# Microservices Architecture

*A White Paper by:*

Somasundram Balakrushnan, Salesforce.com (MSA Project Co-Chair)
Ovace Mamnoon, Hewlett Packard Enterprise (MSA Project Co-Chair)
John Bell, Ajontech LLC
Benjamin Currier, Hewlett Packard Enterprise
Ed Harrington, Conexiam
Brian Helstrom, IBM
Peter Maloney, Raytheon Company
Marcelo Martins, IBM

July 2016

**Microservices Architecture**

Document No.: W169

Published by The Open Group, July 2016.

Any comments relating to the material contained in this document may be submitted to:

The Open Group, 44 Montgomery St. #960, San Francisco, CA 94104, USA

or by email to:

ogpubs@opengroup.org

# Table of Contents

**Executive Summary ..... 5**

**What is a Microservices Architecture? ..... 7**

Microservices Architecture Style ..... 7

The Problem Space ..... 8

Combination of Distinctive Features ..... 10

Components, their Interaction and Governance ..... 10

Granularity ..... 10

Built on Established Heritage ..... 11

Key Defining Characteristics of an MSA ..... 12

Other Related Characteristics of MSA ..... 13

Key Governing Principles of Microservices Architecture ..... 15

**SOA and MSA ..... 16**

Question 1: Vision and Intent Comparison ..... 17

Question 2: Entry Criteria/Applicability for Using One Style ..... 18

Question 3: Business Drivers ..... 19

Question 4: Characteristics Comparison ..... 20

Question 5: Architecture Paradigm and Style Comparison ..... 20

Question 6: Architectural Principles ..... 20

**CASE STUDY: MSA for a Hotel Central Reservation System ... 22**

Introduction ..... 22

Business Scenario ..... 22

MSA-Based Solution ..... 23

Result ..... 23

Conclusion ..... 24

**CASE STUDY: Rainyday Grocer ..... 25**
Introduction ..... 25
Business Scenario ..... 25
MSA-Based Solution ..... 26
Results ..... 31
Conclusion ..... 31

**APPENDIX A: Service Granularity ..... 32**

**Glossary ..... 34**

**References ..... 37**

**About the Authors ..... 38**
Somasundram Balakrushnan, Salesforce.com (MSA Project Co-Chair) ..... 38
John Bell, Ajontech LLC ..... 38
Benjamin Currier, Hewlett Packard Enterprise ..... 38
Ed Harrington, Conexiam ..... 38
Brian Helstrom, IBM ..... 38
Peter Maloney, Raytheon Company ..... 38
Ovace Mamnoon, Hewlett Packard Enterprise (MSA Project Co-Chair) ..... 39
Marcelo Martins, IBM ..... 39

**Acknowledgements ..... 40**

**About The Open Group ..... 41**


*Boundaryless Information Flow™*
*achieved through global interoperability*
*in a secure, reliable, and timely manner*

## Executive Summary

There is much debate about what constitutes a Microservice and a Microservices Architecture (MSA), and whether they represent an evolution or a revolution.

This White Paper presents the viewpoint of The Open Group SOA Work Group, developed through diligent research of the current viewpoints in the industry Work Group members. It provides a clear and specific definition of Microservices and MSA, distills their core principles and key characteristics, and provides a comparison of MSA with Service-Oriented Architecture (SOA).

There is growing impetus in the industry for agility, cost optimization, and shifts between Capital Expenses (CapEx) and Operating Expenses (OpeEx). There is the advent of cloud computing and the Internet of Things (IoT). Digital enterprise transformation is underway. MSA is proving to be the enabler for all of these, provided it is done right. Also, MSA is conducive to the DevOps paradigm and evolving cloud-based architecture. The discussion and case studies in this White Paper highlight the recommended approach and the best practices in undertaking the transformation journey to MSA.

But MSA is by no means a silver bullet. This White Paper presents both the ideal and the pragmatic approach to MSA: developing an understanding of where MSA is the right fit, and recommending a role for MSA in the Enterprise Architecture portfolio.

### *Microservices Architecture*

Like most distributed architectures, MSA is also prone to the impacts of CAP theorem[1] and data consistency complexities. Close attention to architecture and design should be paid to ensure appropriate data storage and management such that the right consistency (strong or eventual) is achieved.

The paper delves into the merits of the "designed to fail" paradigm. Microservices can fail during runtime for many reasons. However, planning for dealing with this failure and providing resilience is a key part of an MSA.

The MSA style of architecture enables The Open Group vision of Boundaryless Information Flow™. The idea that single units of business functionality are available as services across the organization makes it easier to invoke such functionality within and between enterprises. It means that a part of an enterprise can find and use the information it needs as provided by other parts of the enterprise or other enterprises.

[1] Refer to https://en.wikipedia.org/wiki/CAP_theorem.

# What is a Microservices Architecture?

The Open Group defines an *architecture* (in the TOGAF® 9.1 standard) as: "the structure of components, their inter-relationships, and the principles and guidelines governing their design and evolution over time". It defines *architectural style* as: "the combination of distinctive features in which architecture is performed or expressed".

Leveraging the above definitions, this White Paper examines and defines MSA in terms of:

- The problem space
- The combination of distinctive features
- Components, their interaction, and governance

## Microservices Architecture Style

MSA is a style of architecture that defines and creates systems through the use of small independent and self-contained services that align closely with business activities.

An individual *microservice* is a service that is implemented with a single purpose, that is self-contained, and that is independent of other instances and services. Microservices are the primary architectural building blocks of an MSA.

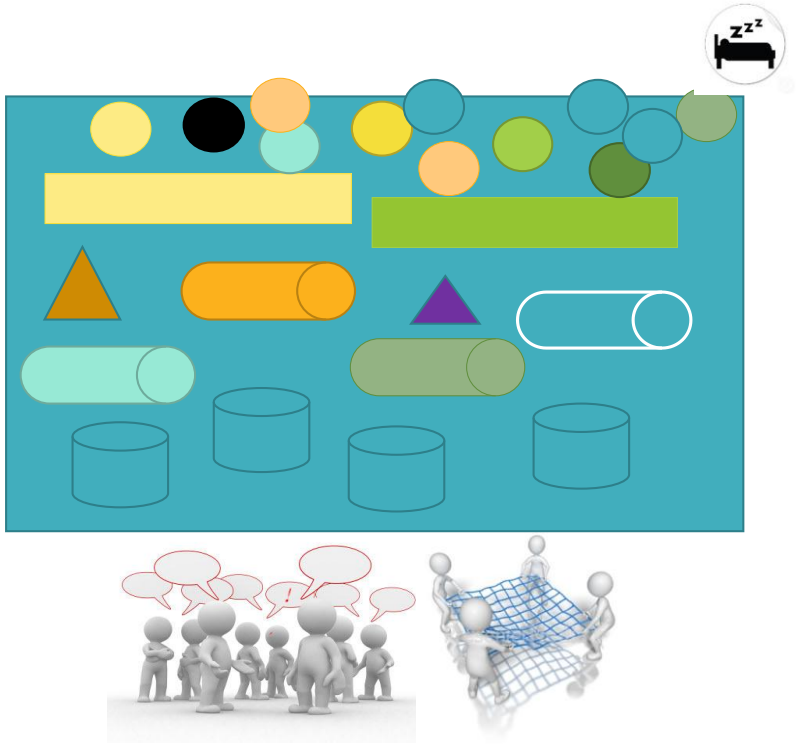


Figure 1: Conceptual Representation of a Monolith Application

### *Microservices Architecture*

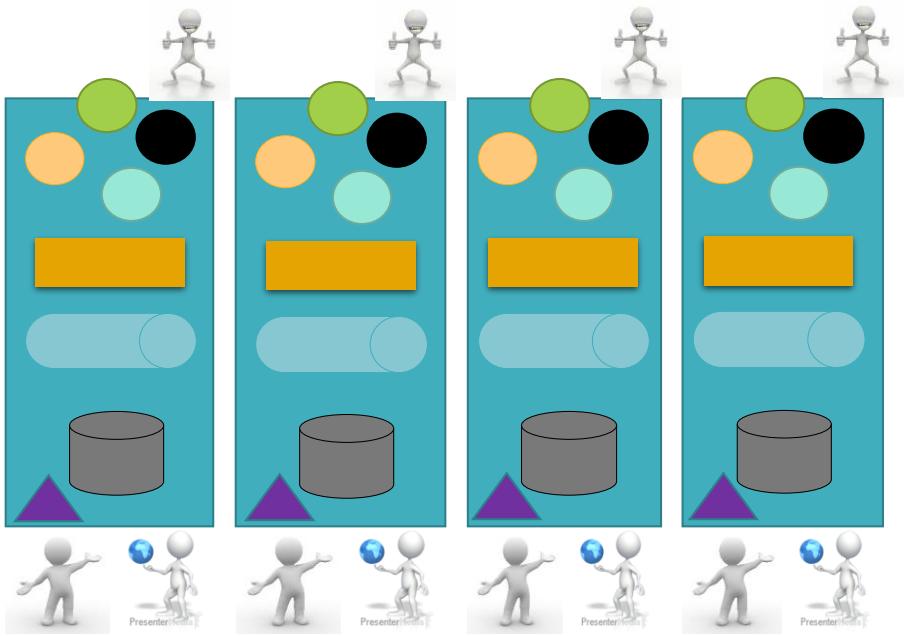


Figure 2: Conceptual Representation of a Microservices Solution

## The Problem Space

The need for MSA is driven by the pain points below.

***Decreasing the complexity of the development, operation, and management of services***

Changes to components in solutions of interdependent applications and services has the challenge of even a small fix requiring a long change cycle as the entire solution must be validated or, in the case of a monolithic application, rebuilt. Even SOA applications may have dependencies and interactions required by services choreography.

The modularity of the components of a non-MSA application tends to weaken over time, making it harder and harder to make changes to only one small part of an application. The independent service nature required for an MSA allows for much easier development and deployment, since the services are both independent and self-contained.

Scaling of a monolithic or tightly-coupled application requires scaling of the entire application solution, rather than just the portion that is demanding more resources. Non-MSA SOA applications may have dependencies or sophisticated choreography requiring additional changes. Scaling of an MSA is achieved much more easily because instantiation of additional services is performed independently.

***Simplify to decrease functional bloat***

Styles other than MSA often expand the scope of the component parts by adding new functionality to a component or its interdependencies. This complicates the solution or application leading to bloat and demand on the system. With MSA, keeping each service independent and self-contained to a single atomic business function reduces this risk of bloating.

## *Microservices Architecture*

#### *Allow faster response to changes in the marketplace or entry into a new market*

An MSA is synchronized to business unit change velocity needs.

- There is joint ownership of the microservices by the business unit and the development team.
- As a result of joint ownership and the principles of single responsibility, independence, and self-containment, the microservices can be changed to cater to the changing needs of the business unit, irrespective of other business units of the enterprise.

MSA development and deployment does not require a large pre-existing infrastructure.

- Since a microservice is based on single atomic business functions they tend to be smaller.
- Therefore, due to the MSA characteristics, microservices can be developed in small environments. A large integrated solution or large monolithic application requires large investment in a development environment to support the management of even a small change.

The time and cost associated with development, operations, and management of large enterprise applications are reduced.

- Testing of a large enterprise application is a significant investment in time and resources, even for relatively minor updates to the application.
- Monolithic applications with centralized governance and data management tend to become dependent on a single technology addressing all business functions.
- Coordination and management of large developer teams requires a large amount of effort to be devoted to these overhead functions.

#### *Advent of paradigms such as Continuous Integration (CI) and Continuous Deployment (CD), and the inability of the monoliths to meet these, as exemplified by the DevOps culture*

The DevOps culture emphasizes the collaborative nature of application development and maintenance. It focuses on the elimination of silos between development and operations; i.e., the use of autonomous teams which own the entire life cycle of an application. The distributed, independent nature of microservices lends itself naturally to implementation in this kind of culture.

Continuous Integration (CI) and Continuous Deployment (CD) rely on near-constant integration of software, with automated deployment of the integrated code. The inherent modularity and independence of an MSA and its services lends itself very well to the CI and CD paradigms carried out by a number of distributed teams working off a controlled code base.

#### *The advent of cloud and the need to distribute workload elastically between on-premise and cloud*

MSA enables capabilities such as Web-Oriented Architecture (WOA) as well as allowing for leveraging the resilience and scalability offered by cloud.

Service independence allows for services to run anywhere and so cloud becomes a natural place to expand through adding more instantiations in support of demand or through expanding functions and services to an application through the addition of new independent services, which can reside in the cloud.

#### *Highly responsive user experience*

The resilience provided through the parallelism of an MSA, which is enabled by the independence of the microservices, permits rapid failover and self-recovery. These characteristics result in high availability and seamless user interaction.

### Combination of Distinctive Features

An MSA will generally contain the following features:

- Software components are broken down into independent services (this is the fundamental feature of an MSA).
- Services are independently deployable.
- Services are mapped to atomic business capabilities.
- Services are fully decoupled.
- Scalability and resilience of the application are achieved through the independence of services and multiple parallel instances of services.

### Components, their Interaction and Governance

The key characteristic of MSA is that software components are implemented as services, rather than, for example, as libraries. These services are independently replaceable and deployable. Services are typically implemented using mandatory well-defined, published interfaces. An MSA will aim for defined service boundaries to avoid interface changes which will affect multiple services, but in some cases this may be unavoidable.

The applications built using MSA should keep the microservices decoupled and fully independent. Any choreography in an MSA is performed by the initiating application and not from within or by the microservices. This is done by leveraging simple protocols (such as REST) rather than complex products or a central tool.

Governance in an MSA is decentralized or distributed. This allows the developer teams to focus on determining the best technology platform or language implementation to solve their particular problem, and not be trapped in a "one size fits all" paradigm.

### Granularity

Determining the right granularity of services in an MSA is more of an art than a science.

However, it is imperative for organizations to set the rules of thumb up-front in determining the right level of granularity. Taking a too coarse-grained approach to services can result in a "monolith", while taking a too fine-grained approach can result in an anti-pattern, referred to as "nano-services" (see APPENDIX A: Service Granularity).

A *granularity line* can be defined based on business activity, business agility, and other considerations.

### *Microservices Architecture*

Services that fall around the granularity line are considered *microservices*; those way above are the *monoliths* and those way below are the *nano-services*.

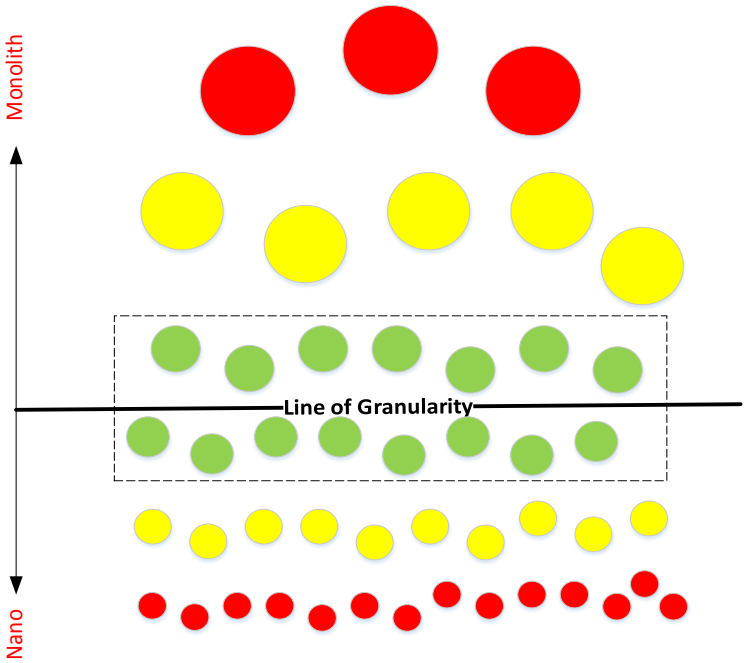



Figure 3: Right Level of Granularity

## Built on Established Heritage

An MSA builds on many well established architectural, design, development, and operations paradigms such as object-oriented programming, SOA (see the chapter on SOA and MSA), and Domain-Driven Design (DDD), which provides a set of principles and methods to manage complexity by identifying core and ancillary domains and addressing architecture, development, operations, teams, etc. within the bounded context of each domain.

Another very relevant model is the scale cube presented in the referenced book "The Art of Scalability".[2] The scale cube defines a model for scalability through decomposition. Of particular interest in the scale cube is the y-axis which focuses on functional decomposition or services.

[2] Refer to http://theartofscalability.com.

## Key Defining Characteristics of an MSA

The definition of MSA, earlier in this White Paper, articulates that it is composed of microservices as building blocks. This section does not distinguish between the characteristics that are specific to the microservices *versus* ones that are applicable to MSA. The intent is to identify the key characteristics of a solution that is architected as an MSA and, as such, consists of microservices.

Most of these characteristics are interrelated to some degree.

#### *Service Independence*

A key imperative for MSA is *service independence*. As business needs change and evolve, the impact to every service must be manageable. Service independence minimizes the impact to the service infrastructure by identifying and isolating those services that undergo constant churn. Once identified, these services should be upgradeable or replaceable without any additional changes to the software landscape. Since each service is also self-contained, redeployment of each independent service is simplified since there are no service dependencies to manage. Simpler service redeployments streamline release processes since fewer software assets need to be managed from release to release.

A microservice is independent of other microservices or other services. Instances of a microservice are independent of the other instances of the same microservice. Development and deployment of a microservice is independent of the development and deployment of other services.

#### *Single Responsibility*

*Single responsibility*, defined by MSA, is the direct alignment of a service to a singular business activity. For the purpose of this discussion, a business activity can be described as a unit of work performed by the organization that supports an existing business process or function. For single responsibility, the obligation of the service is to map completely to the business activity and deliver whatever business logic is necessary to fulfill the activity. Having each service associated with a single business activity enables the tracking of business change impact through the software service landscape. Impacted services can be more readily and easily identified when supporting a single business responsibility.

It is important to decompose the target business process to the right level to achieve the single responsibility, aligned to an atomic business activity, as well as to achieve the decoupled nature to support the resilience and scalability of the solution.

The decomposition can be guided by the scale cube and its three dimensions of scaling, particularly the y-axis scaling, which emphasizes the functional decomposition through splitting of applications into multiple microservices.

The other influencing concept is the bounded context from DDD, which can help determine the right level of decomposition through context mapping.

#### *Self-Containment*

*Self-containment* dictates that a service shall encompass all external IT resources (e.g., data sources, business rules) necessary to support the business activity. In addition, self-containment necessitates that service dependencies falling outside the scope of the development team should be minimized or preferably

eliminated. By adhering to self-containment, services are inherently more easily replaceable and upgradeable. Along with single responsibility, this characteristic also promotes single ownership of the service since it encapsulates not only specific business requirements (single responsibility) but also specific software dependencies (self-containment).

Microservices are packaged with containers and components as single-deployment units.

The self-containment principle does not remove the need for orchestration between microservices. The orchestration function is moved to the application and user interaction layer. Hence, to achieve self-containment, proper granularity in decomposition is essential.

#### *Highly Decoupled*

In order to maintain minimal service dependencies, microservices must be *highly decoupled*. To achieve this decoupling, the business function must be capable of being decomposed down to the level where a microservice is implementing a single atomic business function. It is this decomposition, and the consequent removal of dependencies between the atomic business functions, that permits the service independence and self-containment required in an MSA.

By relying heavily on existing protocols whose sole purpose is to route messages (HTTP, for example), the needs of a service can be reduced to receiving the message request, applying the appropriate business logic, and generating a message response. MSA discourages the use of external data transformation or protocol bridging services since they introduce additional tightly-coupled service dependencies that will need to be managed during upgrades. MSA endorses the concept of *smart endpoints*, where all logic needed to manage the incoming request and produce an appropriate response remains encapsulated within the service.

#### *Highly Resilient*

An MSA must be a highly-resilient architecture. Its microservices must be designed for potential failures because individual service failures should not impinge negatively on the user experience. Since a microservice represents a single responsibility and is self-contained, a service failure could mean that a given business function or process is unable to complete successfully. Lengthy service downtime could also have a significant impact on the entire business. Thus, mechanisms must be in place to ensure timely service recovery. Real-time service monitoring can provide a proactive means for identifying services that are struggling under heavy load or unable to satisfy existing Service-Level Agreements (SLAs). Real-time monitoring of business metrics also provides insight to future business activity changes that will ripple down to IT services. Further, because of the independence of the underlying microservice, new instantiations will be automatically deployed immediately to allow for business continuity without the resolution of the failure itself.

### Other Related Characteristics of MSA

#### *Decentralized Data Management*

Data management is performed by individual services, which are not managed or choreographed in the performance of data updates. Data consistency is achieved eventually, rather than instantaneously.

## Microservices Architecture

#### Implementation-Agnostic

The implementation can support multiple development platforms (e.g., Java®, C++, JS, Ruby, Python, etc.) and technologies, and can be deployed in any of the containers or virtual machines that support these development platforms.

The focus here is on the solution and not each individual microservice. Individual microservices may not be implementation-agnostic, but the solution is.

#### Scalability and Resilience through Parallelism

Though each microservice will fail, the solution is highly resilient. This is achieved by instantiating multiple instances of a microservice in parallel.

This ability to deploy multiple parallel instances results in elasticity and scalability. The number of parallel instances can be increased or decreased to meet the workload demands.

For this characteristic, instrumentation and monitoring as listed below is a requirement.

#### Well-Defined Interface with a Published Contract

The interface (*API*) for a microservice is well-defined and published (i.e., available to the general developer community). This API is consistent across MSA implementations, to encourage re-use and avoid breakage (point-to-point integration).

#### Allows Independent Governance

This is consistent with the idea of a single team ownership for the cradle-to-grave lifecycle of a microservice. Such a team owns every aspect of its microservices including governance. Hence, governance may be decentralized and autonomous.

#### Single Team End-to-End Ownership

One team will own all aspects of a microservice. Its governance, development, testing, deployment, and operations. This is a DevOps model.

#### Instrumentation to Support Elasticity and Resilience

An MSA should have provision for instrumentation. An MSA must have the ability to monitor the services and dynamically create instances as needed.

Elasticity is the ability of a system to autonomously and dynamically adapt its capacity to handle varying workloads. Proper instrumentation and monitoring ensure the overall solution is resilient and scalable.

### Key Governing Principles of Microservices Architecture

| No. | Principle Name | Statement | Rationale | Implications |
|---|---|---|---|---|
| 1 | **Independent Services** | A microservice is independent of all other services. | Independence of services enables rapid service development and deployment, and permits scalability through instantiation of parallel, independent services. This characteristic also provides resilience; a microsservice is allowed to fail and its responsibilities are taken over by parallel instantiations (of the same microservice), which do not depend on other services. When a microservice fails, it does not bring down other services. | Both design and runtime independence of services are required. It is necessary for the business to determine whether providing scalability and resilience of the business function are paramount considerations. If so, MSA provides a means of achieving these characteristics. |
| 2 | **Single Responsibility** | A microservice focuses on one task only and on doing it well. A microservice focuses on delivering a small specific business capability. | This develops ideas including the principle of *Single responsibility, Open-closed, Liskov substitution, Interface segregation, and Dependency inversion* (SOLID), which is a tenet of Object-Oriented Design (OOD) and the *core business domain* and *bounded context* of DDD.<br>Microservices are aligned to atomic business functions, which can be modified and deployed independently. To achieve this it is critical that each microservice caters to a single functional responsibility. | This requires business function decomposition into atomic functional services and data exchanges. |
| 3 | **Self-Containment** | A microservice is a self-contained, independent deployable unit. | In order for a microservice to be independent, it needs to include all necessary building blocks for its operation, or there will be dependencies to external systems and services. | This has architecture, design, implementation, and deployment implications. |

## SOA and MSA

In order to better understand the contrast of MSA with (SOA, let us first understand the definition of an SOA. The standard definition from The Open Group[3] says that service-orientation is a way of thinking in terms of services and service-based development and the outcomes of services. A service:

- Is a logical representation of a repeatable business activity that has a specified outcome (e.g., check customer credit; provide weather data, consolidate drilling reports)
- Is self-contained
- May be composed of other services
- Is a "black box" to consumers of the service

An *architectural style* is the combination of distinctive features in which architecture is performed or expressed. The SOA architectural style has the following distinctive features:

- It is based on the design of services – which mirror real-world business activities – comprising the enterprise (or inter-enterprise) business processes.
- Service representation utilizes business descriptions to provide context (i.e., business process, goal, rule, policy, service interface, and service component) and implements services using service orchestration.
- It places unique requirements on the infrastructure – it is recommended that implementations use open standards to realize interoperability and location transparency.
- Implementations are environment-specific – they are constrained or enabled by context and must be described within that context.
- It requires strong governance of service representation and implementation.
- It requires a "litmus test", which determines a "good service".

In SOA a service may be composed of other services; in MSA we define a service as independent and self-contained, which implies that it cannot be composed of other services. Herein lies one of the main differences between the SOA and MSA architectural styles. In examining each bullet we find that for the most part the frame of a microservice will in fact align with that of a service of the SOA architectural style, with the exception of how much of the business process it encapsulates, as many business processes contain many services in order to complete the task. In an MSA this would be a conflict in purpose. This implies that MSA is really a subset or special architectural form of SOA. MSA provides an approach to delivering SOA in an effective manner for the right set of business drivers.

[3] See www.opengroup.org/soa/source-book/soa/soa.htm.

## *Microservices Architecture*

The following section provides a side-by-side comparison of the key characteristics of SOA and MSA and provides guidelines for when each architecture would be most appropriate.

### Question 1: Vision and Intent Comparison

*Compare and contrast the vision that epitomizes SOA and MSA. What needs, drivers, pain points, and gaps in the industry led to the evolution of the SOA and MSA paradigms? What were the leading thoughts and views that drove the respective evolutions?*

SOA began as an attempt to control the costs of distributed computing by leveraging infrastructure that was originally intended for the web. SOA also emerged in part as an approach to combat the challenges of the large monolithic applications. It was envisaged as a way to "service-orient" business functions to be re-usable across an enterprise, decoupling the complex systems in the enterprise landscape. It is an approach that aims to promote the re-usability of software; two or more end-user applications, for example, could both use the same services. It aims to make it easier to maintain or rewrite software, as theoretically we can replace one service with another without anyone knowing, as long as the semantics of the service remain the same.

The primary motivation of SOA was to increase agility by allowing the underlying IT infrastructure and architecture to change in response to changing business needs. The notion of SOA started with web services and evolved into a more general SOA paradigm where a service provider exposed a set of business-aligned services to service consumers who did not have to be concerned with implementation or technological details.

Some by-products were the ability to share commonly recurring services, to re-factor functionality into common services and their variations.

Precursors of SOA, such as distributed object technology, could be complex.

Initial attempts at SOA were simple, layered on top of rising technologies such as the Hypertext Transfer Protocol (HTTP) and the eXtensible Mark-up Language (XML). Technologies such as XML-RPC or XML over HTTP provided a simple synchronous Remote Procedure Call (RPC) capability that was easily implemented on top of the web technologies then in use. While this led to a simple means of providing basic web services as RPC invocations, it required extension to effectively handle more complex problems. This additional complexity led to technologies such as the Simple Object Access Protocol (SOAP), which broke the dependency upon specific lower-level protocols. As systems became more complex, patterns such as the Enterprise Service Bus (ESB) evolved to address the issues related to the growth and complexity of SOA. However, despite many efforts, effective implementation of SOA remains a challenging problem. The mixture of technologies and the need to understand how the myriad of technology solutions relate to the underlying business problems has often made environments that leverage SOA large and complex requiring highly skilled practitioners. This has led to a certain dependence upon the narrative set out by various vendors in this space.

MSA developed as a push back against this complexity. It has emerged from the lessons learned in real-world use. The idea is to focus on the single business function and create services that implement the operations required by that function. (A key part of developing an MSA is to align with the single responsibility principle, where a service has responsibility for a single part of the functionality provided by the software. This requires strict partitioning of functionality to ensure that inadvertent coupling between functions does not occur.) The MSA paradigm is a variant (subset) of SOA, where focus has been placed on the runtime

autonomy of each service, stressing the independence (of both design and runtime) as well as resilience of each service. Deployed instances of the services are also independent of each other and of other services. MSAs are focused on rapidly developed and deployed services that do not require an extensive overhead, and can be quickly updated and replaced independently. Ultimately, an MSA is an SOA with this independence constraint.

Microservices are a means of implementing an SOA in which individual teams can choose their often differing underlying technologies and deployment schedules. Since functional areas – *aka* business units or Lines of Business (LOBs) or silos – can rapidly be developed without significant dependencies on other functional areas, MSA allows the fine-grained services to be built first for the silos. The need for integration of services or orchestration then enters into the picture and microservices are not necessarily a right fit for these cross-cutting cross-domain applications.

## Question 2: Entry Criteria/Applicability for Using One Style

*Compare and contrast the technical and business conditions and decisive factors that would drive the choice between implementing an MSA or SOA as the solution to a business need. Factors will include deployment schedule, service consumer environment, and development cycles. Provide usage guidelines to aid the decision-making process.*

Microservices should be considered for implementing functionality that needs to be resilient at an operational level, that can be encapsulated as a component independent of all other components of an overall system, and that may be subject to relatively frequent change and releases due to business conditions. Although there are no hard and fast rules forcing the choice of an architecture to meet a particular business need, answers to the following table of questions provides guidance indicating situations where an MSA may be the best choice as opposed to a conventional SOA implementation.

| Question | MSA Candidate | SOA Candidate |
|---|---|---|
| Can the solution be broken up into a collection of individual and independent service components? | MSA design revolves around the concept of independent services. | Independence of services is one of the key tenets of MSA; if services cannot be independent, MSA is not a good choice. |
| Does the solution require resilience and elastic scalability? | An MSA provides resilience and scalability through parallelism. | For some applications, another approach to achieve these ends (e.g., "big iron") may be more cost-effective. |
| Will the solution need to use orchestration across services to implement functionality? | "Smart endpoints and dumb pipes" – MSA relies on a Representational State Transfer (REST)-like philosophy for low-level choreography. | If more complex protocols, such as Business Process Execution Language (BPEL) or Web Service Choreography (WS-Choreography) are required, or a central tool, MSA is not a good choice. |
| Can the service API be defined atomically in respect to other business resources? | If yes, this atomicity is likely to foster the service independence which is key to an MSA. | Loose or decoupled services are desirable for an SOA; not otherwise a driver. |

### *Microservices Architecture*

| Question | MSA Candidate | SOA Candidate |
|---|---|---|
| Is there expected to be a high rate of churn or update of individual services? | If the API definition, service structure, and operational requirements are not stable, MSA may be a better candidate. | High rates of service update will likely be more complicated to support in a traditional SOA infrastructure. |
| Is this a new application (or business area) for the enterprise, or is this building on top of a legacy application? | | |
| Is this a new application without an existing installed base? | New developments represent opportunities to design from the ground up; MSA may represent the most cost-effective solution. | Replacing an existing (possibly monolithic) application requires a business case evaluation. |
| Is the application development/deployment cycle short? | An MSA solution offers low overhead and rapid parallel development. | A more complex full SOA may take longer to develop and deploy. |
| Does the enterprise already have an existing SOA stack and a heavy technical investment in SOA-based products (ESB, Business Process Management (BPM), etc.)? | | |
| What are the maintenance costs of the existing infrastructure? | High operating and maintenance costs may open the door for a re-architecting to an MSA with substantial cost savings over the project lifetime. | If operating costs are not a driver, the re-architecting effort may not be worthwhile. |
| Does the enterprise have a substantial base of domain expertise invested in the current design *versus* purchased (vendor) SMEs? | Trading contract expertise for the less complicated development needed for MSA may be economical in the long run. | High-level in-house expertise may be less expensive overall over the long term than re-working the entire infrastructure. |
| At what level is service governance managed? | MSA emphasizes local, distributed governance to avoid the need for coordination and the penalties of centralization. | Centralized governance is compatible with SOA. |

## Question 3: Business Drivers

*From a business (not technical) perspective, identify decision points that might lead one to choose SOA or MSA, as the implementation architecture.*

| Question | MSA Candidate | SOA Candidate |
|---|---|---|
| Can the organization support multiple development teams working simultaneously and independently? | If yes, fits well with the distributed development nature of an MSA. | If no, MSA development will be serialized, reducing its attractiveness. |

| Question | MSA Candidate | SOA Candidate |
|---|---|---|
| Does the business already have an SOA infrastructure in place? | If not, MSA can be an attractive choice. | If so, the decision to use an MSA will be driven by other factors. |
| Is resilience of the functionality provided by the component in question paramount? | MSA offers an inexpensive means of providing resilience for appropriate applications. | Resilience may be more expensive to provide as it is application-dependent. |
| What is the deployment schedule and what are the drivers for the service update schedule? | Shorter cycles favor an MSA. | Longer development/deployment cycles will not disadvantage SOA. |
| Is this a mature business area where the enterprise is dominant? | Cost of re-architecting is not necessarily worthwhile. | Existing infrastructure and expertise support SOA. |
| Is this a new solution, an exploratory, or niche business? | The low cost of MSA and rapid development and deployment time favor it. | It may be hard to achieve cost and time-to-market targets with more complex architecture. |

## Question 4: Characteristics Comparison

*Compare and contrast the defining and supporting characteristics of MSA and SOA as documented by the Work Group.*

| MSA | SOA |
|---|---|
| In an MSA, a service has to be independent of other services. | In an SOA, there is no requirement for independence. |
| In an MSA, parallelism and architectural resilience and scalability are achieved through this independence. | In an SOA, there is freedom to select how to achieve these goals. |
| MSA is constrained by its focus on single responsibility per service. | An SOA is free to support multiple responsibilities within a service. |
| Within an MSA, services cannot be comprised of other services due to the independence requirement. | Full SOA allows services to be built through orchestrations and choreographies. |

## Question 5: Architecture Paradigm and Style Comparison

*Compare and contrast the two architectural styles.*

MSA is a style of architecture that defines and creates systems through the use of small independent and self-contained services aligned closely with business activities. MSA is a subset of full SOA with the added constraints of service independence.

## Question 6: Architectural Principles

*Compare and contrast the architectural principles of SOA and MSA. Per The Open Group (the TOGAF 9.1 standard, Section* 23.2)*: "Architecture principles define the underlying general rules and guidelines for the*

*use and deployment of all resources and assets across the enterprise. They reflect a level of consensus among the various elements of the enterprise, and form the basis for making future decisions. Each architecture principle should be clearly related back to the business objectives and key architecture drivers."*

Since MSA is a subset of SOA, the basic architectural principles for MSA and SOA are identical. Particularly, both architectures share the key principles of independence:

- *Location-independence*: There is no preferred location for service consumers and service providers. They could transparently both be located on the same system, or in different organizations and in different physical locations.
- *Implementation-independence*: There is no requirement for specific platform or implementation technologies for service consumers and service providers to adopt. They should not need to be aware of the other party's technical environment or implementation details in order to interoperate.
- *Protocol-independence*: From an architecture perspective, the SOA or MSA can be constructed using any available protocols, but any specific implementation may choose to support a limited set of transport and message protocols.

Another key shared principle is that of *self-contained services*. The principle of *self-containment* is achieved when a service can be invoked with only the information available in its description. The service consumer should be isolated from the implementation details of the service. Self-contained services are encapsulated and do not depend on other services for their state, or are stateless.

The distinguishing characteristic of an MSA is that of *service independence*. An MSA service is independent of all other services. Independence of services enables rapid service development and deployment, and permits scalability through instantiation of parallel services. This characteristic also provides resiliency to the MSA; services are expected to fail and their responsibilities taken over by parallel instantiations.

By contrast, the corresponding principle for a full SOA is *loosely-coupled services*, where service consumption is insulated from underlying implementation. This insulation minimizes changes to consumers of services over time, even when versions change or changes are needed for qualities of service or protocol support. It may use an integration layer to provide support for connections, protocol mediation, security, and other qualities of service.

# CASE STUDY: MSA for a Hotel Central Reservation System

## Introduction

This case study is based on a composite of customers in the hotel industry. The specific scenario is a subset of common scenarios encountered within this industry. The mythical business represented here is a mid-size chain of hotels with 50 owned properties, 30 managed properties, and 30 franchised properties. We will call this company One Ten Hotels or OTH. This case study discusses the use of an MSA and how it meets the needs of the scenarios discussed.

## Business Scenario

OTH is a mid-size global chain of hotels that has grown large enough that it sees a potential business advantage in creating its own Central Reservation System (CRS). The CRS needs to be:

- Globally accessible
- Highly available
- Easily built and maintained
- Developed and deployed quickly

A simple CRS system needs to expose the following resources:

- *Hotel Information*: Description of a hotel and its rooms including text, photographs, and other media types. The basic form of this resource includes hotel address and contact information. The following resources are directly related to a specific hotel but may be treated as top-level resources or separate services:
    - *Hotel Inventory and Availability*: Vacancies or unsold rooms during any specific range of dates.
    - *Hotel Room Types*: Specific types of rooms; for example, King Bed or Double Queen. Some room types maybe virtual such as "Run of the House" which matches any available room.
    - *Stay Offers*: An offer to stay in a specific hotel and type of room for a specific period of time at a certain price. Stay offers are dependent on the underlying pricing model which in this case is not exposed.
- *Guest Profile* : Collected information about each guest including name, address, and phone number. May also provide access to loyalty information, stay history, and credit cards for room guarantees.
- *Reservations*: A contract between the hotel and a guest to provide services, normally a rental of a specific type of room in a specific hotel for a specific period of time at a specific price for some time in the future.

The typical work flow for a guest to create a reservation is:

- Locate and select a hotel available for the desired arrival and departure dates.

- Select an offer to stay in the hotel at a specific price in a particular type of room. This offer may also include meals or other add-ons.
- Identify the guest (this may already be done if the guest is logged into the system) and means of guaranteeing the reservation.
- Complete the reservation and issue a reservation number.

## MSA-Based Solution

The CRS requires high availability. Also, since demand for a reservation service varies widely depending on the time of day and the day of the week and month, implementing a dynamically scalable or elastic cloud service should reduce ongoing operational costs. It is also desired to use separate development teams to work in parallel on the core services supporting Hotel Information services, Guest Profile services, and Reservations services and their supporting applications simultaneously. This should accelerate development and delivery times.

An MSA-based solution was suggested to meet these needs. Both the Hotel Information and Guest Profile services can be launched without dependencies on any other components or services. However, a reservation binds the guest with the hotel (and type of room) and impacts the available inventory at the hotel for the period of time covered by the reservation. The application layer normally already has the references to both the guest and the hotel and can easily provide these for a *create reservation* operation on a reservations service. But when a reservation is created, inventory on the hotel must also be adjusted. Creation of a transaction with dependencies on other services is a violation of the principles behind an MSA.

There are a number of ways to work around the dependency between the Reservation services and the Hotel Information services. In this case, an operation was added to the Hotel Information services to reserve inventory. This operation returns a unique inventory ID and removes rooms from the inventory but does not assign rooms to the reservation. The application layer is required to retrieve this inventory ID from the hotel and provide it to the Reservations service when creating a reservation. If the create reservation operation fails, then the inventory can be returned to the hotel using the hotel's cancel inventory operation. An asynchronous clean-up operation can be run periodically between the inventory and reservation stores to make certain that all reserved inventory has a reservation and returning inventory with no reservations to the pool.

The create reservation operation (often implemented as a PUT or POST request from the HTTP protocol) accepts the reference to the guest profile of the primary guest, the hotel, and the inventory ID along with arrival and departure information, room types and offers, and other information relevant to the reservation. Since all of the information is provided by the application creating the reservation, the dependency between the Reservations services and the Hotel Information and Guest Profile services is broken and the Reservations services can be implemented as a microservice.

## Result

While this case study is presented in a generic fashion, it is based on a real architecture that is currently in the process of being implemented. The independence of the service areas has allowed multiple teams to be brought in to extend the services to areas beyond the ones discussed in this case study. Many services required to support a hotel ecosystem can be implemented independently of the other services in the

ecosystem. This allows for rapid development of new solutions. In this case, the rapid development addresses the time-to-market requirements of the organization developing the solution. The resulting MSA solution is also expected to meet the needs for rapidly scaling to accommodate the large number of customers and partners that are lined up to use the services.

## Conclusion

By placing the independence restriction on the SOA, in this solution the resulting MSA has enabled:

- Parallel and independent development and deployment of services
- Scalability through elasticity

The solution is also expected to be resilient through the elastic deployment of parallel service instances and agile since individual services can be changed to rapidly meet new requirements without impacting other services. While MSA may not be suitable for every situation, it is a compelling choice for problems that benefit from the independence constraint.

# CASE STUDY: Rainyday Grocer

This case study is a fictional MSA-based solution deployed in the cloud.

## Introduction

Rainyday Grocer is a cloud-based online grocer that provides customized delivery of groceries. Their business model is targeted towards those "rainy day" moments where a person needs groceries but is unable to go the bricks-and-mortar grocery store to shop.

## Business Scenario

Rainyday Grocer (RG) supports three different ordering options. A customer can:

1. Place an order with one of the approved grocery stores, send the item list, and order confirmation to RG via a web interface or a mobile application.
2. Send a grocery list to RG via a web interface or a mobile application.
3. Select from RG grocery items list and place an order.

There are two delivery options:

1. Doorstep delivery with text message confirming delivery.
2. Collect groceries order from one of the RG collection points.

RG does not own any inventory, supply channels, distribution channels, or data centers. They leverage other service providers for all services, and manage quality through a careful selection process and SLAs. They maintain a lean team of 50 people, only 5 of whom are IT-focused, to manage their operations across five states in the Eastern US.

The customer has to accept a set of constraints to place a successful order:

- The customer must have an active account in good standing (less than five *floods*; i.e., negative points).
- The customer must have a valid payment method registered.
- An order is limited to 10 or less items with a cumulative weight not more than 25lb.
- RG does not guarantee any specific brands.
- There is a four-hour window from order confirmation to requested delivery.
- An order cannot include medicines or hazardous materials.
- The delivery address must be a physical address.
- The delivery address must be within 50 miles of a city center.
- The customer must provide feedback within 24 hours, else get a flood.

## MSA-Based Solution

RG is building an MSA-based solution.

The recommended approach for an MSA-based solution is to drive it from the business needs and address the business processes.

This case study examines the MSA solution for the *receive order* sub-process of RG. The focus is on building blocks and to highlight the MSA characteristics and principles.

The RG business process is shown in Figure 4, in Business Process Modeling Notation (BPMN).

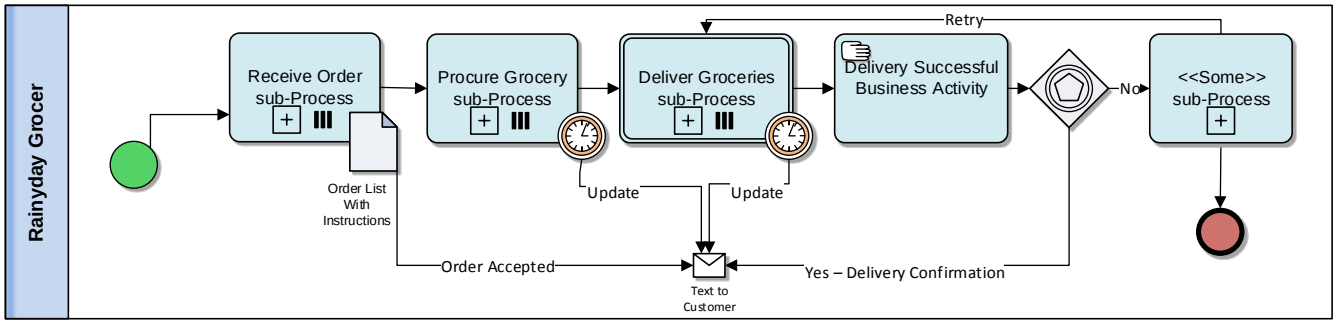



Figure 4: Order-Delivery Process

The receive order business process is decomposed as shown in Figure 5.

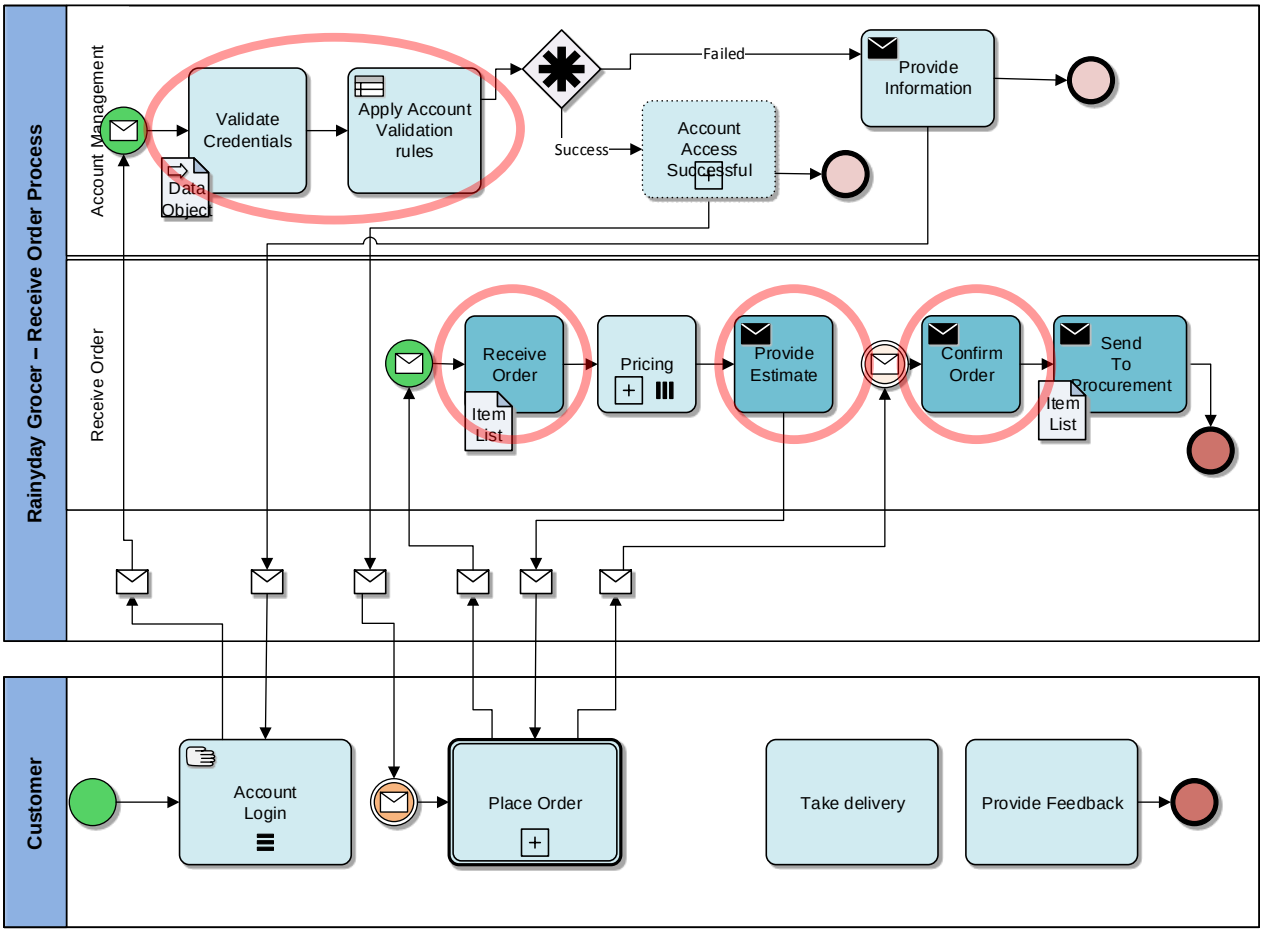



Figure 5: Receive Order Sub-Process Decomposed

The above business process shows the more granular activities. The ones circled in red represent *Atomic Business Activities* for RG.

Adhering to the single responsibility principle of MSA, RG wants each of the atomic business activities to be fulfilled by an *independent* and *self-contained* microservice.

As RG is a lean organization consisting of only five IT team members they realize they will need to outsource this development work.

Taking advantage of the *decentralized governance* and *polyglot development model* of MSA, RG identifies business owners for each of the business processes and with responsibility to create autonomous teams to address the needs of that business process. But to avoid total anarchy, they also institute a lean enterprise governance framework as depicted in Figure 6 and Figure 7.

### *Microservices Architecture*

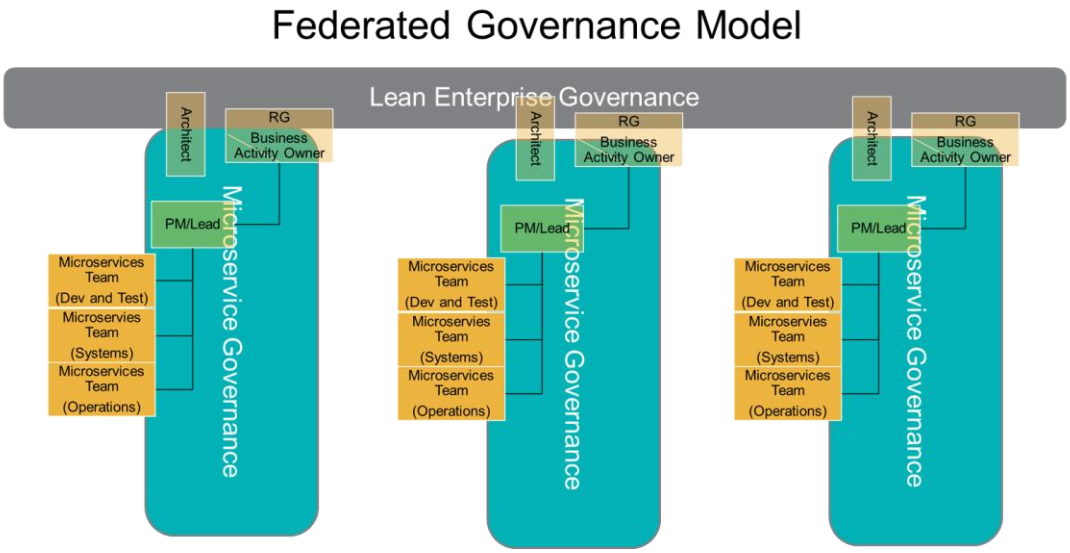



Figure 6: MSA Governance Framework at RG


Figure 7: MSA Governance

Consistent with the MSA principles, there is single team ownership of each microservice. (One team can own multiple microservices, but *vice versa* is not permitted.) Each team is autonomous – i.e., self-governs the entire lifecycle of each of its microservices – and is headed by the business activity owner. The business owner and the architects have a seat on the enterprise governance board, which ensures the enterprise governance model is correctly leveraged by each of the microservices teams.

The scope of the enterprise governance is kept minimal and non-intrusive to allow the autonomy of each of the microservices teams.

RG has standardized on an MSA Reference Architecture and a framework that each of the microservices teams will leverage.

### *Microservices Architecture*

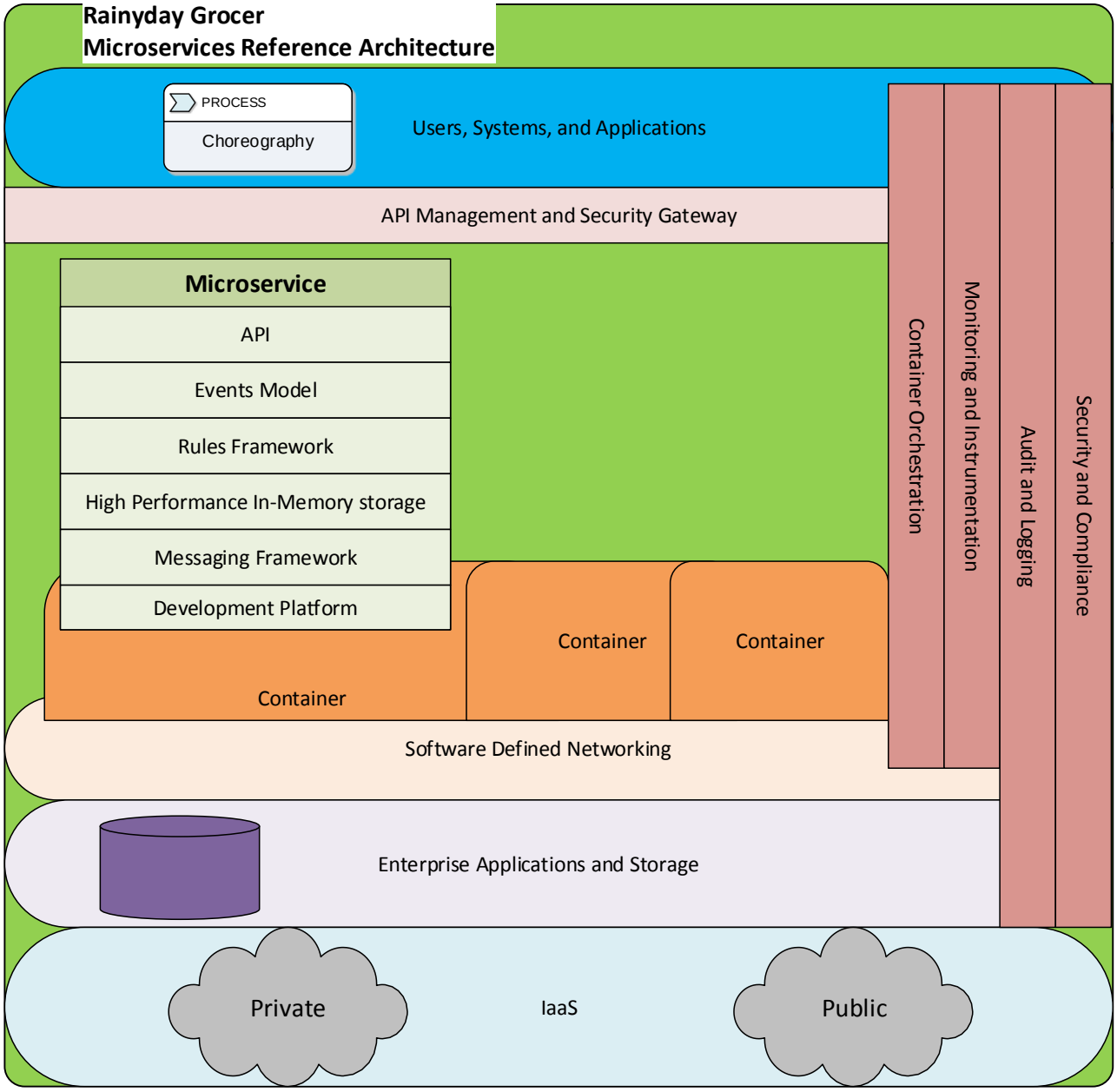



Figure 8: Microservices Reference Architecture at RG

Leveraging the above reference architecture, each of the teams will develop, deploy, and manage its respective microservices.

## *Microservices Architecture*

Here are some of the microservices that are implemented at RG. Consistent with the characteristics of MSA, each of these microservices performs a single business activity and does it well, each is independent, stateless, and spawned as multiple parallel instances to ensure high resiliency.

All of these microservices are choreographed by the application or triggered by the generated events.

#### *Account Login Microservice*

This microservice allows the user to log in. It validates the user's identification and security credentials, and ensures the account is valid based on a set of business rules.

The microservice API accepts an encrypted data object consisting of user credentials and account information. It queries the enterprise Identity and Access Management (IAM) solutions to validate the user credentials. It retrieves the user authentication token and account identifier.

Using the account identifier it retrieves the account history from the cloud-based CRM solution provider.

The rules are persisted in the in-memory storage of the microservice. It applies these rules to determine the account standing. If all validations pass, the service generates three tokens:

1. *User token* that has a Time To Live (TTL) of the *receive order* transaction
2. *Transaction token* that expires once the entire *order delivery* transaction is complete
3. *Service token* that is leveraged by the MSA framework to monitor and audit the transaction. On completion the service also generates a login success or failure event.

This service is leveraged by other business activity owners for their respective business processes.

#### *Receive Order Microservice*

This service is primarily focused on receiving and processing the order. It is triggered by the *successful login* event, and waits for the *order list data* object from the customer.

Upon receiving the *order list data* object, it parses, categorizes, and validates all the items on the list based on the business rules.

It tags this order with the *user token* and *transaction token* generated by the previous services and generates the following:

1. *Order success* (or *failure*) event
2. Service token that is leveraged by the MSA framework to monitor and audit the transaction
3. Data object that includes the items list and other procurement/processing instructions
4. Propagates the *user token* and *transaction token*

#### *Provide Estimate Microservice*

The RG business model is a no-inventory model, hence the price of goods is dependent on the current market price for those goods. Therefore, RG provides an estimate range. This service periodically queries RG's registered grocers for the current price list. It caches this information and is able to build and provide the estimate of prices for the item list published by the *receive order* microservice.

This microservice tags the estimates with the *user token* and *transaction token* generated by the previous services and generates the following:

1. *Estimate available* event
2. Service token that is leveraged by the MSA framework to monitor and audit the transaction
3. Data object that includes the prices for the items list with additional metadata
4. Propagates the *user token* and *transaction token*

#### *Confirm Order Microservice*

This microservice completes the order process and triggers the supply chain procure grocery sub-process. It sends a confirmation to the customer of order completion, with a confirmation ID and instruction on delivery collection of the order.

It tags the order complete event with the *user token* and *transaction token* generated by the previous services and generates the following:

1. *Order complete* event
2. Service token that is leveraged by the MSA framework to monitor and audit the transaction
3. Data object that includes the final items list with additional metadata
4. Propagates the *user token* and *transaction token*

## Results

RG's choice of MSA is well suited to their needs for the following reasons:

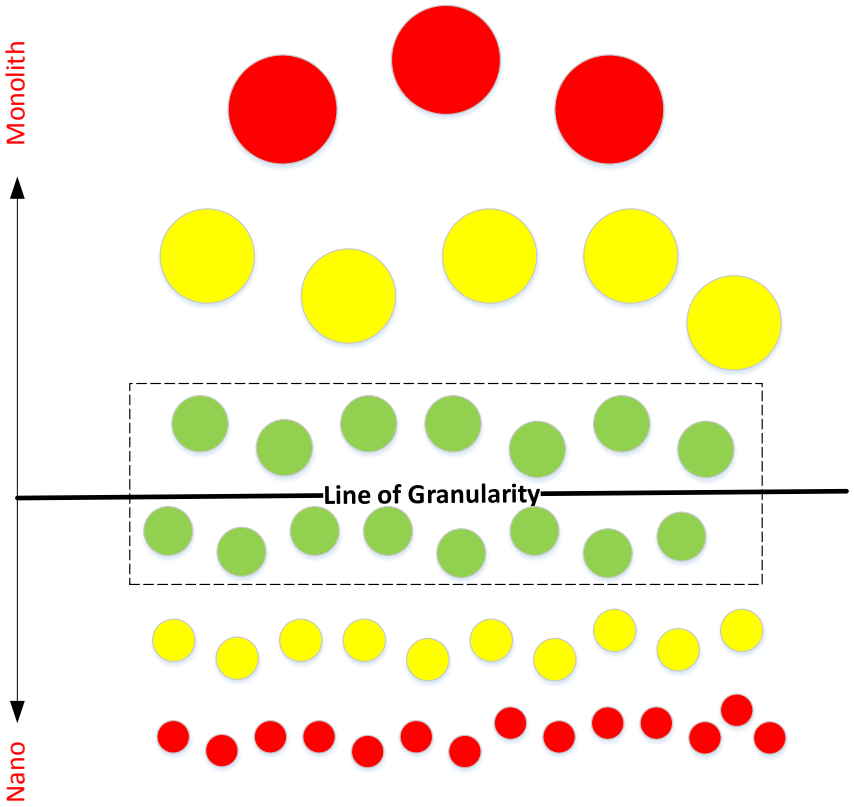
- It allows them to implement a complex process with a very lean team.
- They are able to quickly build teams with readily available skill sets to build each microservice.
- They can make changes on-the-go; i.e., CI and CD and DevOps enablement.
- Agility and flexibility.
- Minimal CapEx and optimal OpEx.

## Conclusion

This is a fictional case study that is deliberately exaggerated to highlight the core and auxiliary principles of MSA that are developed. The intent of this case study is to develop examples to add to the understanding of the microservices concepts and principles.

# APPENDIX A: Service Granularity

The various levels of service granularity and the positioning of microservices can be illustrated using the following diagram:




Assume there is an imaginary horizontal line (x-axis), representing the *line of granularity*; the right level determined for MSA. The services that fall closer, or around this line (green services within the dotted box), are good microservices; those that are way above this line trend towards exhibiting characteristics of monoliths, and those that fall way below the line trend towards exhibiting characteristics of nano-services. Many of the yellow services and all of the red services fall in either of these camps.

Problems with monoliths include:

- Even small, minor changes require rebuilding of the entire code base and re-deployment of the new build.
- Change cycles (for various functions and features) will have to be tied together, causing an undesirable dependency.
- Achieving modular structure within a monolith is hard to enforce.
- Scaling is achieved by replicating the entire application (though specific functions may have different scalability requirements).

Problems with nano-services include:

- Remote calls are expensive (from a performance perspective).
- Communication between services becomes chatty, resulting in a sub-optimal system.
- Unmanageable explosion of services can result in service proliferation, challenging governance.

# Glossary

***API***

An Application Programming Interface (API) is a set of functions, procedures, methods, or classes used by computer programs to request services from the operating system, software libraries, or other service, either locally or via a web-based application or service.

***Architecture***

Architecture is the structure of components, their inter-relationships, and the principles and guidelines governing their design and evolution over time.

***CapEx/OpEx***

Capital Expenses (CapEx) and Operating Expenses (OpeEx) are two different basic categories of business expenses. They differ in the nature of the expenses and their respective treatments for tax purposes.

***Cloud Computing***

Cloud computing is a model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort.

***Continuous Deployment (CD)***

Continuous Deployment (CD) can be thought of as an extension of Continuous Integration, aiming at minimizing lead time, the time elapsed between development writing one new line of code and this new code being used by live users, in production.

***Continuous Integration (CI)***

Continuous Integration (CI) is a development practice that requires developers to integrate code into a shared repository several times a day. Each check-in is then verified by an automated build, allowing teams to detect problems early.

***DevOps***

DevOps (a clipped compound of "development" and "operations") is a culture, movement, or practice that emphasizes the collaboration and communication of both software developers and other Information Technology (IT) professionals while automating the process of software delivery and infrastructure changes.

***Domain-Driven Design (DDD)***

Domain-Driven Design (DDD) provides a set of principles and methods to manage complexity by identifying core and ancillary domains and addressing Architecture, Development, Operations, Teams, etc. within the Bounded Context of each domain.

***Governance***

Architecture governance is the practice and orientation by which Enterprise Architectures and other architectures are managed and controlled at an enterprise-wide level.

## *Microservices Architecture*

### *Granularity*

Decomposition of the functions performed by a software application to determining the size of work performed by a single task.

### *Internet of Things (IoT)*

The Internet of Things (IoT) is the network of physical objects – devices, vehicles, buildings, and other items embedded with electronics, software, sensors, and network connectivity – that enables these objects to collect and exchange data.

### *Microservice*

An individual microservice is a service that is implemented with a single purpose, that is self-contained, and independent of other instances and services. A microservice is a primary architectural building block of a Microservices Architecture.

### *Microservices Architecture (MSA)*

Microservices Architecture (MSA) is a style of architecture that defines and creates systems through the use of small independent and self-contained services that align closely with business activities.

### *Monolithic Application*

A monolithic application describes a single-tiered software application in which the user interface and data access code are combined into a single program from a single platform. A monolithic application is self-contained, and independent from other computing applications.

### *Object-Oriented Design (OOD)*

Object-Oriented Design (OOD) is the application of object-oriented methodology to the design of computer systems or applications.

### *Resilience*

Application resilience is the ability of an application to react to problems in one of its components and still provide the best possible service.

### *SOLID*

SOLID is an acronym coined by Rob Martin distilling the first five principles of a class in OOD.

- SRP: The Single Responsibility Principle – a class should have one, and only one, reason to change.
- OCP: The Open Closed Principle – you should be able to extend the behavior of a class without modifying it.
- LSP: The Liskov Substitution Principle – derived classes must be substitutable for their base classes.
- ISP: The Interface Segregation Principle – make fine-grained interfaces that are client-specific.
- DIP: The Dependency Inversion Principle – depend on abstractions, not on concretions.

## *Microservices Architecture*

### *Scalability*

Scalability is the capability of a system, network, or process to handle a growing amount of work, or its potential to be enlarged in order to accommodate that growth.

### *Service-Oriented Architecture (SOA)*

A Service-Oriented Architecture (SOA) is an architectural pattern in computer software design in which application components provide services to other components via a communications protocol, typically over a network. The principles of service-orientation are independent of any vendor, product, or technology.

### *Web-Oriented Architecture (WOA)*

Web-Oriented Architecture (WOA) is a software architecture style that extends SOA to web-based applications.

# References

(Please note that the links below are good at the time of writing but cannot be guaranteed for the future.)


- Martin L. Abbott, Michael T. Fisher: The Art of Scalability – Scalable Web Architecture, Processes, and Organizations for the Modern Enterprise; refer to: http://theartofscalability.com.
- Blog: Adopting Microservices at Netflix – Lessons for Architectural Design; refer to: www.nginx.com/blog/microservices-at-netflix-architectural-best-practices.
- Blog: James Hughes on Micro Service Architecture; refer to: http://yobriefca.se/blog/2013/04/28/micro-service-architecture.
- Melvin E. Conway: Conway's Law: How Do Committees Invent?; refer to: www.melconway.com/research/committees.html.
- Eric Evans: Domain-Driven Design – Tackling Complexity in the Heart of Software, Addison-Wesley, 2003.
- Martin Fowler article on Microservices; refer to: http://martinfowler.com/articles/microservices.html.
- SOLID: Principles of Object-Oriented Design; refer to: http://butunclebob.com/ArticleS.UncleBob.PrinciplesOfOod.
- Sam Newman: Building Microservices – Designing Fine-Grained Systems, O-Reilly, 2015.
- The Open Group Service-Oriented Architecture (SOA), White Paper (W074), published by The Open Group, July 2007; refer to: www.opengroup.org/bookstore/catalog/w074.htm.
- The Open Group: SOA and Boundaryless Information Flow™; refer to: https://opengroup.org/soa/source-book/soa/soa_bif.htm.
- The Open Group: TOGAF® 9.1, an Open Group Standard, (G116), published by The Open Group, December 2011; refer to: www.opengroup.org/bookstore/catalog/g116.htm.

# About the Authors

### Somasundram Balakrushnan, Salesforce.com (MSA Project Co-Chair)

Somasundram Balakrushnan is a Senior Program Architect at Salesforce.com and a TOGAF 9 Certified Enterprise Architecture practitioner. He is an experienced Enterprise Architect and leader in SOA-based architecture development. He has led multiple teams in developing proof-of-concept architectures using microservices. His association with The Open Group includes: the SOA Work Group, Microservices Architecture (MSA), evolution of the TOGAF® standard, and Open Platform 3.0™. Som is leading the MSA Project in the capacity of Co-Chair.

### John Bell, Ajontech LLC

John T. Bell is the Founder and Principal Consultant of Ajontech LLC. He has over 35 years' experience in the Information Technology industry and 14 years supporting IT within hospitality-related companies. He chairs multiple working groups for the IEEE and the Hotel Technology Next Generation Consortium. John is also co-chair of The Open Group SOA Reference Architecture work.

### Benjamin Currier, Hewlett Packard Enterprise

Benjamin Currier is a Solutions Architect for Hewlett Packard Enterprise. He works primarily with customers in the Airline and Financial industries specializing in Event-Driven Architecture (EDA), Service-Oriented Architecture (SOA), and Business Process Management (BPM). He also works with a variety of customers interested in accelerating IT delivery through agile methodologies, DevOps practices, and cloud computing.

### Ed Harrington, Conexiam

Ed Harrington is an Enterprise Architect at Conexiam and has chaired multiple Open Group Forums and projects. He is a major author of the SOA Ontology, a Standard of The Open Group, and has been involved in The Open Group SOA Work Group since its inception. He has long been a proponent of the concept of Enterprise Architecture sustainability.

### Brian Helstrom, IBM

Brian Helstrom is a Senior Enterprise Architect at IBM. He has a long history of developing Enterprise Architectures for multiple clients. Brian spent more than seven years leading development and implementations of SOA and has developed MSA proof-of-concept architectures.

### Peter Maloney, Raytheon Company

Peter Maloney is a Senior Engineering Fellow at Raytheon Company. He became interested in Enterprise Architectures and particularly SOA as a result of the ever-expanding need for providing access to increasingly complex data products to a diverse group of end users, with the resulting needs for collaboration, throughput management, and security. He is a Raytheon Certified Architect, a program accredited by The Open Group, and a three-time winner of the Raytheon Excellence in Technology Award. He holds one patent and has authored more than a dozen papers.

### Ovace Mamnoon, Hewlett Packard Enterprise (MSA Project Co-Chair)

Ovace A. Mamnoon is an Engineer, an Enterprise Architect, and a Practice Principal at Hewlett Packard Enterprise. His near 20 years of varied work experience encompass areas that have a strong bearing on Microservices Architecture (MSA) including developing embedded systems, SmartGrid design and implementation, Service Oriented Architecture (SOA) and Integration, Internet of Things (IoT), Cloud Computing, Digital Enterprise, and others, all with a strong focus on Enterprise Architecture. Ovace is an active participant in The Open Group and is a Co-Chair of The Open Group MSA Project.

### Marcelo Martins, IBM

Marcelo Martins is an IBM Senior Certified and Open Group Distinguished Certified IT Architect, member of the IBM IT Architecture certification board, and member of the Canadian Academy of Technology Affiliate. In his 25 years of experience in IT, he has focused in enterprise, integration, and application architectures. He has worked with major customers in Canada and worldwide in the design and delivery of complex systems. Currently, Marcelo is a Client Technical Leader working with customers in the financial sector in Canada, helping them define technical strategies and adoption of emerging technologies. Marcelo has recently authored an IBM Redbook on microservices (Microservices from Theory to Practice) and continues to collaborate with the technical community within and outside IBM on furthering microservices understanding and adoption.

# Acknowledgements

The Open Group gratefully acknowledges the contribution of the authors, the reviewers:

- Ali Arsanjani, IBM
- Elizabeth Penisten, Raytheon Company
- Carlos Arturo Quiroga, IBM
- Sriram Sabesan, Conexiam

and the other members of the MSA Project team:

- Ahmed Abdel-Fattah, IBM
- Gautam Bhat, IBM
- Mike Broomhead, IBM
- Chris Harding, The Open Group (SOA Forum Director)
- Herman Hartman, Capgemini SA
- Ram Kodi, EA Principals Inc.
- Heather Kreger, IBM
- Satyajit Malavde, Hewlett-Packard Enterprise
- Lionel Mommeja, IBM
- Mrudul Palvankar, Cognizant Technology Solutions US Corporation
- Carlos Arturo Quiroga, IBM
- Vaidyanathan Ramaswamy, Hewlett Packard Enterprise
- Sarang Shah, Infosys Limited
- Mukund Srinivasan, Capgemini SA
- Ron Tolido, Capgemini SA
- Zhiguo (Antonin) Yang, Hewlett Packard Enterprise
- Shar Zand-Biglari, American Express

# About The Open Group

The Open Group is a global consortium that enables the achievement of business objectives through IT standards. With more than 500 member organizations, The Open Group has a diverse membership that spans all sectors of the IT community – customers, systems and solutions suppliers, tool vendors, integrators, and consultants, as well as academics and researchers – to:

- Capture, understand, and address current and emerging requirements, establish policies, and share best practices
- Facilitate interoperability, develop consensus, and evolve and integrate specifications and open source technologies
- Offer a comprehensive set of services to enhance the operational efficiency of consortia
- Operate the industry's premier certification service

Further information on The Open Group is available at www.opengroup.org.