International Journal of Computer Trends and Technology
ISSN: 2231–2803 / https://doi.org/10.14445/22312803/IJCTT-V72I3P106

Volume 72 Issue 3, 37-45, March 2024

Original Article

System Integration, From Middleware to APIs

Prasenjit Banerjee

Success Architect Director at Salesforce, USA.

Corresponding Author : prasenjit.banerjee@salesforce.com

Received: 15 January 2024 Revised: 26 February 2024 Accepted: 15 March 2024 Published: 28 March 2024

Abstract *- Nowadays any customer journey, whether it is banking or financial services, retail or healthcare, goes through several software systems. Transactions as simple as using a credit card to make a purchase or sending an offer to the customer for a personal loan often involve the orchestration of a myriad of systems. While some systems are deployed in the cloud, others are legacy on-premises systems that maintain customer transactional history. Connecting these systems to make sure that they exchange data and logic in a way that makes the overall process smooth and meaningful can be achieved in a lot of ways. The technology behind System Integration has evolved over time and has taken a paradigm shift from the way it was done in the past to the way it is approached now. For a lack of standardization, in the field of system integration, many organizations have taken radically different approaches towards it and such experiments have led to a diverse range of outcomes. It is possible to look at those different outcomes and objectively assess which approach is better, more robust and scalable than the other. In this study, I have tried to draw several examples from my 16 years of experience to make an objective evaluation of how each of those approaches compare against each other. What are the common challenges faced during System Integration, and what are the broad common patterns that have evolved as best practices that the industry has embraced.*

Keywords *- Middleware, API, Client-Server, MVC, Enterprise Service Bus.*

1. Introduction

The Term middleware was first used at the NATO Science Committee conference in Garmisch[1]. At that time, the term was used to define a computer system that needs to access information from another computer's operating system. The concept did not solicit much attention for the next two decades. In the early 1990s, In the time when Mainframe computers were widely prevalent in business applications, SABRE referred to the term Middleware while pointing to the decoupling of TP monitors from the CICS systems[2]. Until the use of commercial systems for distributed computing, the term Middleware was only a concept that did not find much practical usage. While mainframe computers used terminals and huge servers, they never used the concept of Middleware as we understand and know it today. With the commercialization of distributed computing systems, a wide gamut of different computer systems architecture and technology evolved. Each of these systems helped the industry solve a specific problem. For example, let's take a simple example of opening a bank account.

It starts right from customers being targeted by the bank with a plethora of offers. These offers are derived based on the customer's demographics, purchase history, etc. After receiving the offer from various neighborhood banks to open bank accounts, let's say the customer chooses to apply to one of them, possibly because their offer was the most lucrative for the customer given her situation. The customer can apply for this account online using her home internet connection. The bank acknowledges the offer based on the offer reference number and takes all the necessary details from the customer to open the account. Then, after identity and income verification, the customer is provided with the account details and a pre-printed debit card.

Let's now understand how this use case is accomplished with the systems information technology systems available to the bank. The bank gets the prospect's information, which it uses to target the customer. When the customer accepts the offer, The bank uses a website that welcomes the customers to apply with all the demographic, identity and income details for the bank account creation. The bank then verifies the identity of the customer. The bank then allows or denies the customer based on the income and identity verification results. Let's say the bank allows the prospect, given that the information was accurate and create a profile for the customer, a bank account and sends a debit card to his/her home address. Let's stop here for the purpose of simplicity and understand what systems were at play here.

The bank is probably using a third-party *Ad tech aggregator system* to get customer information, which it is using to target the customer. This Ad Tech Firm is scuffing the data from a host of different systems, which includes the internet, customers' browsing history, customers' geo-location

tracking, if enabled, etc. Next the bank is using a *Marketing System* to send a blast of emails to all the prospects who may possibly convert to a customer. Thank Bank is using a web application to build a website that allows the customer to apply for an account. Then the bank is leveraging some government agency's identity and income verification system to verify the identity of the customer. The bank is creating the customer's profile in the Customer Relationship Management system (CRM)[3].

The bank is using a Core banking system[4] to open the bank account and generate the account number. It uses a statements and *letters dispatch system* to send regulatory information to the customers, including statements and notices. So, with this very straightforward example of the customer's journey, the customer's information is travelling through 9 distributed systems. Now previously these could have been simplified and processed by a single monolithic application. But a monolithic[5] application is crippled in the sense that it does not have very many ways to extract data from other systems only if they are not built leveraging the same technology stack or, as was the case, from the same specific vendor.

2. Monolithic Systems Vs Distributed Systems

The evolution of computer systems for large commercial use started with the Mainframe systems. Although several companies started creating these systems, it was IBM who monopolized the business with their research and contribution in advancing the technology. With the commercialization of the Windows operating system and the spread of the Unix/Linux operating system[6], PCs came into use in the early 1990s and transformed the landscape. The Distributed software systems were built on commodity PC hardware; they used limited computer and memory and were able to build software systems using programming languages such as C, C++, Java[7], etc. that catered to various affordable business needs. This democratization of PCs led to the rise of distributed computing. While distributed software systems catered to the needs of many different businesses and essentially created a more intuitive and user-friendly experience that is not just limited to computer scientists, it created an enormous need to connect these systems for the exchange of data that could elevate the experiences of the users in many ways.

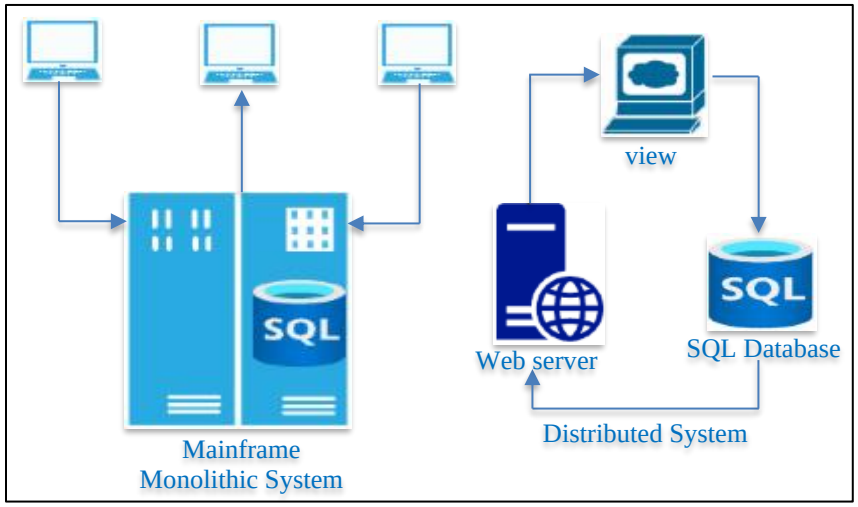

Fig. 1 A logical diagram differentiating a monolithic system from a distributed system

3. Client-Server Programming[8]

In Mainframes, with the proliferation of the use of computers beyond the computation of general ledgers and reconciliation purposes, financial institutions were eager to make the use of computers ubiquitous. They wanted Bank tellers to use the computers, but there was a problem. Bank tellers working for banks are often non-technical people and cannot be expected to interact with computers as scientists or researchers would do. For this use case, IBM introduced Terminals; Terminals (Clients) were lighter interfaces that allowed the mainframe IT systems to capture data and respond to commands while Servers would process the command and respond to the terminals. Thus, Client-Server Programming was introduced. Although this is not a decoupling of systems by any means, this was the first step that got people to believe that multiple computers serving the same or different purposes can possibly be orchestrated to drive greater value.

4. Distributed Architectures

Mainframe applications are monolithic systems. They are big applications written in CICS or COBOL. These systems would have the business logic and data closely knit together. It had all the systems, including the database, sometimes embedded into the same physical hardware or co-located and tightly coupled. The tight coupling had some advantages. Because the hardware was co-located, it had very low latency and so was able to support a high volume of transactions despite relatively low compute and memory as we have now-

a-days. But because everything was closely packed, it had several disadvantages. It was limited in scope as it was not able to integrate with low-cost PCs seamlessly, so affordability was low. It was very specifically tied to a single vendor, and so had the vendor lock-in and all the disadvantages of a market when dealing with a monopoly business. To break free, distributed computing showed the light, and Distributed architecture evolved to show the way. Distributed architecture is a lightly coupled architecture that allows computers serving different functions to be logically and physically separated from each other and communicate over common networking protocols such as TCP/IP9.

5. MVC Architecture

The 3-tier architecture became popular and came to be known as the MVC architecture or Model-View-Controller (MVC) Architecture[10]. In MVC architecture, there is a model, which is a representation of an entity. Referring to our example, the model is the entity which represents the customer. Now the attributes of the customer describe the model. So, the name, address, phone number, email, etc., are all the attributes of the customer entity. Likewise, there will be several entities that will define the models to be used in the application. Usually, the database to persist customer information is a replication of the model in a tabular format that helps to store that information. Next is the View, which is the user interface. Usually, in any application, there is a user interface for each channel of experience, so websites for browsers, mobile apps for Mobile, Point of Sales (POS) devices for point of sales and so on. Controller is the component that resembles the Server of the client-server programming. It was a module with all the business logic; it would process the commands/ requests coming from the views, and it would structure the information that can be contained in the model and send it back.

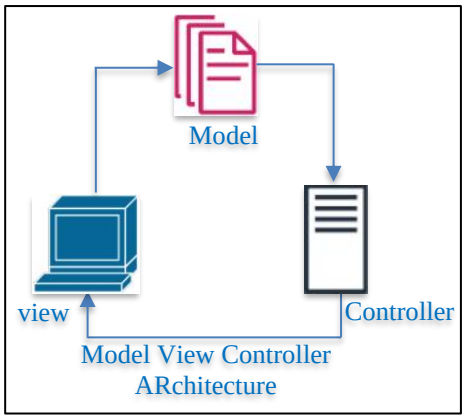

Fig. 2 A logical diagram with the flow of control in MVC Architecture

6. Evolution of Distributed Systems

With the proliferation of distributed computer systems, the need to connect these systems to exchange data and business logic soon became apparent. But there was a challenge. These distributed systems are built in different technology stacks and so some common protocols had to be established to connect these systems directly. Remote procedure call (RPC) was one of the technologies that is heavily used even today to communicate from one system to another. However, the proliferation of the internet, which was largely an outcome of distributed architecture, brought us a host of different technology stacks and a myriad of different approaches to connecting systems.

6.1. The Evolution of Middleware

After The first generation of middleware systems was conceptualized as sending and receiving packets of information to and from different systems; these could be unicast or multicast. Messages. Unicast is analogous to a term we use nowadays called *point-to-point* and Multicast is analogous to a concept called *Broadcast*. This architecture was called Message Oriented Middleware (MOM)[11]. WebSphere (later brought by IBM) were the first to commercialize these as products.

7. Message Broker Architecture[12]

With several systems trying to communicate with each other, directly connecting these systems created a spaghetti mesh over time. Plus, there was no control over the traffic volume. So, it could be easily possible for one system to initiate too many requests to the other system, thereby causing the other system to become slow to process those messages and eventually unresponsive. Transactional consistency was also missing. Suppose multiple systems are directly reaching out to each other. In that case, it is possible that they are extracting the different states of the same transaction, and thereby, systems may not be in an eventually consistent state. So, it was important to have a central system, often called a message broker, that is responsible for channeling the messages to each system and will be responsible for the governance. This is also called the hub and the spoke model or later came to be known as the Enterprise Service Bus Architecture[13].

8. The Enterprise Service Bus (ESB) Pattern

By this time, financial service companies have built out a plethora of applications, either procured or developed using open-source technologies that cater to a variety of needs. Other industry verticals are rapidly catching up with the commercial use of computers in their business beyond the research areas, given the gains in efficiency the financial companies were experiencing. There was a strong demand and market for standards for connecting these systems in a scalable way. IBM came up with a couple of different products to address the needs of the market. One was the WebSphere application server and the WebSphere Message Queue, popularly called IBM MQ. They allowed applications to serialised14 packets of information called messages that adhere to certain protocols.

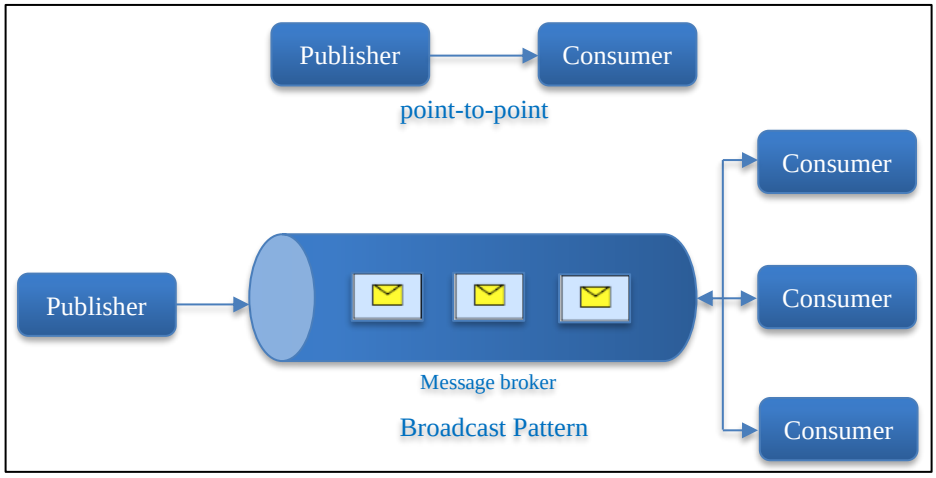

Fig. 3 Message broker architecture with 2 common patterns, the point-to-point and broadcast pattern

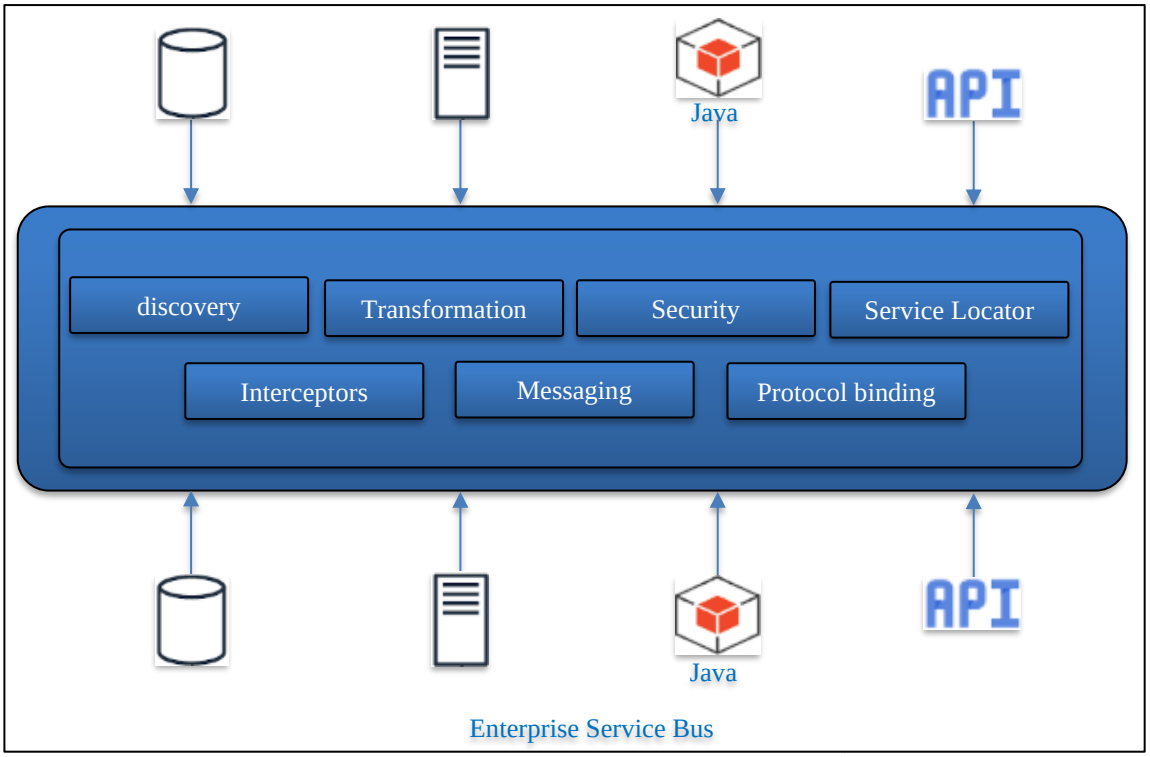

Fig. 4 Enterprise Service Bus pattern with distributed applications connecting to a common Service Bus for message transportation and routing. It allows for orchestration and transformation of messages as well

The broker is the hub at the center and takes care of the distribution, retry, error handling and governance of these message exchanges between these systems. Distribution of these brokers was developed by open-source projects (the ActiveMQ from Apache software foundation) and became very popular in the use of commercial applications. (Figure 1)

The centralization of the infrastructure served to avoid the spaghetti mess of interconnected systems and provided a structure. The centralization allowed for the reuse of those systems as necessary. For example, in our bank account opening example, the successful opening of the bank account can be broadcasted to a few downstream applications such as a) Issuance of the Debit card, b) sending a welcome email, c) sending an email to capture privacy policy and d) capturing communication and notification preferences. This could be achieved using a single flow instead of wiring out all the systems directly.

As enterprises scaled the adoption of computer information systems, the number of systems and applications within a typical enterprise grew exponentially, and so did the

volume of messages and interactions. The word "Enterprise" in the enterprise service bus was taken too literally as large and complex organizations onboarded thousands of applications into their ESB systems, creating some scale challenges. Any downtime for these systems meant a significant business impact. Introducing changes, updates, or bug fixes into the message broker system was risky for the same reasons as downtime. There were cost implications of running such parallel systems for disaster recovery and high availability. The development of system integrations was deemed complex because of scaling challenges and inherent complexities associated with these systems. There was no good way to discover the capabilities already built, so although the technology supported high reuse, it was practically nonexistent as developers were not aware of them or they had difficulty aligning on Model attributes that allowed for reuse.

9. Service Oriented Architecture

In the late 1990s and the early 2000s, with the proliferation of the internet, HTML came into the forefront and TCP/IP became the standard of communication between distributed systems in the internet—the need to communicate externally. While ESB was widely embraced and was in full swing, there was an attempt to overcome some of the drawbacks of the ESB architecture pattern with an alternative—Service Oriented Architecture[15] (SOA). SOA started on the fringe of ESB and is embraced into organizations typically as a supplement to an ESB architecture pattern rather than an alternative. So, while ESB was used to integrate different applications by the transfer of messages orchestrated through a common hub, SOA allowed exposing the functionalities of the applications as a service using a standard Interface.

This was quite revolutionary. It allowed large enterprise applications, built-in whatever technology stack (Java, Microsoft .Net), to communicate with any other systems, built-in some other technology stack, to communicate with each other using a standard protocol (usually SOAP) over the TCP/IP network. The Service Oriented Architecture came up with the concept of a registry of these Services, which would allow the services to be easily discovered from a centralized location. Universal Description, Discovery, and Integration (UDDI[18]) became the standard specification for describing and enlisting web services, built as part of the Service oriented architecture. Simple Object Access Protocol (SOAP) allowed the web services to publish a specification that other services or applications could use. While this was not designed to replace the ESB architecture pattern, it allowed greater adaptability and faster time to market and yet allowed for reuse, avoiding the hard wiring of interconnected applications.

10. Rest APIs

While SOAP services are still here, the industry swiftly adopted a lighter and easier approach to communicate over the web using the same HTTP protocol that is popularly used to communicate over the internet. REST web services16 are lightweight. Does not mandate a strict adherence to standards and specifications, although it is always good to have one. REST, because of its adherence to HTTP protocol, became popular very quickly and found wide adoption amongst Internet applications. Nowadays these web services are better known as Application Program Interface (APIs).

11. Microservices Architecture

As wide adoption of Information technology-enabled services became mainstream, there was an exponential growth of products and services available to all industries. Some were specifically catering to financial services, while others were cross cutting and applied to all industries. The heterogeneous growth of services challenged the large monolithic enterprise applications, which were coined legacy applications. Legacy applications have their share of challenges. It was difficult to keep pace with innovation, given the time it takes to make changes and test it rigorously to get it live. Also, the new set of technologies was built with the latest advancements in programming languages, considering the enhanced memory and computer of the 21st-century hardware. It was a little difficult for the legacy technologies to keep pace with them.

Microservices[17] architecture evolved as a popular choice. Microservices were first introduced by Netflix as the alternative technology that they were using to support their unusual business needs to be able to stream high-quality video over the internet that is catalogued and cached based on frequency. Microservices is about taking atomic functionalities and exposing them as lightweight Services. So, functionalities within the same applications can be exposed securely as services. These Services are then governed for security, throughput, etc., through a service proxy. (Istio or envoy).

The benefit of this architecture is that it allows for independent development and deployment of services without touching the related services. It is easy to scale these services independently without scaling the whole ecosystem of applications. Service-to-service communication can be seamlessly achieved via APIs. Microservices follow a polyglot model, which means that each service can be developed in any technology stack.

Microservices architecture took a big leap with the adoption of cloud technologies. Several organizations are on their path to migrate legacy monolithic applications to cloud and microservices architecture is widely adopted to be successful. While microservices are the latest and most advanced approach that software development communities have to offer, it is important to understand that if the organization is not mature with its API management and governance or if they do not have robust CICD capabilities, they may face roadblocks for implementing microservices architecture.

12. Event-Driven Architecture

To complete the circle of evolution, at the start, we had message-oriented Middleware (MOM); this concept has made a comeback in a whole new way as event driven architecture. In Event Driven Architecture, we have the message producers, which produces 1 or more streams of events and the event consumers, who subscribe to these streams of events. Microservices and EDA work in tandem to deliver great scalable results.

The best part of the event-driven architecture (EDA) is that they are asynchronous in nature. We have a lot of practical use cases which benefit from the asynchronous nature of the transaction. So, let's go back to our example: if someone is successfully able to open a bank account, sending them a welcome email does not have to happen synchronously, meaning it can be delayed for a few minutes without much business impact. Such transactions may be decoupled from the original transaction that verifies the identity to open the bank account. The asynchronous nature of the transaction, or in other words, the ability to proceed without having to wait for a response, has helped achieve enormous scale and the ability to follow through with a better customer experience.

13. System Integration

While we now have many different choices for integrating systems, such as implementing an Enterprise Service Bus (ESB), Service Oriented Architecture (SOA), Microservices architecture or even an Event-driven architecture (EDA), it is important to understand how to evaluate them based on the objective metrics. In a large organization with many different lines of business that have grown organically or inorganically, it isn't easy to apply a one-size-fits-all model. There may already be a mix of all these architectures adopted based on the necessities or by business units based on their level of maturity. But, it is important to put things into perspective to understand what a better approach is and how an organization can approach it.

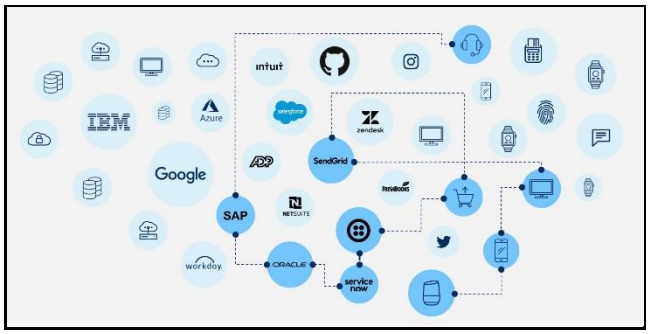

Fig. 5 Proliferation of applications in delivering a complete customer experience

We often see organizations hardwired systems directly, often under the pressure of completing the project faster. It is absolutely evident that while it gives slightly less turnaround time, it is not the right thing to do as it uses repetitive code, takes time, is fragile and increases technical debt. Over a period of time, these implementations become very complex and difficult to maintain. They are also not reusable, so anytime any 2 or more systems are integrated, the past integrations that have already been created of such systems are of no use. So, over a period, the time to deliver a project does not get any faster.

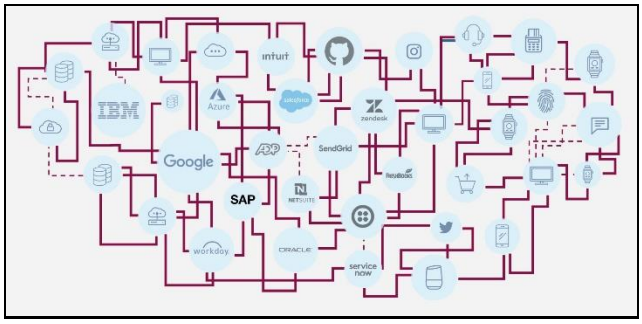

Fig. 6 Point-to-point integration over time creates a spaghetti mess of a network that is difficult to maintain and hard to reuse

There is nothing wrong with Implementing an ESB. But, implementing an ESB is complex, it takes time, and it is hard to maintain. ESB can also introduce a single point of failure, and modern distributed architectures may not find that favorable.

Service Oriented Architectures (SOA) are typically implemented using synchronous SOAP APIs; this was a great advancement to expose web services over legacy monolithic systems but fell short in terms of Service discovery and the lightweight nature of its counterpart, REST APIs.

REST APIs, Microservices and Event-Driven architectures are all very relevant to modern application development. However, careful considerations need to be given as to what is implemented where and how it is being implemented.

Now that we have discussed the complete evolution of System Integration, it is important to understand how an organization can be successful in achieving its system Integration. A good understanding of these technologies is not sufficient to implement these technologies at scale. It is important to recognize that the right combination of people, processes and technology is important to meaningfully integrate the systems and unlock the value of all the underlying systems. Let us start backwards and look at what each element has to offer to build a composable enterprise with flexible integrations.

14. The Technology Platform

To create a business that can weather the effects of constant change, it is important that services are built as

building blocks (as APIs or microservices as appropriate). However, they are easy to plug and unplug with other systems as necessary. Moreover, these connections are nimble and standardized over HTTP as much as feasible.

These APIs or microservices can be synchronous or asynchronous in nature, depending on the nature of the business outcomes and priorities. Event-driven architecture can be built using Kafka, which in turn supports event streaming options. These events-driven systems can be built as Async APIs. Decoupling the request and the response can help in scaling the service and elevate customer experience.

The process plays a pivotal role in making this happen. It involves laying out a change management procedure that starts with business needs trickling down in the format of API specifications. The API specification documents the functions of the business services and models the attributes and the behavior. APIs are decoupled from the business process but are modelled around functionalities and an orchestration of APIs should help to accomplish business functions.

The APIs designed must be published into a catalogue that is accessible to the entire organization. This will promote reuse. API specifications should allow mock testing for use case validation of new businesses. Once the APIs are developed in any technology platform of choice, they should be flexible to be deployed on any platform of choice. This could be on-premises or cloud. However, the design and development of APIs should not be an impending factor in the choice of deployment options. It should allow cloud native deployment by adhering to 12-factor standards.

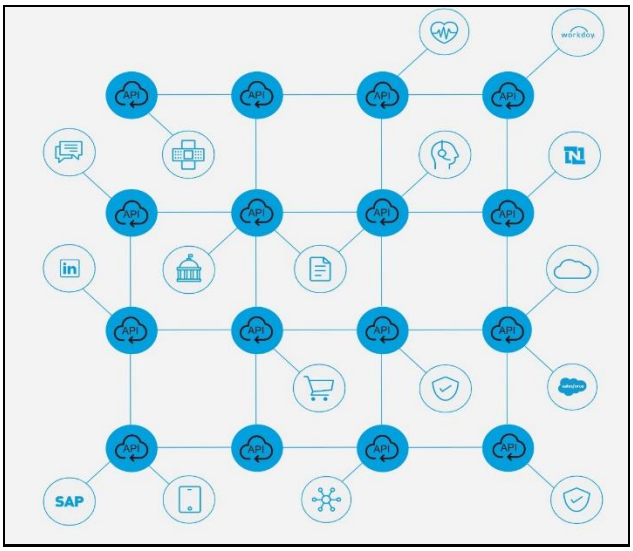

Fig. 6 A integration of systems using APIs as a building block of connectivity that allows for reuse

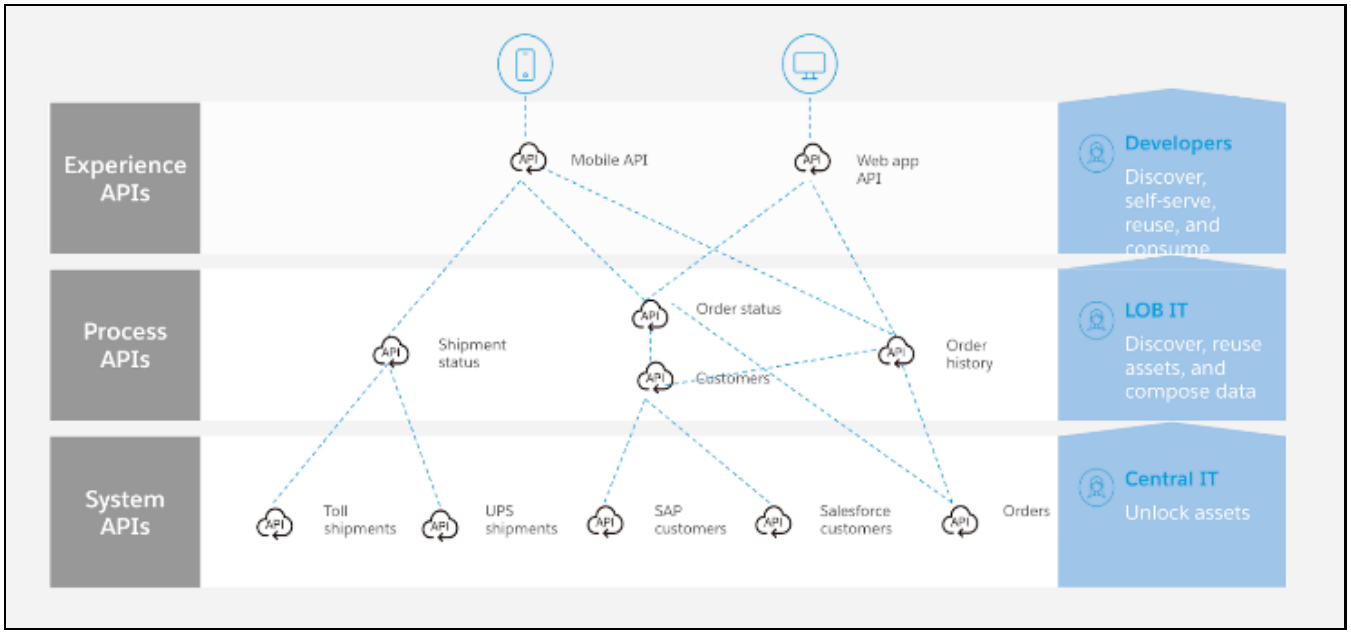

Fig. 7 APIs can be classified as System, Process and Experience; System APIs allow connecting to systems, Process APIs allow for Orchestration and Experience APIs cater to the specific channel

14.1. API Specification

An API specification is usually written in a declarative language such as OAS 3.0, which is the de facto industry standard. API specification helps an API client, the user of the API, understand the API functions and attributes of the structure of the request and response. It also gives information about the publisher of the API, its security policies, etc. This type of document establishes the identity of the API and ensures trust.

14. 2. API Management / API Governance

It is important to establish the security and governance for every APIs. This is typically established via organizational policies and enforced with API management solutions available in the market or some open-source solutions such as Istio or Envoy. At an organizational level, these policies should define the security standards that are linked with the organization's Identity management provider.

Each APIs, based on their business needs, must fall into a throughput layer so that they are adequately resourced to serve. These SLA tiers would restrict the APIs to overwhelm the underlying systems of records that the APIs interact with in case of excessive traffic.

If the APIs are deployed on-premises or cloud, they should be well guarded with perimeter security either by a (Web access firewall) WAF or Shielded via DDoS attacks.

14. 3. API Discovery

Every organization needs to make a conscious attempt to publish an API developer portal. This is nothing but a catalogue of all the published APIs within the organization. At the bare minimum, this API catalogue must be accessible to all the employees of the organization. This would democratize access to every employee and allow them to understand the API already available. If the use case allows for external collaboration, either the same or a separate instance of the API portal with a limited number of external-facing APIs must be exposed to the external world.

15. A Composable Enterprise

"A composable enterprise is an organization that delivers business outcomes and adapts to the pace of business change. It does this through the assembly and combination of packaged business capabilities (PBCs)." - Gartner. In today's world, an average customer's data transcends through 35 applications for any single business event or transaction. A typical mid-sized organization uses 935 different applications to store customer data or implement a business logic that is leveraged in some way or the other. – Forrester

The idea of a composable enterprise does not have to start big. It can start from a single project a from a single business unit and can slowly expand after initial success or validation and slowly be adopted across the enterprise. It is not necessary nor required to align to a single technology stack. Integrations and APIs or microservices can be built on any technology stack.

Moreover, last but most important are the people who can make this happen. Any organization, irrespective of its size and complexity, must have an API Strategy. The API strategy will not be the same for every enterprise. Every organization is on a path to digital transformation. Every organization has some technical debt that has been acquired either organically or inorganically.

The API strategy will define the goals that the organization can set forth to achieve with an effective system integration. The API strategy goals will be achieved through an effective API Program. This API program is focused towards creating APIs as products, carefully curated and maintained and not created as project collaterals. Lastly, these APIs are owned by product owners, who design, secures, and maintain the product from abuse.

References

[1] Michael Castelle, "Middleware's Message: The Financial Technics of Codata," *Philosophy & Technology*, vol. 34, pp. 33-55, 2011. [CrossRef] [Google Scholar] [Publisher Link]

[2] Steve Vinoski, "An Overview of Middleware," *9th Ada-Europe International Conference on Reliable Software Technologies*, Palma de Mallorca, Spain, vol. 3063, pp. 35-51, 2004. [CrossRef] [Google Scholar] [Publisher Link]

[3] Aydin Rashidi, "Customer Relationship Management and its Use in Insurance Industry," *International Journal of Insurance*, pp. 1-20, 2012. [CrossRef] [Google Scholar]

[4] Fikri Aydemir, and Fatih Başçiftçi, "Building a Performance Efficient Core Banking System Based on the Microservices Architecture," *Journal of Grid Computing*, vol. 20, 2022. [CrossRef] [Google Scholar] [Publisher Link]

[5] Mohsen Mosleh, Kia Dalili, and Babak Heydari, "Distributed or Monolithic? A Computational Architecture Decision Framework," *IEEE Systems Journal*, vol. 12, no. 1, pp. 125-136, 2018. [CrossRef] [Google Scholar] [Publisher Link]

[6] N.H. Giri, V.N. Nandgaonkar, and Rahul Gosavi, "Virtual Operating System for Windows to Linux Migration," *2017 International Conference on Energy, Communication, Data Analytics and Soft Computing (ICECDS)*, Chennai, India, pp. 2125-2127, 2017. [CrossRef] [Google Scholar] [Publisher Link]

[7] Subarna Shakya et al., "Distributed High-Performance Computing Using JAVA," *2017 International Conference on Computing, Communication and Automation (ICCCA)*, Greater Noida, India, pp. 742-747, 2017. [CrossRef] [Google Scholar] [Publisher Link]

[8] Rolou Lyn R. Maata et al., "Design and Implementation of Client-Server Based Application Using Socket Programming in a Distributed Computing Environment," *2017 IEEE International Conference on Computational Intelligence and Computing Research (ICCIC)*, Coimbatore, India, pp. 1-4, 2017. [CrossRef] [Google Scholar] [Publisher Link]

[9] Hong RiLi, "Research and Application of TCP/IP Protocol in Embedded System," *2011 IEEE 3rd International Conference on Communication Software and Networks*, Xi'an, China, pp. 584-587, 2011. [CrossRef] [Google Scholar] [Publisher Link]

[10] Andreas Holzinger, Karl Heinz Struggl, and Matjaž DebevcApplying, "Model-View-Controller (MVC) in Design and Development of Information Systems: An Example of Smart Assistive Script Breakdown in an e-Business Application," *2010 International Conference on e-Business (ICE-B)*, Athens, Greece, pp. 1-6, 2010. [Google Scholar] [Publisher Link]

[11] Mohammad Kazem Haki, and Maia Wentland Forte, "A Service Oriented Enterprise Architecture Framework," *2010 6th World Congress on Services*, Miami, FL, USA, pp. 391-398, 2010. [CrossRef] [Google Scholar] [Publisher Link]

[12] Guangxuan Chen et al., "Research of JMS-Based Message Oriented Middleware for Cluster," *2013 International Conference on Computational and Information Sciences*, Shiyang, China, pp. 1628-1631, 2013. [CrossRef] [Google Scholar] [Publisher Link

[13] Hafiyyan Putra Pratama, Ary Setijadi Prihatmanto, and Agus Sukoco, "Implementation Messaging Broker Middleware for Architecture of Public Transportation Monitoring System," *2020 6th International Conference on Interactive Digital Media (ICIDM)*, Bandung, Indonesia, pp. 1-5, 2020. [CrossRef] [Google Scholar] [Publisher Link]

[14] Min Luo, and Liang-Jie Zhang, "Practical SOA: Service Modeling, Enterprise Service Bus and Governance," *2008 IEEE Congress on Services Part II (Services-2 2008)*, Beijing, China, pp. 13-14, 2008. [CrossRef] [Google Scholar] [Publisher Link]

[15] Tobias Simon et al., "A Lightweight Message-Based Inter-Component Communication Infrastructure," *2013 Fifth International Conference on Computational Intelligence, Communication Systems and Networks*, Madrid, Spain, pp. 145-152, 2013. [CrossRef] [Google Scholar] [Publisher Link]

[16] Ridhima Mishra et al., "Transition from Monolithic to Microservices Architecture: Need and Proposed Pipeline," *2022 International Conference on Futuristic Technologies (INCOFT)*, Belgaum, India, pp. 1-6, 2022. [CrossRef] [Google Scholar] [Publisher Link]

[17] Mehdi Bahrami, and Wei-Peng Chen, "Composing Web API Specification from API Documentations through an Intelligent and Interactive Annotation Tool," *2019 IEEE International Conference on Big Data (Big Data)*, Los Angeles, CA, USA, pp. 4573-4578, 2019. [CrossRef] [Google Scholar] [Publisher Link]

[18] Yongxin Feng, and Qin Li, "The Distributed UDDI System Model Based on Service-Oriented Architecture," *2016 7th IEEE International Conference on Software Engineering and Service Science (ICSESS)*, Beijing, pp. 585-589, 2016. [CrossRef] [Google Scholar] [Publisher Link]

[19] Ian Thomas Newcombe, "*Mainframe Relevance in Modern IT: How a 50+ Year Old Computing Platform Can Still Play a Key Role in Today's Businesses*," University of New Hampshire, Durham, pp. 1-44, 2016. [Google Scholar] [Publisher Link]

[20] Campbell-Kelly Martin, and Daniel D. Garcia-Swartz, *From Mainframes to Smartphones: A History of the International Computer Industry*, Harvard University Press, pp. 1-220, 2015. [CrossRef] [Google Scholar] [Publisher Link]

[21] Tasneem Salah et al., "The Evolution of Distributed Systems Towards Microservices Architecture," *2016 11th International Conference for Internet Technology and Secured Transactions (ICITST)*, Barcelona, Spain, pp. 318-325, 2016. [CrossRef] [Google Scholar] [Publisher Link]

[22] Andy Neumann, Nuno Laranjeiro, and Jorge Bernardino, "An Analysis of Public REST Web Service APIs," *IEEE Transactions on Services Computing*, vol. 14, no. 4, pp. 957-970, 2021. [CrossRef] [Google Scholar] [Publisher Link]

[23] Dominic Lindsay et al., "The Evolution of Distributed Computing Systems: From Fundamental to New Frontiers," *Computing*, vol. 103, pp. 1859-1878, 2021. [CrossRef] [Google Scholar] [Publisher Link]