A MICROSERVICES ARCHITECTURE

ADDRESSES PROBLEMS THAT MODERN ENTERPRISES FACE

LIKE APPLICATION SCALABILITY, TRAFFIC SPIKES, FAULT TOLERANCE, AND

ABILITY TO CHANGE.

MICROSERVICES PATTERNS HAVE BEEN ESTABLISHED TO ACHIEVE MODERN, AGILE APPLICATIONS.

MICROSERVICES:

PATTERNS FOR ENTERPRISE AGILITY AND SCALABILITY

KEYHOLE SOFTWARE WHITE PAPER

Introduction to the Microservices Software Architecture Style, Concepts, Recommended Patterns, And Suggested Adoption Approach

KEYHOLE SOFTWARE

WITH TOOLING SUGGESTIONS

Microservices: Patterns for Enterprise Agility and Scalability

A Keyhole Software White Paper

Introduction To The Microservices Software Architecture Style, Concepts, Recommended Patterns, And Suggested Adoption Approach

Table Of Contents

White Paper Introduction

Microservices is an architectural pattern gaining steam in the development community.

A Microservices architecture addresses problems that modern enterprises often face, including responding to market demands, handling spikes in traffic, and being tolerant to failure. These benefits are achieved by functionally decomposing a business' domain into *microservices*, services that handle only a single responsibility.

This aids agility by allowing teams to focus on a narrower domain, increasing scalability by giving smaller units of scale so additional instances of a service can be spun up in response to demand, and enhancing fault tolerance providing isolation units that can contain the scope of faults.

The Microservices approach originally evolved from web companies that needed to be able to handle millions of users with significant variance in traffic, while being able to also maintain the agility to respond to market demands. What those companies pioneered—technologies, design patterns, and operational platforms—have been shared with the open source community in an effort to help other organizations to adopt Microservices. While a formal standard does not yet exist, certain common characteristics define a Microservices architecture: independently deployable services, automated deployment, intelligence in the endpoints, and decentralized control of languages and data.

We have led many projects following this architectural pattern at Keyhole Software and results have been extremely positive. For many of our clients, this has become the default style for building enterprise solutions going forward.

We will discuss a number of topics related to Microservices in this white paper, with a particular emphasis on the enterprise. An overview of covered topics include:

- How Microservices Came To Be
- Contrasting Architecture Patterns
- Features Of A Microservices Architecture
- Established Patterns
- Getting Started With Microservices
- Suggestions For Microservices Adoption And Migration

How Microservices Came To Be

The Golden Age Of Object-Oriented Programming

The world of computing has changed since the introduction of object-oriented programming and modern computing paradigms. Corporations have been invaded by personal computers with tools for end users to improve their productivity (spreadsheets, word processing, etc.) IT departments have since striven to exploit this localized computing power by implementing client server-based applications that interacted with a server-based data store.

Object-oriented programming languages proved that the Graphical User Interface (GUI) realm could also be used to effectively implement, isolate, decouple, and reuse application logic. Initial OO languages, such as Smalltalk, introduced one of the first operating system-independent programming environments that enabled developers to shift their focus away from the many hardware-specific concerns that were previously necessary.

This new development thought process led to runtimes such as Java in 1995, and then the .NET framework and C# in 2000. While Java and C# have evolved with architecture styles over the years, many patterns and practices adopted have conceptually stayed steady.

These practices, patterns, and runtimes all had one common thing in their evolution: setting the bar of abstraction higher and the barrier to entry lower. Separating the need for the end developer to have to worry about minutia, repetitive tasks, and plumbing; thus allowing them to focus more squarely on delivering business value.

Interestingly, distributed computing (which has the same potential and motivation as Microservices) started to gain traction in the 1990s. Frameworks such as CORBA, DCOM, and later web EJB and SOAP-based services were solutions for scalability and reuse. However, the complexity and coupling of these technologies arguably limited this potential.

The Web

The web caused a major shift in application architecture: from the client/server style to the web application. No longer would applications need to be deployed across user bases; anyone with a browser and internet connection could now access applications.

From this shift, web application architectures that dynamically generated HTML on the server for user interfaces were pioneered in many technology stacks. Microsoft .NET and Java became popular, especially in the enterprise.

The Birth Of Agile

When it came to patterns and practices for software development, our industry had made excellent strides. However, we had not kept up the same pace for advancing and abstracting our software architectures around product development and the software development lifecycle (SDLC).

The patterns around SDLC that existed (waterfall, big-bang, spiral, etc.) were considered by many as overly-bureaucratic, regimented, and not in alignment with developers' newfound ability to more rapidly execute on tasks. Developers were now outpacing the business on building functionality. Most life cycle time was spent on building top-heavy business requirements documents and products that were not focused on value. They would then code these requirements into a monolithic single deployable application, which was then passed along for a monolithic testing effort.

In 2001, a group of thought leaders sought to create a guideline for how to think about SDLC. This document came to be known as the Agile Manifesto.[1]

Previous approaches emphasized up-front planning and detailed requirements from the beginning. Software projects could go off-track if the initial requirements phase made incorrect conclusions, with the delivered software not being what was initially wanted or needed. Agile attempted to address this shortcoming by implementing a rapid iterative cycle of review and planning. Bringing in project stakeholders to review the progress of the project gives an opportunity to provide input on the software that is being delivered, ensuring it aligns with their wants and needs. The planning process allowed project managers to incorporate any input from the stakeholder review as well as helped to ensure the most critical tasks were being worked on.

Businesses, especially enterprise organizations, were initially reluctant to this type of thinking and abstraction of business processes. Eventually it took hold and we saw a veritable arms race by businesses that were in an increasingly competitive landscape as compared to before the "dot-com bubble" collapse. There was more demand, fewer resources, and so there was an increased focus on value propositions and rapid iterations.

As opposed to waiting years for a product to come out, with Agile, you would see more focused, streamlined products coming out in quarterly or biannual releases. Feature enhancements would be released even more frequently. Usable code would actually be available much earlier than the release date.

[1] **Manifesto For Agile Software Development**. http://agilemanifesto.org/

The Perfect Storm

With business, engineers, and operations collaborating, more code went out more rapidly, and with a higher quality than before. We saw more code released in the last 5-10 years than we had seen in the previous 30 years of software development.

Now much of that code is starting to show its age. Maybe we were using good programming patterns eight years ago, but used bad business patterns until two years ago. Or we started out as an Agile shop but were not great about adhering to best practice development standards.

There are countless possibilities for this theme. The bottom line is that you have a lot of assets—some more modern, some less—all providing business value. All need to be maintained while they continue to add business value to your organization.

Many organizations struggle with how to coordinate their resources and spread them out evenly. As an example, let's say you have between two and five agile teams developing different parts of a product or different products entirely.

- Where do those projects land?
- What team owns the hardware?
- How do you make sure that mission-critical applications maintain high availability even if they weren't originally designed to be scaled horizontally or with any degree of fault tolerance?
- What happens if a machine dies? Do you keep moving forward, or do you lose some percent of your accounts and earn a negative reputation?

This 'perfect storm' has caused the development community, particularly in the enterprise, to look to how to modernize applications in a way that won't hurt the business. Though the code is aging, the business side still relies on those applications to run the organization.

The goal becomes to modernize, and to modernize in such a way that promotes agility, code quality, and is less resistant to change so they don't find themselves in the same position ten years down the road.

Contrasting Architecture Patterns

Monolithic Applications

Monolithic-based architectures have been a common approach for the enterprise. In a monolith, all application functionality is placed in a single deployable unit or

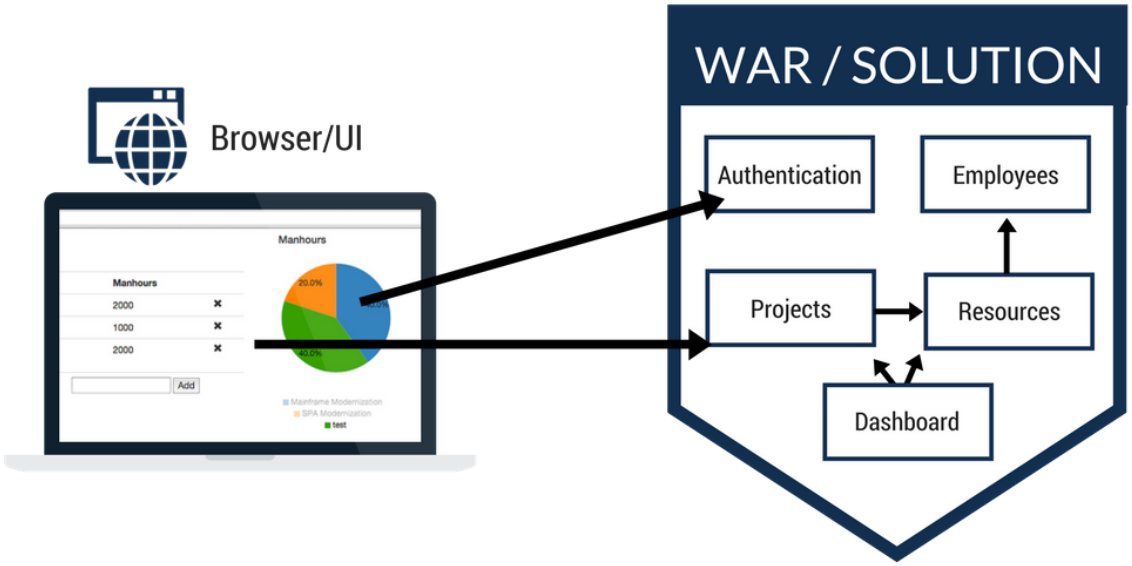
executable—such as a WAR/EAR or a solution package. You are unable to horizontally scale specific, individual components without scaling the entire application.

The problem here is that with applications of any size, changes made to any single feature of the application often need to be tested extensively for unwanted side effects. All change cycles usually end up being tied to one another.

Also, once in production, there is a dance that occurs between fixing defects and adding new features. What does a team do, when in the midst of a development sprint production, defect remedies have to be applied? Sometimes they can be nestled into the current build, but other times new features have to be backed out so that production issues can make their way into the next production build. This compromises efficiency and negatively affects new development timelines.

This approach can also be error-prone. With code flowing in and out, the chance for collisions or accidental inclusions/removals make this a high-stakes process. Additionally, as the application grows, development times of new and needed features get longer, and the QA burden gets heavier as the test surface area increases. Finally, a monolithic application is usually built with a single technology, therefore introducing new technologies can be an all-or-none proposition.

For reference, a conceptual picture of a Monolithic architecture follows as “Gallery 1.” It depicts a very simple project management application that manages resources for projects.



Gallery 1: Monolithic Application Architecture Concept

Tel 877-521-7769 | keyholesoftware.com

SOA / Distributed Computing

Distributed computing technologies have been around for a long time. These are mostly heavyweight protocols that require specific server implementations and expertise, and have complexity in use; think CORBA, EJBs, or WCF. Distributed computing frameworks are not implicitly easy to use and require supporting server software and infrastructure.

Service-Oriented Architecture (SOA) gained momentum starting in the mid-2000s. While conceptually valid and similar to Microservices, SOA's focus was on modeling coarse-grained services that had the ability to be reused across the enterprise. A plethora of expensive middleware software (ESBs, Orchestration, Asset Repositories) made up the SOA ecosystem. Heavy business resources (i.e. humans) were required to analyze the entire business to extract these services. These required a rigid governance model to ensure implementation and usage, which produced another organization silo.

There are organizations that have had success with SOA, but our experience says that this came with a very heavy investment in software and technical personnel, and took a long time. Martin Fowler makes a nice distinction between Microservices and SOA: SOA has smarts built into the wiring between services, while Microservices' wiring or connectivity is dumb and the smarts are found in each service.[2]

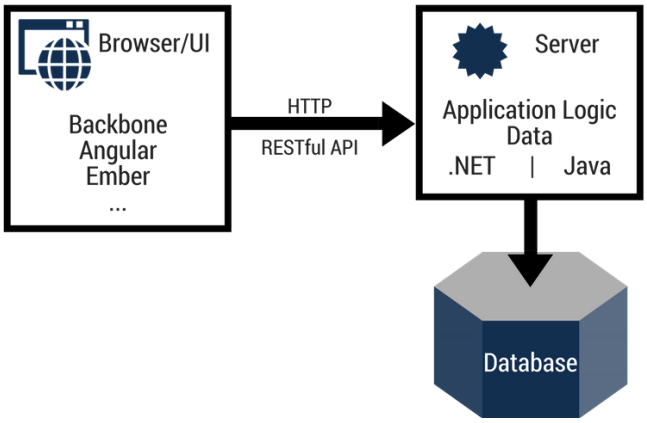
Single-Page Applications (SPA)

The single-page application architectural approach gained popularity in recent years. A single-page application (SPA) is a web application or website that fits on a single web page with the goal to provide a user experience similar to that of a desktop application.

Specifically, SPAs provide an improved, responsive user experience that can be mobile device friendly and bandwidth efficient by rendering all dynamic HTML and elements for a user interface in the browser. Additionally, HTML5-based SPAs provide a solution for the soon-to-be-extinct, browser-based plug-in technologies like Applets, Flex/Flash, and Silverlight.

Look at the example in "Gallery 2." In this configuration, server-side logic provides a RESTful API layer into application logic and data access.

[2] **Microservices.** Fowler, Martin. (2014, March 24). http://martinfowler.com/articles/microservices.html



Gallery 2: Single-Page Application Configuration Concept

Architecturally, SPAs decouple the user interface entirely from the server-side API. In a scenario of change, if the browser and HTML were to be replaced by some new technology, then the server-side API layer would continue to be valid and would serve this new user interface technology. Browser HTML rendering engines and JavaScript interpreters' efficiencies have made SPA technology shine.

Moreover, this physically-decoupled API style of architecture is a move toward the microservice style of architecture the next section will introduce.

Microservices

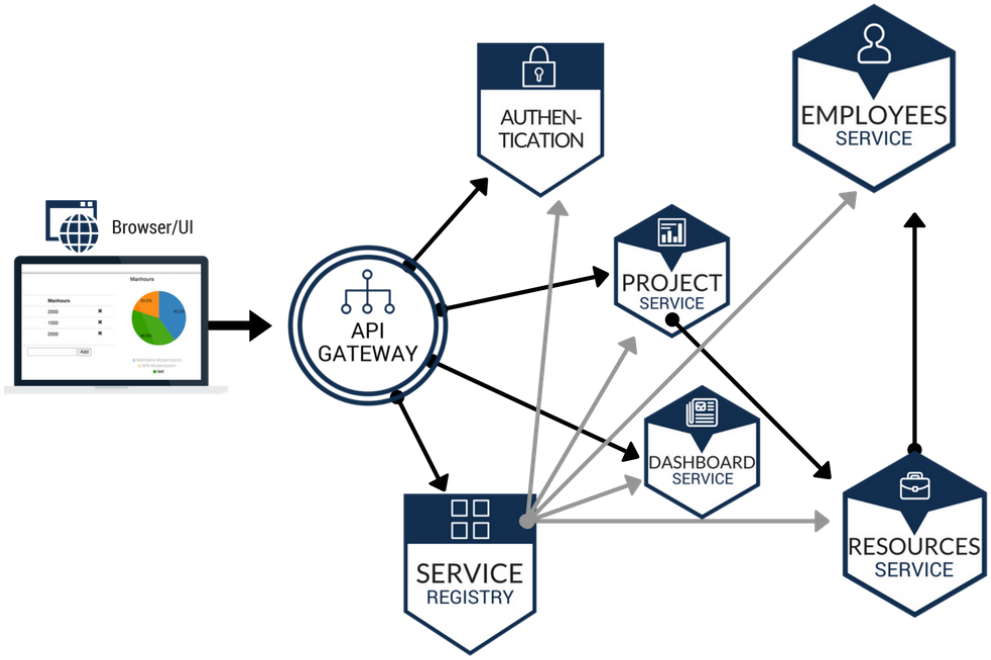
Introduction

There is no standard, formalized definition of Microservices. However, there are certain characteristics that help us to identify the increasingly-popular architecture style. At its core, Microservices is a method of developing software applications as a suite of independently-deployable, modular services. Each service is configured to run as a unique process and communicates through a well-defined, lightweight mechanism to serve a business goal.

The communication protocol and format is flexible and can be manifested in many forms depending on application requirements. A common communication approach is via HTTP/RESTful APIs with a lightweight, simple data transport protocol such as JSON.

Compare the Monolithic application architecture shown in "Gallery 1" to "Gallery 3," which shows that same monolith broken up into a Microservices style of architecture.

Tel 877-521-7769 | keyholesoftware.com



Gallery 3: Microservices Application Architecture Concept

From this example we can very quickly see that there are many more moving parts in a Microservices architecture. The "Gallery 3" image certainly looks more complicated as when compared with "Gallery 1."

Some of the hurdles that Microservices help to overcome include deployment, upgrading, failover, health checks/monitoring, discovery, and state management. Using Microservices versus a monolithic approach addresses specific issues that are inherent in monolithic applications. The benefits from this approach often outweigh any complexity that is introduced.

Features Of A Microservices Architecture

Smaller, Independently-Deployable Services

Microservices architecture breaks up application function into smaller independent units that are accessed and discovered at runtime, whether over HTTP or a IP/Socket protocol using RESTful APIs.

As Martin Fowler says[3]:

- "These services are built around business capabilities and independently deployable by fully automated deployment machinery. There is a bare minimum of centralized

[3] **Microservices.** Fowler, Martin. (2014, March 24). http://martinfowler.com/articles/microservices.html

management of these services, which may be written in different programming languages and use different data storage technologies."

These smaller services are independently deployable and scalable. Each service also provides a firm module boundary, allowing for different services to even be written in different programming languages.

- *Note:* just because you can write each service in a different programming language doesn't mean that is the best path! Often this depends more on your resource pool and the actual problems you are trying to solve.

Services are activities that an application performs. Ideally, each service will map to a single business function.

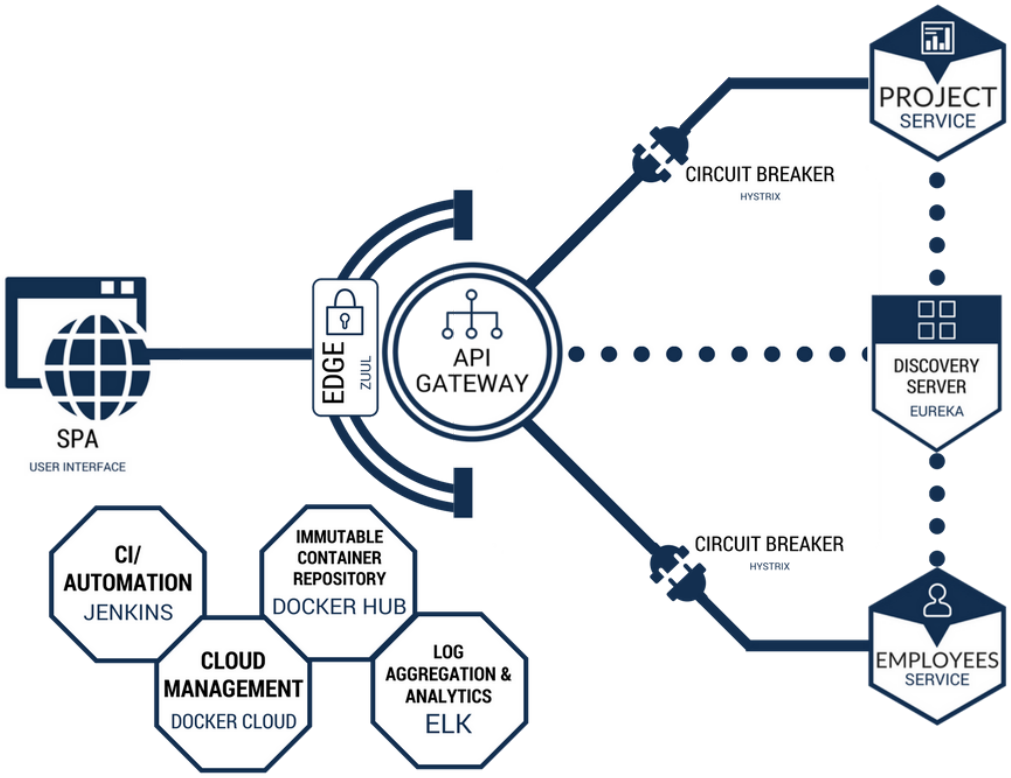
- An example of a microservice is maintaining a list of users and their permissions. In this example, this functionality would be done by a User Service that executes those actions to maintain the list (CRUD). These smaller, decoupled services focus on one activity in contrast to monolithic applications that perform all activities for an entire application in a single, tightly coupled code base.

Services should have a small granularity and the protocols should be lightweight. This creates a smaller surface area of change, making it easier to add functions and features to the system at any time. As a result of this smaller surface area, instead of having to redeploy entire applications as you might have with a monolithic application, you might only need to deploy one or more distinct services.

Established Patterns

There is no need to invent many common Microservices patterns in-house. Multiple companies that have had success with this style of architecture have released patterns and frameworks that support Microservices architecture.

A diagram of Microservice patterns is shown in "Gallery 4."



Gallery 4: Common Microservices Patterns

A quick description of some of these common patterns follows.

- ***Service Registry*** - A service registry communicates when and if a service is available for use, and determines if a service is alive. The Service Registry contains the current location (server:port) of all instances of a deployed service.

- ***Edge Controller*** - Provides routing, load balancing and authentication mechanisms from user interface and integration clients into API gateways. Typically, a single Edge Controller can route requests to multiple API gateways.

- ***API Gateway*** - A server that is the single entry point into a system of integration points or application. Serves as a gateway for an application user interface into application services. Service calls can be aggregated here to serve specific user interface functions, or simply passed through in order to avoid having to use CORS.

- ***Circuit Breaker*** - As there could be many layers of service calls involved in an application, a failure of in one of them could cause a cascade of errors up the calling routes. The circuit breaker pattern captures failures and defines a threshold of failures within a specified time, and opens the circuit. This immediately reports an error

instead of attempting to repeatedly keep accessing the service. See Keyhole blog Hystrix To Prevent Hysterix[4].

- ***Fallback Method / Chain Of Responsibility Pattern*** - Gives more than one object an opportunity to handle a specific request by linking receiving objects together. Each processing object contains logic that defines the types of command objects that it can handle; the rest are passed to the next processing object in the chain. A mechanism also exists for adding new processing objects to the end of this chain.

- ***Command Pattern*** - A behavioral design pattern in which an object is used to encapsulate all information needed to perform an action or trigger an event at a later time. For example, wrapping each call to a downstream dependency in a command and isolating it in its own execution thread.

- ***Bounded Context Pattern*** - The bounded context concept promotes an object-model-first approach to a service, defining a data model that a service is responsible for and is "bound to." In other words, the service owns this data and is responsible for its integrity and mutability. This supports the important Microservices features of independence and decoupling. See Keyhole Software blog Implementing a Bounded Context.[5]

- ***Failure as a Use Case*** - A Microservice platform has many moving parts and software processes that make up the system. If the platform is not durable and fault tolerant in an automated way, teams will spend most of their time troubleshooting and putting out fires. With failure as a use case, failure is intentionally introduced to the system to test durability and automated recovery. If failures are happening without "all hands on deck" and defcon levels being escalated, then you can assume a stable system. See Keyhole Labs blog Failure As A Use Case.[6]

Highlight: Service Registry / Service Discovery

Microservices are constantly changing; services go up and down as development teams make changes, instances are added in response to demand, or outages occur for unforeseen reasons. In order to handle this dynamic environment, your Microservices architecture needs a robust service discovery solution.

A Service Registry is a store of available service instances. If a service is added or removed, the Service Registry needs to know about the change. Through a heartbeat mechanism, the

[4] **Hystrix To Prevent Hysterix.** Keyhole Software Blog. Monson, Dallas. (2016, February 21). https://keyholesoftware.com/2016/02/01/hystrix-to-prevent-hysterix

[5] **Implementing A Bounded Context.** Keyhole Software Blog. Pitt, David. (2016, March 21). https://keyholesoftware.com/2016/03/21/implementing-a-bounded-context

[6] **Failure As A Use Case**. Keyhole Labs Blog. Pitt, David. (2017, January 17). https://keyholelabs.com/2017/01/17/failure-as-a-use-case

Service Registry determines if a service instance is still alive. The Service Registry contains the current location (server:port) of all instances of a deployed service.

- Eureka, built by Netflix, provides a REST API for managing service-instance registration and for querying available instances. It is often used in the AWS cloud for locating services for the purpose of load balancing and failover of middle-tier servers. It provides more sophistication in registering and de-registering servers with load balancing on the fly.[7]

There are two main type of service registration: self-service registration and third-party registration. Following the **Self-Service Registration Pattern**, when a service goes up, the network location of a service instance is registered with the service registry. When a service goes down or terminates, it is removed from the service registry.[8]

Third-party registration by comparison, leverages the aptly-named **Third-Party Registration Pattern**. It is in this pattern that a process or service manages all the other services, checking in on which microservice instances are running and automatically updating the service registry on behalf of the services.[9]

Highlight: Edge Controller

The Edge Controller provides routing, load balancing and authentication mechanisms. Typically a single Edge Controller can route requests to multiple API Gateways. Routing rules can be defined that "route" a sampling of users to a new API gateway for time period to help validate stability.

- Zuul, built by Netflix, is an edge service that provides dynamic routing, monitoring, resiliency, and security.[10] Zuul has a series of filters that are capable of performing a range of actions during the routing of HTTP requests and responses. It provides a framework to dynamically read, compile, and run these filters. Zuul supports any JVM-based language. The source code for each filter is written to a specified set of directories on the Zuul server that are periodically polled for changes. Updated filters are read from disk, dynamically compiled into the running server, and are invoked by Zuul for each subsequent request.[11]

The Edge Controller provides a place to monitor and condition request traffic. An example would be to apply a correlation ID to request traffic in order to provide log output that can be correlated to a single user request.

---

[7] **Eureka.** https://github.com/Netflix/eureka
[8] **Pattern: Self Registration.** Microservices.io. http://microservices.io/patterns/self-registration.html
[9] **Pattern: 3rd Party Registration**. Microservices.io. http://microservices.io/patterns/3rd-party-registration.html
[10] **Zuul**. https://github.com/Netflix/zuul/wiki
[11] **Announcing Zuul: Edge Service in the Cloud**. The Netflix Tech Blog. (2013, June 12). http://techblog.netflix.com/2013/06/announcing-zuul-edge-service-in-cloud.html

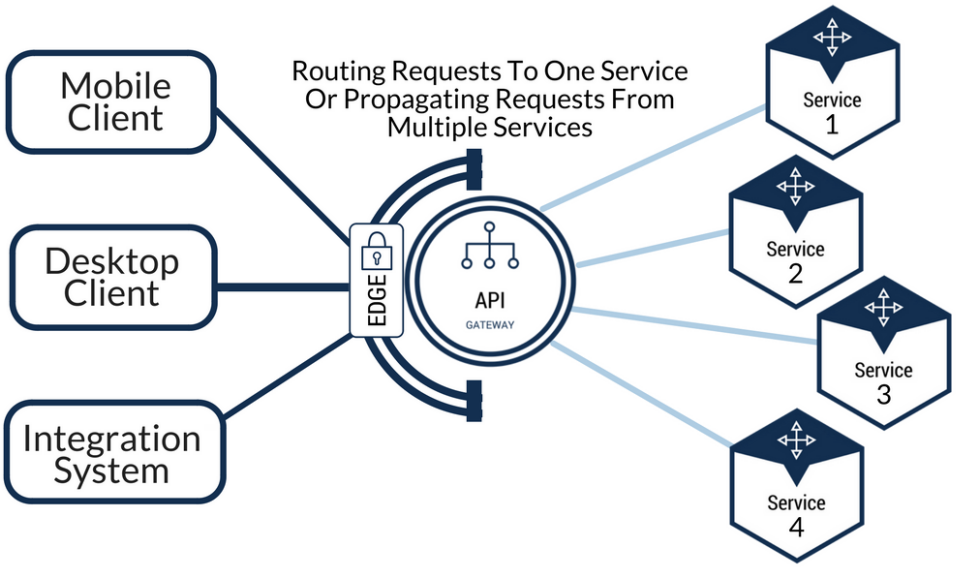
Highlight: API Gateway

How do the clients of a Microservices-based application access the individual services? They use an API gateway that is the single entry point for all clients.

In general, the Microservices API Gateway locates microservices using a Services Registry like Eureka. The API Gateway encapsulates the internal system architecture and provides an API that is tailored to each client. It accesses APIs, merging and transforming them into API shapes specifically for client access. It essentially provides a "front end" used to access the microservices underneath. It reduces the amount of round-trips between the application and the client. Load balancing, user identity authentication, and authorization can also be applied at the API Gateway. All requests from the client are routed from an Edge Controller to the API Gateway first, if one is defined.

- ➢ *Note:* An Edge Controller is an optional component. Client APIs can go directly to an API gateway. Reference the Edge Controller highlight for reasoning.

The API Gateway then routes that request to the appropriate microservice. The API Gateway often handles a request by invoking multiple microservices and aggregating the results into a succinct result.



Gallery 5: API Gateway

The Microservices API Gateway automatically performs all of the required protocol translations, so all appear to speak the same protocol even when they don't. It can translate between web protocols (such as HTTP and WebSocket) as well as web-unfriendly protocols that are used internally.

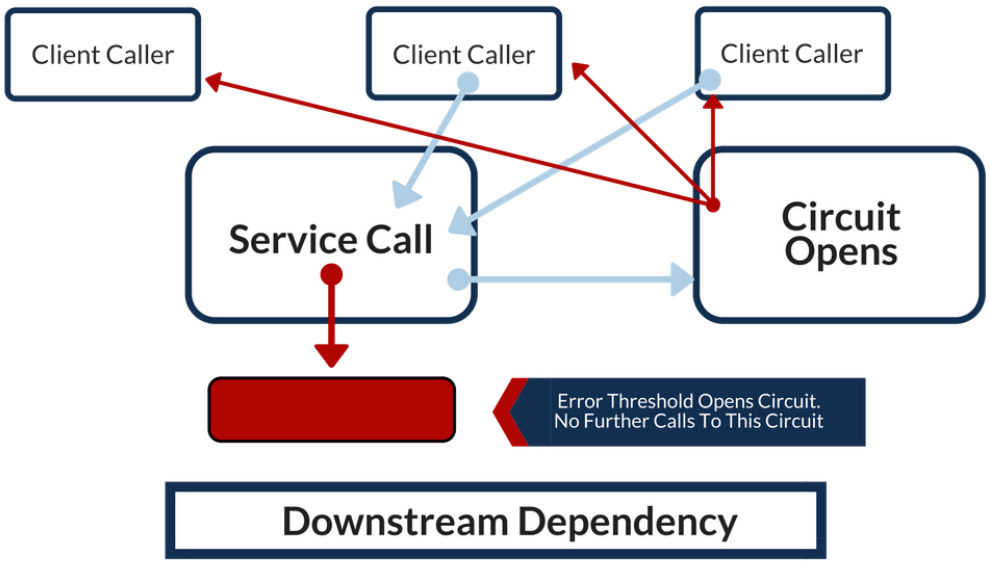
The caveat is that you need to ensure that the API Gateway does not become an application bottleneck or choke point. It is important that the process for updating the API Gateway is as

lightweight as possible. Partial failures must be handled; a single unresponsive microservice shouldn't make the whole request fail. Caching can often be used as an effective strategy for handling failure.

Highlight: Circuit Breaker

Circuit Breaker is a stability design pattern for gracefully handling when a downstream service becomes unhealthy. The two major goals of the Circuit Breaker pattern is to contain problems to the smallest area possible of a system and to provide a responsive interface to clients, even if the response is a failure.

The Circuit Breaker patterns works by wrapping requests to downstream dependencies in a component that monitors requests for slow and/or failed responses. If a defined threshold is reached the circuit is "opened" and calls to the downstream service are prevented for a defined wait period. After the wait period is exceeded the circuit enters a "half-open" state and a single request is passed through to the downstream service. If that request is successful the circuit is "closed" and the system returns to behaving normally. If the request fails the circuit remains "open" and restarts the wait timer. This process will continue until the downstream process recovers or some other steps are taken to address the issue.



Gallery 6: Circuit Breaker Pattern

The Circuit Breaker patterns improves system stability by preventing resources from being consumed by requests that will not or have a low possibility of succeeding. This is particularly important as circuit breakers, just like their real-life counterparts, are frequently "popped" in response to heavy demand.

Tel 877-521-7769 | keyholesoftware.com

Circuit Breakers also improve user experience by making the system more responsive. While in the most basic case the response might still simply be an error returned to the client, some implementations of the circuit breaker pattern, like Hystrix, allow for defining of special behavior to executed when a circuit is open. An example could be sending a static response to the user.

Hystrix Circuit Breaker

Many circuit breakers are binary and simplistic in implementation. However, there is a stand out option that Keyhole teams have begun to use whenever possible: Hystrix.

- Hystrix is an open source library focused on latency and fault tolerance. Hystrix helps you control the interactions between distributed services by adding latency tolerance and fault tolerance logic. [12]

Hystrix was developed by the Netflix API team. It is designed to stop cascading failures and enable resilience in complex distributed systems where failure is inevitable. The goal is for services to fail fast, rapidly recover, and gracefully degrade when possible.

Additionally, Hystrix enables near real-time monitoring, alerting, and operational control through the visual Hystrix Dashboard.[13]

- An implementation of the Hystrix Circuit Breaker pattern is a part of the Spring-Cloud project, Hystrix-Javanica. Hystrix-Javanica leverages the declarative nature of Spring annotations with very little configuration or manual coding; we can add Hystrix functionality directly to method signatures and to the project as a whole.

Hystrix Circuit Breaker is able to achieve these results by leveraging some other common design patterns. First, it wraps each call to a downstream dependency in a command, using the **Command pattern**, and isolates it in its own execution thread.

Second, it keeps a counter of all the errors and timeouts for each thread to help manage each call in isolation. If a circuit opens for one of the threads, then that individual call just emits an exception that can be handled by the application, and does not cause the calling method to crash.

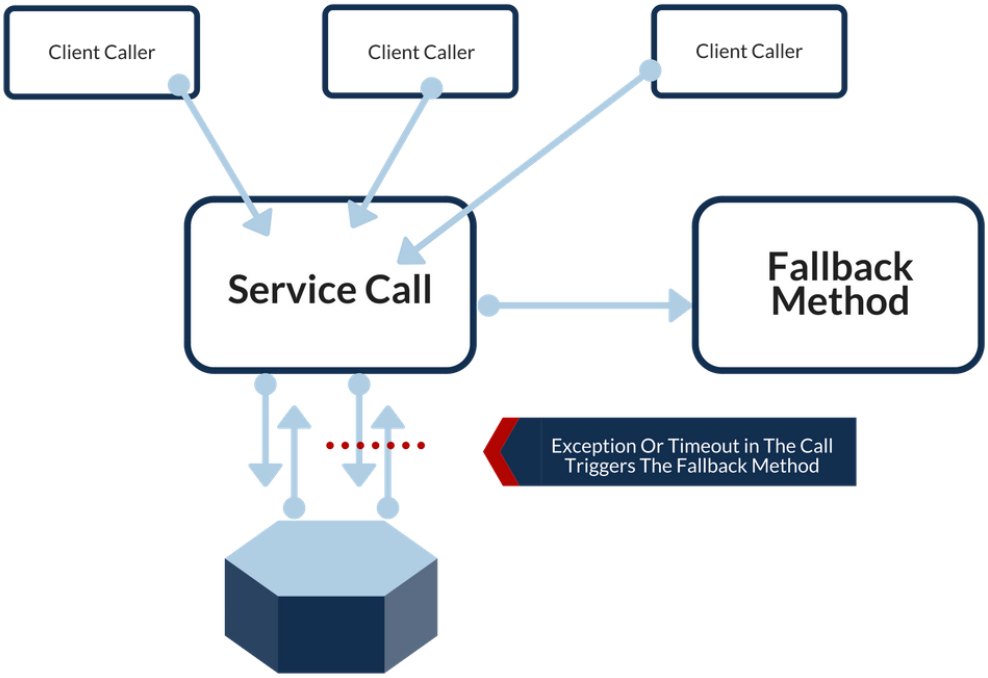
Fallback Method / Chain Of Responsibility

The Hystrix project also allows us to implement a fallback method that gets triggered when the circuit opens, and allows the application to more gracefully handle latency and fault issues.

[12] **Hystrix: Latency and Fault Tolerance For Distributed Systems.** https://github.com/Netflix/Hystrix
[13] **Hystrix Dashboard.** https://github.com/Netflix/Hystrix/wiki/Dashboard

➢ The Fallback Method is an implementation of the **Chain of Responsibility pattern** that allows a series of commands to hand off tasks when one is unable to complete it.

In the following example in "Gallery 7," the fallback method is an alternative method with the same signature as the original command that usually returns a null result or an empty object of the same shape, but could also be designed to return a static response. For example if the service for suggesting movies to watch based on your interest is down, a static list of popular movies could be returned instead.



Gallery 7: Fallback Pattern

If the original method is not responding in the configurable time span, then the fallback method is implemented to help the application triage requests and respond to requests in a reasonable timeframe, albeit with a null result. This is a preventative measure that can help the downstream dependency from being overrun with requests, and it can be implemented to include a retry.

Coupling the fallback method and circuit breaker allows for durable, resilient services that run until they are stopped.

Highlight: Failure As A Use Case

In an actual production environment, a multitude of things can go wrong. Problems like memory over-utilization and leaks, port exhaustion, thread exhaustion, connection pool

timeouts, too many resource file handles, and numerous others.

Even more potential issues are introduced when distributed systems such as Microservices are adopted, as the entire system has more moving parts than a standard monolithic web application. Service registries, load balancing, failover, and redundancy are essential. There is even more surface area for these types of potential failures and handling these types of failures is a characteristic of system stability.

A successful Microservices platform requires a durable and resilient environment that supports the ability to continuously deploy multiple services. Automated deployment is a must, and when possible, automated recovery from failures should be implemented, because failures will happen, (i.e. Murphy's Law[14]: Anything that can go wrong, will go wrong).

Instead of waiting for a failure to occur and seeing how durable and resilient your platform is, we suggest that you use a proactive approach and make failure a use case of your platform. A use case is a description of how users will perform and experience tasks on your website. It outlines a system's behavior as it responds to a request.

- Netflix has been a pioneer of this purposeful error strategy, using a framework called Chaos Monkey[15] which can be configured to randomly take down AWS resources (i.e. load balancers, etc.) during normal business hours.

- Keyhole Labs' Trouble Maker[16] is an open source tool that invokes service troublemaking issues both randomly and on-demand, configurable to produce heavy loads, killing of services, and exception throwing. Additionally, it provides an ad hoc dashboard console for testing of durability on-demand. Trouble Maker is not cloud dependent nor is it coupled to any platform, so it can be used within most enterprise environments.

When failure occurs—via Murphy's Law, Chaos Monkey, or Trouble Maker—automated or manual procedures should occur to remediate problems. That remediation should occur while still continuing to operate and serve users. If you know that failures are occurring at 3 p.m. and the help desk is not being called, then you know your system is durable and your pager will not be going off at 3 a.m!

Configuration

Appropriate configuration management must be addressed to ensure that Microservices complexity doesn't lead to inefficiencies and helps with developer agility.

[14] **Murphy's Law.** https://en.wikipedia.org/wiki/Murphy's_law
[15] **Chaos Monkey.** https://github.com/Netflix/SimianArmy/wiki/Chaos-Monkey
[16] **Trouble Maker.** https://github.com/in-the-keyhole/khs-trouble-maker

A traditional monolithic architecture only requires configuration of a single server side component. So, moving this single component through environments and applying load balancing is fairly simple. Contrast this to a MicroService style of architecture where there are many deployable elements for a given application. Automation is key along with a way for each component to obtain configuration specific to it's environment when it is deployed and started. This needs to be done with an automated mechanism, because manually supplying configuration properties is error prone and time consuming decreasing agility.

To help promote developer agility, Microservice components should apply a localized configuration convention. Defaulting to a local configuration , will allow developers to checkout code from the source code management (SCM) environment, and immediately start developing with out having to apply or worry about configuration.

Luckily, the open source community has supplied ready-to-use configuration management solutions for this.

- Apache Zookeeper is an open source distributed configuration mechanism that can be a solution for this.[17]

- Spring Cloud Config allows Spring-based applications to utilize a centralized configuration service.

Transparency

A critical aspect of managing any software system is having visibility into your system's health. This can provide valuable information on how users are interacting with your applications as well as vital information for when errors inevitably occur.

With monolithic applications, this is a relatively straightforward process of configuring a logging utility and heartbeat monitor to give a window into the health of your application. In a microservice environment however, there might be dozens of services with each service possibly having multiple instances. More thought must be put into how to monitor the health of your environment due to so many services and requests possibly interacting.

As with other common issues that have arisen from the introduction of Microservices, this too has been addressed with a combination of patterns and practices. The two biggest solutions would be log aggregation and correlation.

The first part, aggregation, means using a service to collect the logging output into a single location. A popular solution for this is the Elastic Stack (formerly known as ELK Stack), though

[17] **Apache Zookeeper**. https://zookeeper.apache.org

other options are available, such as Splunk.

Correlation means adding identifiers to requests so their interactions across your environment can be properly tracked. Within the Spring framework there is the library Spring Cloud Sleuth that handles request correlation automatically as well as providing several other useful features.

Options exist as well for providing health monitoring. Spring Actuator provides a RESTful API that can be further customized to provide information on the general health of a service, resource usage and thread dumps.

- The Elastic Stack (formerly known as the ELK Stack) is a popular open source solution to manage, filter, and view all the logging statements across a Microservices architecture.[18] It pairs Elasticsearch, Logstash, and Kibana.

- Spring Cloud Sleuth - Is an open source library developed by Pivotal. Sleuth provides a suite of tools for tracing requests through a service calls. [19]

- Spring Actuator - Provides a RESTful API for providing health and configuration information, as well as other features like thread dumps and stack traces. [20]

- Zipkin - Part of of the Spring Cloud Sleuth library, though an independent project in its own right. Zipkin was born out of lessons learned from Google's Dapper project. Zipkin randomly samples incoming request to map and time its execution through your microservices environment to give a snapshot of health over time. This can be useful for resource planning for regular traffic spikes.[21]

API Governance

API Governance is an integral part of Microservices. In Microservices, APIs are the primary mechanisms that services interact with to access business logic and data. This will naturally lead to a proliferation of available APIs.

Also, when organizations begin reorganizing teams into cross-functional teams, governing available APIs becomes even more important to avoid duplication of efforts and to not reinvent the wheel.

Documentation

An API developer portal mechanism can help ensure that you don't end up with run-away,

[18] **Elastic Stack.** https://www.elastic.co/products
[19] **Spring Cloud Sleuth.** https://cloud.spring.io/spring-cloud-sleuth
[20] **Spring Actuator.** https://spring.io/guides/gs/actuator-service
[21] **Zipkin.** http://zipkin.io

unmanageable APIs. It also provides a way to discover available APIs in an organization, demonstrating how they can be used, and how to access. This is especially helpful when onboarding new members to your team.

➢ GrokOla is a tool for development teams to document all API information - real-time usage statistics, API structure mapping, API usage, big-picture wiki documentation, and visual representations. GrokOla allows you to discover, learn, and visibly see how APIs are being utilized in the enterprise ecosystem.[22]

Automation Of API Documentation
Early adopters of Microservices emphasize another proven tenant: automate everything, even documentation.

Documentation has always been a point of contention for enterprise development teams. It may start strong, but never be updated. There are two strategies that we recommend to keep that from happening:

1. Have a standardized way of documenting your APIs, and
2. Have that documentation automatically integrated into your developer portal.

➢ The OpenAPI Specification (formally known as Swagger) provides a standardized way to document APIs and is widely recognized as the most popular open source framework for defining RESTful APIs.[23] Language-agnostic and both human and machine readable, OpenAPI maps all resources and operations associated with APIs, defining a set of files required to describe such an API.

A benefit of automated documentation is that it requires much less hands-on work to document APIs, allowing developers more time to spend on activities they enjoy.

Specific Developer Documentation Portals (like GrokOla) allow for the automatic integration of that Open API documentation into that portal when code has been successfully deployed.

➢ GrokOla API Governance features allow OpenAPI documentation to be integrated automatically into your documentation. Build mechanisms can be configured to import the information when code has been checked into a code repository and successfully built and deployed. Alternatively, Open API JSON files can be manually imported.

[22] **GrokOla.** https://grokola.com
[23] **Open API Initiative.** https://www.openapis.org

Shifting To Agility

Adopting immutable, containerized Microservices can help mitigate the ever-changing technology shifts in software development, tools, techniques, and frameworks. By having smaller surface areas of change in your application, you can more rapidly innovate when a change is needed.

But when using an agile process like Scrum where you are timeboxing development in an attempt to deliver "production-ready" software in short bursts of time, then the Microservices architectural style is a perfect fit.

Say that you're developing a Microservices application feature using an agile methodology. When a demo occurs of production-ready software, a button can pushed and it is really in production. This is as opposed to going into QA and acceptance testing for a long period of time.

Adoption By Different Development Communities

In regards to the Java side of enterprise development, support has been rampant. Companies such as Netflix, Groupon, PayPal, Airbnb, and others, have all implemented a Microservices architecture. Many have pioneered design patterns and produced various support frameworks released open source.

Conversely, the .NET community has simply not had the same kind of tooling and innovation that has been brought out by companies like Netflix and Amazon in the Java community. It took nearly three years after the release of Docker for Microsoft to get on the train. Nonetheless, it is coming along very nicely: specifically in regards to Service Fabric.

With Service Fabric you not only get the benefits of containerizing your deployable bits, but you also get the added benefit of having Microservices best practices built in.[24] Plus, Service Fabric is not limited to Azure. It is not even limited to Windows. You can run Service Fabric on Linux, in your local data center, or on AWS. If that wasn't enough to make you excited, your applications don't have to be .NET – they can be Java, C++, Ruby or others.

The "Platform" Is Key

As most Microservices practitioners should tell you, the most important and arguably most difficult part of adoption is the automation required to move all of these numerous moving parts in and out of environments.

Software companies such as Google, Amazon, and Microsoft have inspired and influenced the

[24] **Microsoft Azure Service Fabric.** https://azure.microsoft.com/en-us/services/service-fabric

shift and adoption of Microservices. They have provided infrastructure, services, and applications for the world to consume and access via the internet. These self-service web pioneers with no telephone numbers have invented and applied technologies essential to providing a DevOps platform that supports Microservices.

Just having the ability to obtain infrastructure—servers, resources, and datastores—in an on-demand manner is not enough. An automated platform that supports the numerous moving parts of a Microservice architecture is required. The following sections will describe components of this platform.

A Case For Containerization

Most enterprises utilize hardware virtualization from companies like VMware and Citrix.

However, Google needed to harness computing power to spread across an ever-increasing demand for processing power. So, Google engineers exploited an efficient operating system virtualization feature available in the Linux operating system.[25] This enables multiple isolated application server software stacks (containers) to share a single operating system.

Contrast this to an operating system hypervisor sharing multiple virtualized operating systems, which is much less efficient.

Applications have long been packaged into immutable artifacts like WARs, JARs, or DLLs. While these artifacts were immutable, the environments they ran in were not. As an application was introduced to a new environment, that environment would have to be configured to support it. This is often a frustrating and error-prone process as component software versions did not match between environments or they had complex configuration requirements.

Additionally, applications were prone to break unexpectedly as maintenance tasks were run or configurations were changed to support new applications being added to an environment. Containerization seeks to address this issue by allowing an application and the environment it runs in to be moved as a single immutable unit. These containers can be managed individually, scaled as needed, and deployed in the same fashion as compiled source code. Containers have been key to attaining agility, quality, scalability, and durability.

[25] **Google: 'EVERYTHING at Google runs in a container.** The Register. https://www.theregister.co.uk/2014/05/23/google_containerization_two_billion

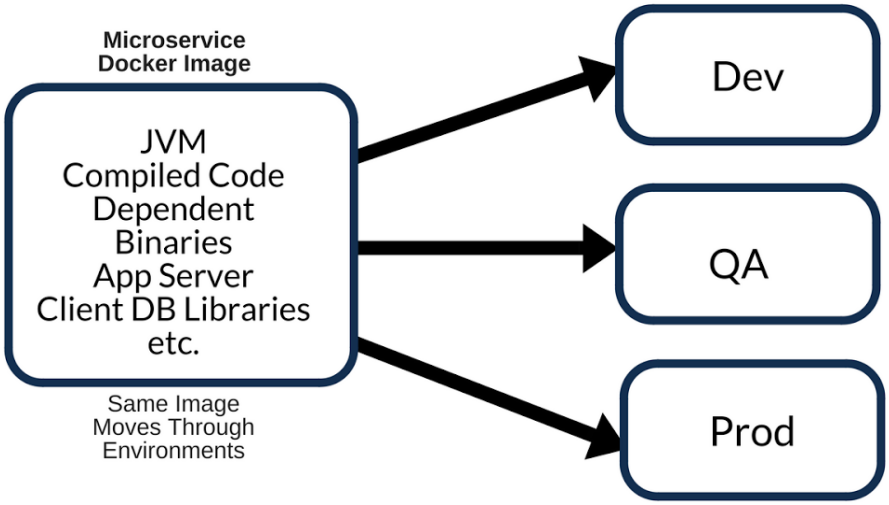
Containerization In Microservices

There will be many more moving parts in a Microservices architecture. Services need to be **immutable**. Essentially, this means that services need to be easily deployed, started, stopped, and discovered.[26] This is where Docker comes on the scene.

- ➢ Docker is open source software that allows you to build, run, test, and deploy distributed software containers. A Docker image can contain everything that software needs to run: code, runtime, system tools, system libraries, etc.

A Docker image enables you to quickly, reliably, and consistently deploy applications regardless of environment.[27] Essentially, a Docker container is an image of all the resources and software for an application or service.[28]

A Docker image that is tested can easily and quickly be moved to a virtualized server and started on a Port. Using a self-discovering service registry, these services can be moved, started, and stopped without downtime.



Gallery 8: Immutable Docker Container

For more on Docker, read an introductory blog written by Keyhole Software Consultant Zach Gardner titled Docker: VMs, Code Migration, and SOA Solved.

[26] **Rethinking building on the cloud: Part 4: Immutable Servers.** Butler-Cole, Ben (2013, June 10) http://www.thoughtworks.com/insights/blog/rethinking-building-cloud-part-4-immutable-servers
[27] **Docker.** https://www.docker.com
[28] **Docker: VMs, Code Migration, and SOA Solved.** Gardner, Zach (2014, November 20). https://keyholesoftware.com/2014/11/20/docker-vms-code-migration-and-soa-solved

Repository And Immutability

Docker has become so effective that a number of other open source and commercial products became available to help manage, orchestrate, and store Docker images. There are cloud options available for container repositories. Docker Hub, Amazon and Google have cloud-based repositories.

- Docker Hub is a cloud-based registry service that provides a centralized resource for container image discovery, distribution and change management, user and team collaboration, and workflow automation throughout the development pipeline.[29]

- Amazon's EC2 Container Registry (ECR) is fully-managed Docker container registry that makes it easy for developers to store, manage, and deploy Docker container images. It is integrated with Amazon EC2 Container Service (ECS), simplifying the development workflow.[30]

- Google Google Cloud Platform's Google Container Registry is a private Docker registry with continuous integration and delivery systems. It runs on Cloud Platform to provide consistent uptime on an infrastructure protected by Google's security.[31]

There are also public repositories that allow containers to be stored and queried based on metadata, just like Maven artifacts.

- Nexus/Sonatype[32] and Quay[33] are premise-based Docker repositories that are available.

Continuous Integration And Continuous Deployment

As the previous section described, a container image is the only artifact that is built, tested, and deployed. As containers form a complete software stack for a given Microservice, they only need to deployed to a docker-aware server, then started.

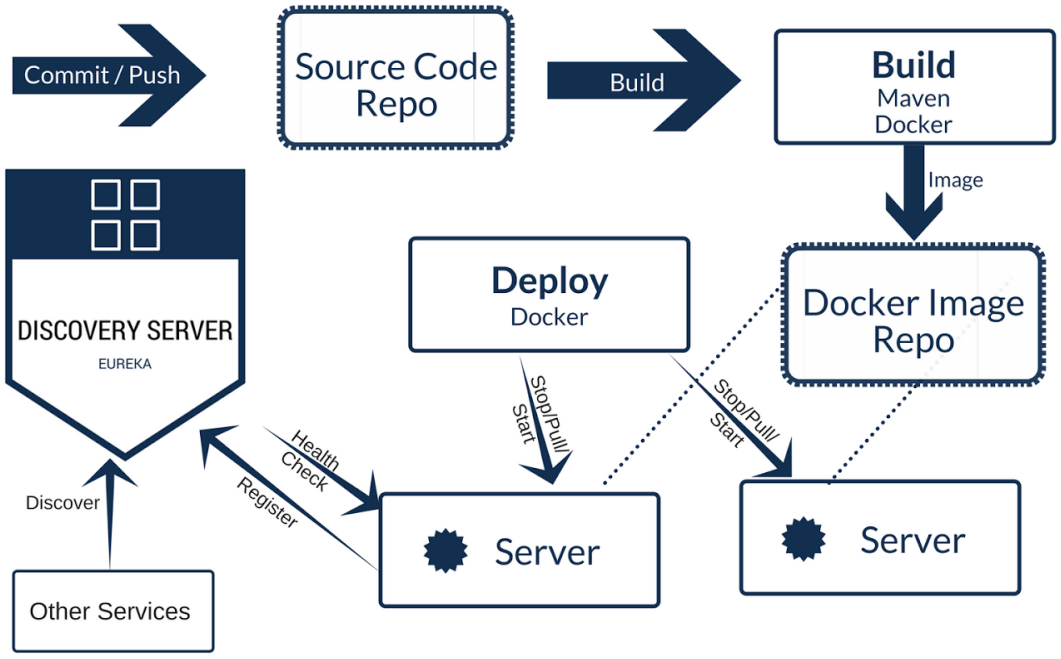
- Jenkins is an open source, Docker-aware continuous integration server. Jenkins would be installed on a server where the central build will take place. It is cross-platform, so you can integrate Jenkins with a number of testing and deployment technologies.[34]

The stage is set for a continuous deployment platform. Simply point developers to a source

[29] **Docker Hub**. https://docs.docker.com/docker-hub/
[30] **Amazon EC2 Container Registry**. https://aws.amazon.com/ecr/
[31] **Google Cloud Platform**. https://cloud.google.com/container-registry
[32] **Nexus Repository OSS**. Sonatype. https://www.sonatype.com/docker
[33] **Quay**. https://quay.io
[34] **Jenkins**. https://jenkins.io

code repository that kicks off a Docker image build on check-in. The continuous integration mechanism (something like Jenkins) can stop one of two deployed Microservices, build an image, run unit and automated tests, and if they pass, then deploy, and start. The service would register itself and be ready for access. Repeat the process for the other redundant services that have been deployed.

The “Gallery 9” diagram depicts this.



Gallery 9: Continuous Build And Deployment

Another strategy that validates stability is to delay deployment of redundant services for a “curing” timeframe. If all is well, deploy redundant services.

Combining containerization with the decoupled, discoverable, and isolated nature of the Microservices architecture style supports a continuous deployment environment that allows microservices to be deployed at will. This is in contrast to the traditional monolithic testing and offline all-hands-on-deck deployments required for large traditional monolithic applications.

Platform Infrastructure And Orchestration

Organizations can exploit on-premise or off-premise infrastructure-as-a-service (IaaS) solutions. Computing resources like servers, data sources, and storage to be acquired in an on-demand manner.

Orchestration software, such as the following, can be leveraged in your software applications.

Tel 877-521-7769 | keyholesoftware.com

All of these solutions can help you to automate and manage deployments of containers to an infrastructure platform, in an automated way.

- Kubernetes by Google, is an open source container management platform for managing containerized applications across multiple hosts. It provides basic mechanisms for deployment, maintenance, and scaling of applications. Additionally, it enables scheduling, upgrades on-the-fly, auto-scaling, and constant health monitoring.[35]
- Universal Control Plane by Docker[36] is an enterprise-grade cluster management solution. Install it on-premises or in your virtual private cloud. It helps you manage your Docker cluster and container applications from a single place.[37]

- Mesos, by the Apache Foundation, is a distributed systems kernel. The Mesos kernel runs on every machine. It provides applications (e.g., Hadoop, Spark, Kafka, Elasticsearch) with APIs for resource management and scheduling across entire data center and cloud environments.[38]

- Cloud Foundry (developed by VMware and part of Pivotal Software) is an open source cloud application platform for developing and deploying enterprise cloud applications. It automates, scales and manages cloud apps throughout their lifecycle. The platform leverages containers to deploy applications. It enables businesses to take advantage of the latest innovations from projects, such as Docker and Kubernetes, to increase the ease and velocity of managing production-grade applications.[39]

- OpenShift, by Redhat, is a Platform-as-a-Service container application platform that allows developers to quickly develop, host, and scale applications in a cloud environment. It natively integrates technologies, like Docker and Kubernetes, and combines them with an enterprise foundation in Red Hat Enterprise Linux.[40]

- Service Fabric , by Microsoft, is a distributed systems platform that makes it easy to package, deploy, and manage scalable and reliable microservices. In addition to containerization, you get the added benefit of having Microservices best practices built in.[41] Service Fabric is not limited to Azure or Windows, as you can run Service Fabric on Linux, in your local data center, or on AWS.

While there are many options available, the bottom line is there needs to be some kind of container mechanism to obtain infrastructure and deploy containers in an automated fashion. Automation is the key to success.

[35] **Kubernetes**. https://kubernetes.io
[36] **Docker Enterprise Edition**. https://www.docker.com/products/docker-datacenter
[37] **Universal Control Plane Overview**. https://docs.docker.com/datacenter/ucp/2.1/guides
[38] **Mesos**. http://mesos.apache.org
[39] **Cloud Foundry**. https://www.cloudfoundry.org
[40] **Open Shift**. https://www.openshift.com
[41] **Microsoft Azure Service Fabric.** https://azure.microsoft.com/en-us/services/service-fabric

Getting Started With Microservices

What's The Catch?

Microservices are not a panacea for success. To avoid the giant-ball-of-mud problem, infrastructure and DevOps need to be put place in order for this to be agile and manageable. Otherwise, you'll have a bigger mess to deal with.

Fortunately, mechanisms and processes to manage this have been pioneered and made available to us by high-profile companies like Netflix that give us an approach for successfully supporting a Microservices architecture.

On the other hand, skilled resources and time are required to define proper services and their sizes, in addition to implementing the required DevOps infrastructure. Current thought leaders in this space are debating if every application should begin as a Microservice, or whether smaller applications should stay as a single deployed architecture with only larger size applications using a Microservices architecture.

Our opinion at Keyhole Software is that there is no reason the Microservices infrastructure cannot be used to deploy an application's traditional single server-side components along with more granular Microservices. There is likely a hybrid approach where certain services are reused by a traditional single deployment-based application.

A existing monolithic application architecture can be adapted to engage a microservices platform architecture via a filtering and routing mechanism. The required automated platform to support Microservices can be built out, including the legacy monolithic components. This will allow new functionality to be created and made available for reuse. And this approach will allow the existing monolith to be pruned down, instead of tacking new functionality on and making the monolith a bigger monolith.

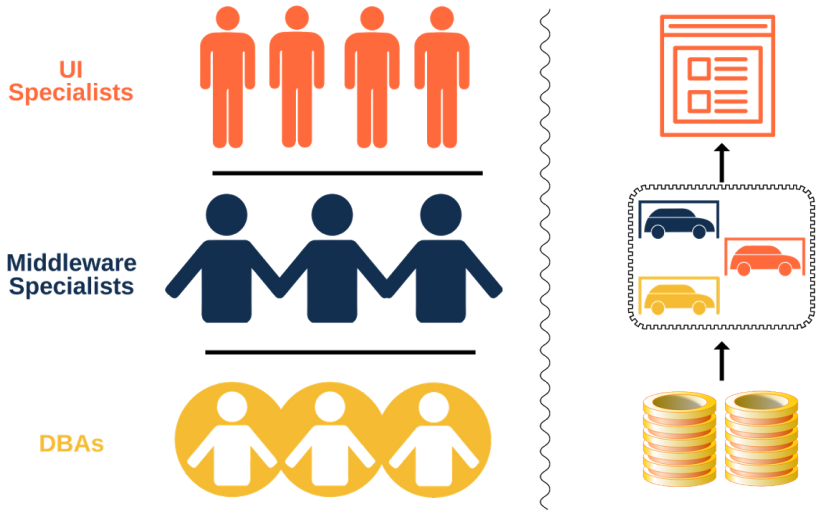
Organizational Shift

Adopting a Microservices architecture has a bigger impact on the organization given that there are more moving parts.

System Administration personnel will need to get on board with allowing teams to move containerized services in and out of service frequently and rapidly. Management and stakeholders may have to tolerate a little more expense and time to get the proper DevOps environment established in order to accommodate, govern, and manage many services.

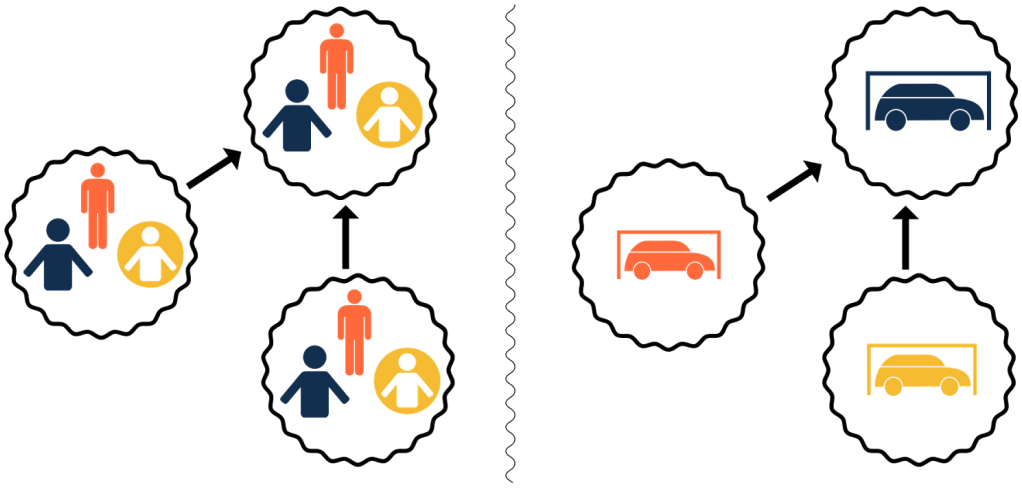
Traditionally, groups of individuals in an application initiative surround their competencies, with separate teams for DBAs, Middleware, and the User Interface, as shown in the following

image. This follows Conway's Law, which asserts that software organizations are constrained to produce application designs that are copies of their communication structures.[42] This often leads to unintended friction points.



Gallery 10: Software Teams Following Conway's Law

We suggest a shift in the traditional communications paths of an IT shop: taking organizations from Conway's Law to the "Inverse Conway."



Gallery 11: Software Teams Following Inverse Conway's Law

Inverse Conway implies that the organization should partition staff to be responsible for a specific service from birth to grave.[43] Contrast this to the normal IT roles of responsibility and

[42] **Conway's Law.** https://en.wikipedia.org/wiki/Conway's_law
[43] **Dealing with Creaky Legacy Platforms.** Leroy, Jonny and Simons, Matt (2011, February 3). Originally published in December 2010 issue of the Cutter IT Journal. http://jonnyleroy.com/2011/02/03/dealing-with-creaky-legacy-platforms

communication. There include siloed teams focused on gathering requirements, development, QA, DBA, and production.

This doesn't mean that resources aren't moved around. They are assigned to maintain and support specific services. These include production support for those services. You can see how this organizational change could have a big impact.

Team Adjustment

This transition gives everyone the opportunity to work with modern methodologies and tools. As learning any new skill takes time, we suggest that you get buy-in from your development team early on.

There is a learning curve that must be factored into the project schedule. Adequate support of the learning process of your team is an integral part of initiative success; whether that means bringing in outside consultants or mentors to establish best practices, or providing courseware on new topics.

A major project of this scope won't happen overnight, but the agility, performance, and real-time nature of the results are well worth the effort.

When Not To Apply Microservices

It is clear that Microservices has immense potential for enterprise application. A Microservices approach is not a fit for every application, however.

Be mindful of the distributed computing characteristic of Eventual Consistency. The CAP theorem explains and identifies it. The CAP theorem states that it is impossible to create a system that possesses all three of the following points:

1. strong consistency (where all nodes see the same data at the same time),
2. guaranteed availability for data—create, update, and delete operations, and,
3. provides fault tolerance. [44]

Due to this theorem, we assert that applications that have actual real-time requirements and atomic transaction needs are most likely not a good match for Microservices-based applications. Or, at least service boundaries should not span a real-time/transaction API domain.

Consider that many enterprise applications are basically CRUD-based applications that require reliability and scalability, and eventual consistency is not a barrier. Often your CRUD

[44] **Brewer's conjecture and the feasibility of consistent, available, partition-tolerant Web services.** Gilbert, S., Lynch, N. (2002). ACM SIGACT News 33(2): 51-59.

approach can be rethought to utilize event-based mechanisms to overcome transactional challenges.

An Example

An example application that might not be a good fit, or at least where critical functionality should not be distributed, would be a transaction-based financial application such as banking account management system or some kind of real-time device control application. NoSQL applications that provide schema-less persistency that can be scaled across process Nodes also have to exist in an eventual consistency environment.

This does not mean a Microservices-based architecture cannot be used and you have to go back to a Monolithic application. The takeaway is the service boundary might be a little larger for a service implementation that requires full atomic transactions or very low latency.

Suggestions For Microservices Adoption

We suggest first trying an incremental adoption approach having adequate analysis and proof-of-concept. As a first step, try implementing and supporting some architecture-based services. Suggested ideas follow.

- Create an authentication service that your existing applications can be retrofitted to access. Alternatively, create a role/privilege-based service that your existing applications can be retrofitted to access.
    - Almost every application requires this kind of functionality and you will be able to establish the required DevOps/continuous delivery environment without addressing the difficulty of end user requirements while tooling up the architecture.

- Move the user interface of your applications to an Single-Page Application style. Separate the data from the presentation of data by having a model layer that handles data and a view layer that reads from the models.

- Another idea is to implement a Microservices environment for support code services. Many applications require presenting a list of support codes such as city, states, zip codes, or more domain-specific codes.

- Introduce containerization, continuous integration, and continuous deployment and apply automation wherever you can.

While these incremental steps seem trivial, the process will provide easy-to-understand services that have much reuse potential.

Most importantly, these exercises will allow your team to engage and establish the mutable containerized services deployed in a continuous way, before the explosion of Microservices that will result once this style is fully embraced.

When You Have A Current Monolith

We suggest an incremental approach to adoption, especially when you have a monolithic application you are seeking to modernize to Microservices.

There are a number of strategies for modernization, but in all modernization efforts, success is much more likely when approached strategically.

Incremental Growth

Stop making your current monolith bigger with new features; implement new functionality as microservices. Gradually build a new application consisting of standalone microservices, and run it in conjunction with your monolithic application. Over time, the monolithic application will shrink until it either disappears or becomes a microservice itself. This can be done by implementing a proxying filter at the application server level.

Splitting The Front And Back End

Take your current monolith and split the application Presentation Layer apart from the Data Access Layer and Business Logic. This approach allows you to develop, deploy, and scale the now-two-separate applications independently of one another. Now you would have two smaller monoliths, so it is only a partial solution.

Incremental Extraction Of Services

Over time, turn existing modules within your monolithic application into standalone microservices. With each module that is taken from the monolith, the application shrinks. Over time, the monolith will disappear or become small enough to just be another service.

How Services Are Sliced

Whenever you're breaking something down, you must make the decisions as to where to split the pieces. Especially when modernizing a currently-existing application, ample care must be taken in determining by what principles your team will separate services.

Domain-driven design[45] is one way you could approach slicing up the monolith as a way to help define service boundaries. It can help enforce a bounded context which is important in isolating services. This can be a challenge when a monolith is also accessing a monolithic data

[45] **Domain-Driven Design.** https://en.wikipedia.org/wiki/Domain-driven_design

store, but there are some realistic strategies that can be applied where service-bounded context data is backed by a monolithic data store. See Keyhole Software blog Implementing a Bounded Context.[46]

Converting a monolith "module" into a service is time consuming. You must prioritize the conversion, ranking modules by the benefit you will receive. The application should be broken down as a composite of business workflows. First identify service boundaries. Define coarse-grained boundaries (or seams). Existing seams can often provide logical points at which to make a separation, turning into a service.

Many microservice groups explicitly expect certain services to be scrapped rather than evolved in the longer term. Independent replaceability of services is also a key consideration. Look for points that you can imagine rewriting without affecting its collaborators.

Depending on your application functionality:

- Typically, it can beneficial to extract **frequently-changing modules**. Once you have converted a module into a service, you can develop and deploy it independently of the monolith. This will accelerate development.
- Consider turning modules with particularly **high resource requirements** into services first. For example, it could be worthwhile to extract modules that implement particularly complicated or expensive algorithms. Then the service could then be deployed on hosts with lots of CPU.
- Consider transitioning features to services that are **inherently temporary**. For example, consider specialized pages that are time or market-dependent.

Alternatively, if you find yourself repeatedly changing two services simultaneously, that's a sign that you should merge the two services. If the combined service can provide multiple functions without adding design complexity or increasing message sizes, it could reduce implementation and usage costs.

Serverless Architecture Shoutout

It's worth mentioning an emerging architecture style that is a natural extension of a Microservices style of architecture: "Serverless Architecture." The two can arguably coexist.

Serverless architectures refer to applications that significantly depend on third-party services to remove the server from the equation. It has been called "Backend-as-a-Service."

Essentially, you have a platform that allows small APIs to be implemented, just as you would

[46] **Implementing A Bounded Context.** Keyhole Software Blog. Pitt, David. (2016, March 21). https://keyholesoftware.com/2016/03/21/implementing-a-bounded-context

within a Microservice API. However, this is done in a more granular manner. You don't have to provision and configure servers or data sources; this is done automatically by the platform. When the APIs are executed, computing resources and server configuration are automatically provisioned, started, and your API deployed.

Depending upon usage and load, resources are automatically added. When not in use, it is discarded. Hence the meaning of "serverless," due the notion that the server is abstracted away and exists in a transient, on-demand manner. When not in use, there is no server instance alive or accessible. Just like in the Microservice style, API Gateways exist to aggregate APIs for individual applications or integrations.

It requires an advanced automated computing resource operations platform. A couple of notable cloud-based, Serverless Architectures implementations are:

- Amazon Web Services' Lambda allows you to run code without provisioning or managing servers. You pay only for the compute time you consume as there is no charge when your code is not running. Supports virtually any type of application or backend service.[47]

- Azure Functions is an event-based, serverless solution on Microsoft Azure. Azure will scale as needed and you are charged based on the number of resources Azure Functions needs to do so, for only as long as it takes your code to execute. It supports JavaScript, C#, F#, as well as scripting options.[48]

Agility is an obvious benefit of this architecture style, in addition to cost savings. Amazon's Lambda allows a massive amount of users to be supported for pennies on the dollar. Services are expanded or contracted based on fluctuations of the numbers. Horizontal scaling is completely automatic and managed by the provider. This cost savings is due to Economy of Scale.

Vendor dependency is a concern. With any strategy that includes outsourcing, you will give up control of some of your systems to a third party. That lack of control could be seen as system downtime, unexpected limits, change in costs, forced API upgrades, and more. You will be relying more on vendors for the success of your server.

A significant amount of enterprise vendor lock-in has to do with how data is stored. If you choose the Serverless approach, we suggest you separate the business logic and data access away from deployment. For example, say that your enterprise is using Amazon S3 Cloud storage and is heavily using S3 APIs. In that case, you will likely remain with S3 unless significant rewriting is done, unless you've abstracted it a bit.

[47] AWS Lambda. https://aws.amazon.com/lambda
[48] Azure Functions. https://azure.microsoft.com/en-us/services/functions

Serverless Architecture is bleeding edge, especially for the enterprise. It is an unproven concept at scale. As such, some of the benefits we see could end up being unproven as the solutions mature. At some point we will begin to see patterns of recommended practices come onto the scene, just like Netflix and Airbnb introduced for Microservices. However, you must determine if your enterprise is willing to be the pioneer.

For enterprises that wish to modernize, it is our recommendation that enterprises should concentrate first on establishing a platform, be it on or off-premise, that supports a Microservice style of architecture. We suggest that you consider Serverless as an option, but only after migrating to Microservices. The technologies, patterns, and processes defined will position your organization to possibly take advantage of the emerging Serverless architecture style.

White Paper Conclusion

Enterprise organizations seek to remain competitive and efficient. To do so, they are in a never-ending process of adding new features while also maintaining existing functionality. Organizations that rely on Monolithic applications have been pushed to the breaking point. The cost of maintaining the Monolith has increased to the point that a Monolith can no longer increase organizational competitiveness and efficiency. Additionally, the path to modernize from a monolith can be difficult.

Moving forward, enterprise organizations need to establish an architecture that allows them to change in the future so they do not find themselves in the same spot ten years from now.

Understand that Microservices is not a silver bullet. Microservices seeks to address the failings of previous architectural styles by: allowing teams to focus on a narrower domain with a smaller surface area of change increasing their agility, having smaller units of scale to increase scalability, providing smaller isolation units to increase fault tolerance. By applying the features of a Microservices platform as covered in this white paper, these benefits can be realized.

A Microservices architecture is achieved by implementing independently-deployable services that are built around business capabilities, automated deployment, containerization, and intelligence in the endpoints. Additionally, configuring the correct “Platform,” DevOps, and infrastructure, in conjunction with established design patterns like API Gateway and Circuit Breaker, can increase chances for success.

If you are a typical organization looking to rewrite a Monolithic application or build a new application, then a Microservices styles of architecture deserves an in-depth look. As with any

new technology, we suggest that you initially try an incremental adoption approach with adequate analysis and consulting support.

We have highlighted a number of both free and paid Microservices software tools. We suggest you consider them, others on the market, and new technologies as they emerge. Adequately analyze the options to determine their suitability for your organization's goals.

We at Keyhole Software have led many projects in the Microservices architectural style the last few years. Results have been extremely positive. For many of our clients, this has become the default style for building enterprise applications going forward.

Will you use a Microservices style of architecture for your next application?

About Keyhole Software

Keyhole Software is a software development and consulting firm with a team that loves technology. Our expert employee consultants excel as "change agents," helping our clients to be successful with software technologies and approaches that bring competitive advantage.

We consult nationally with clients across the United States. The Keyhole Software corporate office is located in Leawood, Kansas, just south of Kansas City. Additional teams are located in St. Louis, Chicago, Lincoln, and Omaha. We frequently assist clients with custom application design, development, and modernization initiatives with Java, JavaScript, and .NET technologies. See recent Keyhole Software projects here.

Keyhole was founded on the principle of delivering quality solutions through a talented technical team. As such, knowledge transfer is important to us. To our clients, we offer various techniques to provide the most value: one-on-one or group mentoring, lab/lecture educational courses, and access to our knowledge transfer engine GrokOla.

Related Services Snapshot

- Technology Consulting – Expert Keyhole Consultants work with organizations to analyze the current state of applications, recommend technical directions, and develop strategic plans for modernization.

- Application Development – Individual members or entire teams of Keyhole Software Consultants perform expert development services: application analysis, design, development, testing, and enhancement of custom applications.

- Education – Instruction of a wide range of custom courses and technical mentoring programs for knowledge transfer to groups or individuals, including:
  - Course: Microservices For The Java Enterprise[49]
  - Course: Microservices In The Microsoft Enterprise[50]
  - Presentation: Java DevOps & Microservices In Action[51]
  - Presentation: .NET Microservices Architecture Made Simple[52]

Contact

Keyhole Software Address: 8900 State Line Road Suite 455 Leawood, KS 66206
Contact: 877-521-7769 | asktheteam@keyholesoftware.com | https://keyholesoftware.com

[49] **Microservices For The Java Enterprise**. https://keyholesoftware.com/microservices-course/
[50] **Microservices In The Microsoft Enterprise**. https://keyholesoftware.com/microservices-microsoft/
[51] **DevOps & Microservices In Action**. https://keyholesoftware.com/devops-and-microservices-live/
[52] **Microservices Architecture Made Simple**. https://keyholesoftware.com/microservice-architecture-simple/