
  https://www.google.com/trends/explore?q=%22microservice+architecture%22
LXC001: Linux Containers (LXC), https://linuxcontainers.org/lxc/
WMV001: Virtualization system including a virtual machine monitor for a computer with a segmented
  architectur,
ORA002: JavaTM 2 Enterprise Edition Developer’s Guide (v. 1.2.1),
  http://docs.oracle.com/javaee/1.2.1/devguide/html/
NET001: Options for department networking, 1986,
WIK001: Society for Worldwide Interbank Financial Telecommunication,