DZone

THE DZONE GUIDE TO

Microservices

Breaking Down the Monolith

VOLUME I

BROUGHT TO YOU IN PARTNERSHIP WITH

ca technologies

CLOUD FOUNDRY FOUNDATION

DZone Jobs

INSTANA

IBM

Lightbend

macaw

Twistlock

DEAR READER,

The architectural pattern known as "microservices" has yet to hit its 10th birthday, but it already feels well-established. First coined in the 2010s, it aims to make large-scale application development and maintenance easier. Microservices allow applications to adapt and scale when needed by breaking their architecture into loosely coupled services that developers can easily change, replace, and scale.

They help large teams work collaboratively on projects, as each team or team member can work on individual services without blocking other team members. Spikes in demand can be easier to handle with microservices, which allow you to add new instances of services when needed, and removing them when they're not.

The pattern attracted so much popularity that it's led to increased interest and development in technologies that work well in collaboration with it. This includes new design processes, containers, API design (and subsequently GraphQL), message queues, front-end technologies, CI/CD, and service orchestration frameworks.

Developers love microservices, as they enable them to have more control over the design of their particular application area. They are also able to experiment with new languages and practices without changing an entire application, swapping in experimental components and monitoring their effects. Architects, product owners, and engineering managers should be aware that they use microservices-based architectures only when necessary and have enough resources. There will be significant migration time, and the approach can increase complexity in testing, deployment, and deciding how to divide the services of an application.

Recent years have seen the pattern applied in a wider context. I spent much of my programming past working with huge monolithic content management systems (CMS) where we struggled against a maze of interconnected parts. There is now a world of "headless CMS" platforms that allow developers to create specialized services that excel at their tasks. For example, a solid content management service that feeds content to a JavaScript front-end framework.

This guide focuses on crucial pieces of the microservices puzzle, helping you construct an effective, meaningful application architecture. We'll focus on the best approaches to reduce overhead during migration; how individual services communicate with each other, including the messaging options and formats; and how teams working with microservices can better communicate with each other. When it comes to digging deeper into individual applications, you'll learn about the best hosts for your applications, from container options and patterns to the best approaches on the JVM. We also cover how to maintain consistency between distributed nodes of an application, as what use is a distributed application if it's data never matches? Finally, to inspire you when you need guidance, we have case studies from developers and architects explaining how they tackled problems and their experiences.

Microservices introduce a new way of developing and managing your applications, but there's a lot of knowledge available, and with this guide, we will get you on the right path as quickly as possible. Enjoy.

BY CHRIS WARD
ZONE LEADER, DZONE

TABLE OF CONTENTS

3 Executive Summary
BY MATT WERNER

4 Key Research Findings
BY G. RYAN SPAIN

8 Microservices on the JVM with Actors
BY MARKUS EISELE

12 Streamlined Microservice Design in Practice
BY MATT MCLARTY & IRAKLI NADAREISHVILI

15 An API-First Approach for Microservices on Kubernetes
BY BORIS SCHOLL & CLAUDIO CALDATO

20 Infographic: Big Things Come In Microservices

22 Microservices and Team Organization
BY THOMAS JARDINET

26 Communicating Between Microservices
BY PIOTR MINKOWSKI

29 Diving Deeper into Microservices

32 What to Consider When Dealing With Microservices Data
BY CHRISTIAN POSTA

36 Reactive Persistence for Reactive Systems
BY MARK MAKARY

38 Executive Insights on the Current and Future State of Microservices
BY TOM SMITH

40 Microservices Solutions Directory

45 Glossary

PRODUCTION

Chris Smith
DIRECTOR OF PRODUCTION

Andre Powell
SR. PROD. COORDINATOR

G. Ryan Spain
PROD. PUBLICATIONS EDITOR

Ashley Slate
DESIGN DIR.

Billy Davis
PRODUCTION ASSISSTANT

MARKETING

Kellet Atkinson
DIR. OF MARKETING

Lauren Curatola
MARKETING SPECIALIST

Kristen Pagàn
MARKETING SPECIALIST

Natalie Iannello
MARKETING SPECIALIST

Miranda Casey
MARKETING SPECIALIST

Julian Morris
MARKETING SPECIALIST

BUSINESS

Rick Ross
CEO

Matt Schmidt
PRESIDENT

Jesse Davis
EVP

SALES

Matt O'Brian
DIR. OF BUSINESS DEV.

Alex Crafts
DIR. OF MAJOR ACCOUNTS

Jim Howard
SR ACCOUNT EXECUTIVE

Jim Dyer
ACCOUNT EXECUTIVE

Andrew Barker
ACCOUNT EXECUTIVE

Brian Anderson
ACCOUNT EXECUTIVE

Chris Brumfield
SALES MANAGER

Ana Jones
ACCOUNT MANAGER

Tom Martin
ACCOUNT MANAGER

EDITORIAL

Caitlin Candelmo
DIR. OF CONTENT & COMMUNITY

Matt Werner
PUBLICATIONS COORDINATOR

Michael Tharrington
CONTENT & COMMUNITY MANAGER

Kara Phelps
CONTENT & COMMUNITY MANAGER

Mike Gates
SR. CONTENT COORDINATOR

Sarah Davis
CONTENT COORDINATOR

Tom Smith
RESEARCH ANALYST

Jordan Baker
CONTENT COORDINATOR

Anne Marie Glen
CONTENT COORDINATOR

Want your solution to be featured in coming guides?
Please contact research@dzone.com for submission information.

Like to contribute content to coming guides?
Please contact research@dzone.com for consideration.

Interested in becoming a DZone Research Partner?
Please contact sales@dzone.com for information.

Special thanks to our topic experts, Zone Leaders, trusted DZone Most Valuable Bloggers, and dedicated users for all their help and feedback in making this guide a great success.

Executive Summary

BY MATT WERNER
PUBLICATIONS COORDINATOR, DZONE

While service-oriented architectures have been a fixture in the software development world for years, the concept of microservices is much more recent, and has taken the world by storm. There's an enormous amount of excitement around these architectures on DZone and beyond, but when writing this guide, DZone wanted to know how many people were actually using them, and have they really helped developers with their lives? To find out, we surveyed 605 DZone readers, and have shared the results in our first ever Guide to Microservices.

WHY MICROSERVICES?

DATA

Of those who are using microservices, 81% want to make easily scalable applications, 71% want to enable faster deployments, 50% want to have teams focused on particular pieces of an application, and 42% want an easy way to find where an application is failing.

IMPLICATIONS

81% of respondents who use microservices say their lives are easier as a result, implying that moving to a microservices architecture for their applications led them to achieve most of their goals. In contrast, only five respondents (about 1%) tried microservices and decided not to use them.

RECOMMENDATIONS

The majority of microservices users say that they are worth the investment and helped them accomplish their goals. If an organization is developing a new application, using a microservices architecture may be the best way to create additional value and future-proof software. Before developing microservices, determine if they fit your application's use case. Does it need to be easily scalable? Does it require several different components that would be better to separate into services?

NO LACK OF TOOLS, BUT LACK OF USE

DATA

58% of respondents do not use a service discovery tool, 74% do not use a platform to manage their microservices, and 72% do not use a distributed tracing tool. 58% of users do not use a service discovery tool, and 22% do not secure their services.

IMPLICATIONS

Many developers have yet to use tools that might make microservices easier to manage beyond tools like containers and frameworks to build them, such as Java EE, Spring Boot, and MicroProfile. This is likely due to the recent abundance of options and popularity of the technology.

RECOMMENDATIONS

47% cited monitoring as a chief concern, but tools that assist with monitoring microservices, such as service discovery or management platforms, are not being used. Investigate tools like service meshes and API gateways that make microservices management easier.

THREE OF A PERFECT PAIR

DATA

67% of respondents are currently using DevOps processes (similar to results we've seen in our 2017 DevOps and Containers guides). 34% use containers in development, 12% use them in production, and 19% use them in both stages. Those who use microservices in either prod or dev are more likely to use DevOps or Continuous Delivery, and those who use microservices in prod are more likely to use containers in prod, as well.

IMPLICATIONS

Developers using microservices are also likely to use technologies that lend themselves to building distributed applications like containers, or are likely to have experience creating cultural and technical change using DevOps methodologies to further decrease time-to-market.

RECOMMENDATIONS

Those using DevOps or Continuous Delivery methodologies may find that microservices and containers, while requiring a lot of training and setup, may help teams deploy code faster than before. It is probably easier to start experimenting with a new application or product, and leave the refactoring of legacy apps for later once an organization can evaluate how effective microservices are for themselves.

Key Research Findings

BY G. RYAN SPAIN
PRODUCTION COORDINATOR, DZONE

DEMOGRAPHICS

605 software professionals completed DZone's 2017 Microservices survey. Respondent demographics are as follows:

- 39% of respondents identify as developers or engineers, 17% identify as developer team leads, and 15% identify as software architects.
- The average respondent has 13 years of experience as an IT professional. 51% of respondents have 10 years of experience or more; 17% have 20 years or more.
- 40% of respondents work at companies headquartered in Europe; 28% work in companies headquartered in North America.
- 18% of respondents work at organizations with more than 10,000 employees; 20% work at organizations between 1,000 and 10,000 employees; and 25% work at organizations between 100 and 1,000 employees.
- 79% develop web applications or services; 49% develop enterprise business apps; and 24% develop native mobile applications.

THE HYPE

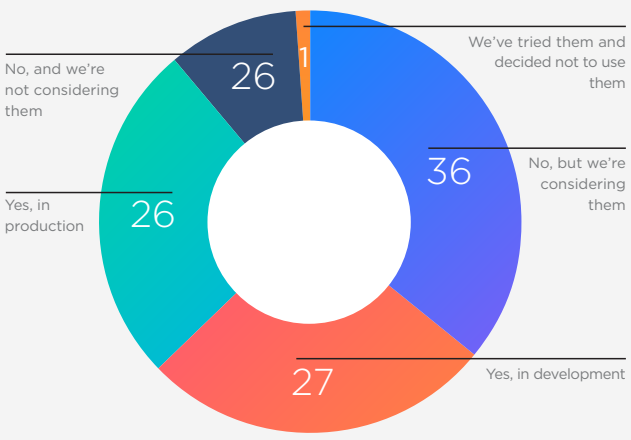
DZone's 2017 Microservices survey saw more than half of respondents claiming to be using microservices now: 27% of respondents said they use microservices in development environments, and 26% said they are using microservices in at least some of their applications in a production environment. Another 36% of respondents have not used microservices in any of their applications yet, but are interested in trying a microservices architecture. Only 10% of respondents said they have no interest in experimenting with microservices at all, and less than 1% of respondents said they have tried switching to microservices architecture in the past, but decided it wasn't for them.

Of the 53% of respondents who currently use microservices in one capacity or another, 81% said that using microservices architectures has made their job easier. And of all survey respondents, 75% said they believe that the excitement surrounding microservices in the current developer landscape is warranted–though interestingly, this sentiment is held slightly more amongst respondents interested but not currently using microservices (80%) than in the subset of respondents using microservices in production (76%) or in development (71%). Regardless, it seems that a large majority of devs are preparing (or have already prepared) to escape the monolith and see what it's like on the microservices bandwagon.

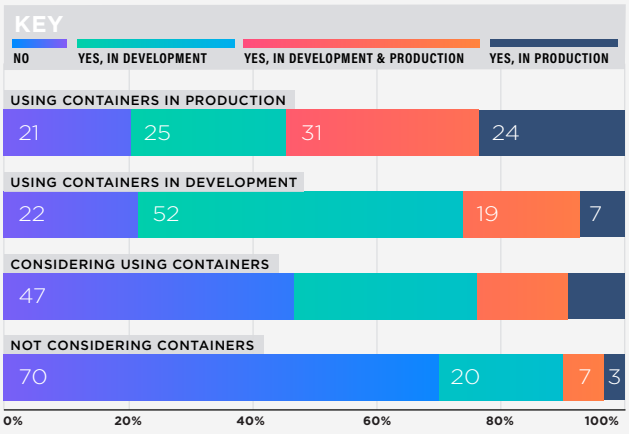
TOOLS, FRAMEWORKS, PROTOCOLS

By far, most respondents (81%) said that one of the best languages for designing microservices is Java (considering the general bias of DZone readers towards Java, this is unsurprising). Other favored languages for microservices were Node.js (35%), Python (29%), and JavaScript (27%). Most respondents also said they use a framework to help build their microservices, with the majority using either Spring Boot (32%), Java EE (11%), or both (17%). Regarding protocols for microservice communication, HTTP was the go-to choice, with 82% of respondents using it,

ARE YOU CURRENTLY USING A MICROSERVICES ARCHITECTURE FOR ANY OF YOUR APPLICATIONS?



MICROSERVICES AND CONTAINER USAGE:
ARE YOU CURRENTLY USING A MICROSERVICES ARCHITECTURE FOR ANY OF YOUR APPLICATIONS?



but Apache Kafka (21%) and RabbitMQ (19%) had some traction as microservices communication protocols. Respondents mostly said they handled security for their microservices applications with OAuth2 (44%), JSON Web Tokens (43%), and user authentication (32%).

Other tools around microservices were used more sparingly. 74% of respondents said they did not use a platform for managing microservices in their application, and the supplied answer choices were unpopular (Linkerd was the most popular choice at 5%), while write-in choices were scattered. 58% of respondents said they do not use a service discovery tool, while 19% said they use Apache Zookeeper and 17% said they use Eureka. 72% said they don't use distributed tracing tools, with Zipkin (10%) coming in as the most popular over OpenTracing (8%).

MICROSERVICES AND OTHER PRACTICES

"Microservices" is a term you hear a lot these days if you're keeping an eye on developer-related content. And as we saw earlier, it seems like many developers support microservices architectures beyond considering "microservices" a mere buzzword. The actual use of microservices fits well with other current software development trends, and we saw this reflected in our survey results. For example, respondents who said they use microservices in production were more likely to say they use Continuous Delivery or DevOps processes (81%) than those using microservices in dev environments (75%), those merely interested in trying microservices (56%), and those uninterested in microservices completely (45%).

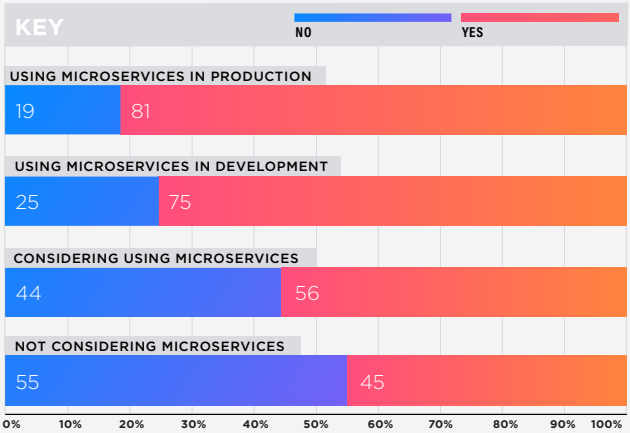
Likewise, the use of container technologies also correlates with the use of microservices technologies. Respondents who said they use microservices in dev environments were most likely to use containers in dev environments (71%), and those using microservices in production environments were most likely to use containers in production environments (55%), while those interested in microservices are most likely to not use container technologies (47%), as are those uninterested in exploring microservices (70%).

BENEFITS AND CHALLENGES

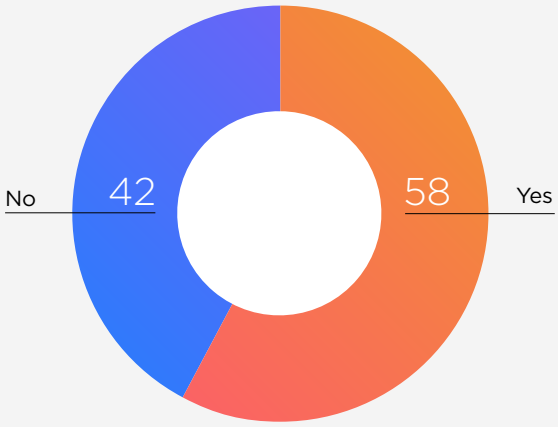
While a majority of respondents are on board with microservices one way or another, and most of these developers believe microservices make development easier and that the excitement surrounding microservices architechtures are indeed warranted, we still haven't looked at what benefits microservices can provide, and what challenges can be expected in implementing microservices. Regarding benefits, respondents using microservices now largely focused on these architecture's ability to make applications easily scalable (81%) and to enable faster deployments when only one part of an application needs to be altered (72%). Other common benefits seen were an improvement of software quality by allowing teams to focus their efforts on individual services in a larger application (50%), an improvement of software quality by narrowing failure sources to isolated application parts (42%), and flexibility regarding languages, frameworks, and tools for different pieces of a large application.

Of course, developing in a microservices architecture isn't without its challenges. When building microservices applications from scratch, respondents found that the biggest difficulties lied in monitoring an array of different services (47%), changing organizational culture to be open to trying microservices architectures (39%), and ensuring that services, as well as the entire application, are secure. The challenges involved with refactoring legacy applications to a microservices architecture included finding where to break up existing monolith components and tight coupling in existing software (each 51%, as these issues themselves are tightly coupled, as it were), and finding the time to invest in refactoring monolithic legacy applications.

MICROSERVICES AND CD/DEVOPS PROACTICES:
ARE YOU OR YOUR TEAM CURRENTLY DOING CONTINUOUS DELIVERY OR DEVOPS PROCESSES?



DO YOU FEEL THAT TOOLS AND FRAMEWORKS HAVE PROVIDED SUFFICIENT BEST PRACTICES FOR WORKING WITH MICROSERVICES?



CLOUD FOUNDRY

YOUR ENTERPRISE-READY, CLOUD-NATIVE TOOLBOX

Run apps in any language or framework on the clouds of your choice

Developers report significant time savings after moving to **Cloud Foundry**

cloudfoundry.org/why-cloud-foundry

46% LESS THAN **one week** development time

25% LESS THAN **24 hours** development time

Join us **April 18-20** in **Boston** for Cloud Foundry Summit.

Use promo code DZONE to be entered to win a Fitbit Surge.

Must register before January 11, 2018. Winner will be notified within 7 days.

Cloud Foundry Overcomes Operational Complexity for Microservices

Microservices add a layer of complexity to application development — we can agree on that. But they also enable choice, flexibility, and independence far more than developing within a monolithic architecture. Microservices enable small teams to work nimbly on a particular function apart from the larger team. You can iterate, release, and manage that function independently, giving you concentrated control.

If you use microservices to make your teams move faster and ship code more frequently, the path from writing code to production should be as quick as possible. Platforms play a big part in this, coupled with a Continuous Integration tool, which takes responsibility for testing the code, approving it for release, compiling, and publishing it.

Platforms like Cloud Foundry Application Runtime have evolved to support the operational complexity of microservices. It's necessary to have a platform like Cloud Foundry in place when you choose microservices to offset operational complexity. Both the Cloud Foundry Application Runtime and the Cloud Foundry Container Runtime manage the health of microservices. What does this mean? Your platform will automatically scale up your app when there's demand, watch for unhealthy instances, describe services talking to each other, accept code being offered by Continuous Integration pipelines, and more.

It's necessary to have a platform like Cloud Foundry in place when you choose microservices to offset operational complexity.

You don't need to use a microservices architecture to get value out of Cloud Foundry — you can certainly run monolithic applications inside it too, and it will get the same benefits. But if you choose a microservices architecture, you will need a platform like Cloud Foundry to avoid operational complexity and focus on shipping great software.

WRITTEN BY CHIP CHILDERS
CTO, **CLOUD FOUNDRY FOUNDATION**

PARTNER SPOTLIGHT

Cloud Foundry

CLOUD FOUNDRY

Cloud Foundry gives you the right tool for the right job with two complementary open source technologies for app developers and operators.

CATEGORY
Platform-as-a-Service (PaaS) or "cloud application platform"

NEW RELEASES
Continuous

OPEN SOURCE
Yes

STRENGTHS

- Polyglot and multi-cloud
- Run apps at scale
- Simplify the development lifecycle and container management

CASE STUDY

Insurance giant Liberty Mutual knew it needed to make a radical change. CIO Mojgan Lefebvre explained, "We knew we had to become a software company that sells insurance to survive in today's competitive world." After adopting Cloud Foundry, the team went from hypotheticals to standing up a minimum viable product (MVP) in just 28 days. Embracing agile methodologies and taking a cloud-native approach by deploying Cloud Foundry, the team created a fully functional portal that was ready within six months and provided:

- 40% strike rate against 20% for industry on-average
- Referral time of only three minutes – more than 3x less than competitor products
- 200 quotes and 60 policies within one month of operation

Cloud Foundry's flexibility, agility, and scalability enabled Liberty Mutual to move closer to its commitment to digital transformation.

NOTABLE USERS

- Allstate
- American Airlines
- Cloud.gov
- Ford
- The Home Depot

WEBSITE cloudfoundry.org | **TWITTER** @cloudfoundry | **BLOG** cloudfoundry.org/blog

Microservices on the JVM with Actors

QUICK VIEW

01 Building an individual service is easy. Building a system out of many is the real challenge.

02 The applicability of actors to the challenges of modern computing systems and microservices-based systems has been recognized and proven to be effective.

03 Designing a system with the assumption that messages can be lost in the network is the safest way to build a microservices-based architecture.

04 The actor model provides a higher level of abstraction for writing concurrent and distributed systems.

BY **MARKUS EISELE**
DIRECTOR OF DEVELOPER ADVOCACY, **LIGHTBEND**

As mobile and data-driven applications increasingly dominate, users are demanding real-time access to everything everywhere. System resilience and responsiveness are no longer "nice to have;" they're essential business requirements. Businesses increasingly need to trade up from static, centralized architectures in favor of flexible, distributed, and elastic systems. But where to start and which architecture approach to use is still a little blurry, and the microservices hype is only slowly settling while the software industry explores various architectures and implementation styles.

For a decade or more, enterprise development teams have built their Java EE projects inside large, monolithic application server containers without much regard to the individual lifecycle of their module or component. Hooking into startup and shutdown events was simple, as accessing other components was just an injected instance away. It was comparably easy to map objects into single relational databases or connect to other systems via messaging. One of the greatest advantages of this architecture was transactionality, which was synchronous, easy to implement, and simple to visualize and monitor. By keeping strong modularity and component separation a first-class priority, it was manageable to implement the largest systems that still power our world. Working with compartmentalization and introducing modules belongs to the core skills of architects. Our industry has learned how to couple services and build them around organizational capabilities. The new part in microservices-based architectures is the way truly independent services are distributed and connected back together. Building an individual service is easy. Building a system out of many is the real challenge, because it introduces us to the problem space of distributed systems. This is the major difference from classical, centralized infrastructures.

THERE ISN'T JUST ONE WAY OF DOING MICROSERVICES

There are many ways to implement a microservices-based architecture on or around the Java Virtual Machine (JVM). The pyramid in Figure 1 was introduced in my first book. It categorizes some technologies into layers, which can help identify the level of isolation that is needed for a microservices-based system. Starting at the virtualization infrastructure with virtual machines and containers, as they are means of isolating applications from hardware, we go all the way up the stack to something that I summarize under the name "application services." This category contains specific microservices frameworks aimed at providing microservices support across the complete software development lifecycle.

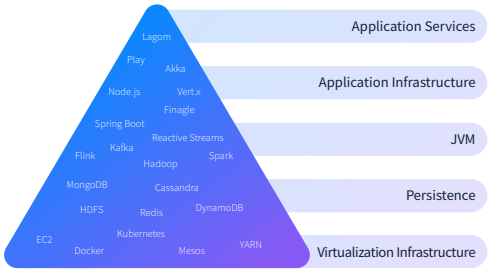


Figure 1: Pyramid of modern enterprise java development (Source: Modern Java EE Design Pattern, Eisele)

The three frameworks in the application services and infrastructure categories are all based on the principles of the Reactive Manifesto. It defines traits that lead to large systems that are composed of smaller ones, which are more flexible, loosely-coupled, and scalable. As they are essentially message-driven and distributed, these frameworks fit the requirements of today's microservices architectures. While Lagom offers an opinionated approach on close guardrails that only support microservices architectures, Play and Akka allow you to take advantage of the reactive traits to build a microservices-style system but doesn't limit you to this approach.

MICROSERVICES WITH AKKA

Akka is a toolkit and runtime for building highly concurrent, distributed, and resilient message-driven applications on the JVM. Akka "actors" are one of the tools in the Akka toolkit that allow you to write concurrent code without having to think about low-level threads and locks. Other tools include Akka Streams and Akka HTTP. Although Akka is written in Scala, there is a Java API, too.

Actors were invented decades ago by Carl Hewitt. But, relatively recently, their applicability to the challenges of modern computing systems has been recognized and proven to be effective. The actor model provides an abstraction that allows you to think about your code in terms of communication, not unlike people in a large organization. Systems based on the actor model using Akka can be designed with incredible resilience. Using supervisor hierarchies means that the parental chain of components is responsible for detecting and correcting failures, leaving clients to be concerned only about what service they require. Unlike code written in Java that throws exceptions, clients of actor-based services never concern themselves with dealing with failures from the actor from which they are requesting a service. Instead, clients only must understand the request-response contract that they have with a given service, and possibly retry requests if no response is given in some time frame. When people talk about microservices, they focus on the "micro" part, saying that a service should be small. I want to emphasize that the important thing to consider when splitting a system into services is to find the right boundaries between services, aligning them with bounded contexts, business capabilities, and isolation requirements. As a result, a microservices-based system can achieve its scalability and resilience requirements, making it easy to deploy and manage.

The best way to understand something is to look at an example. The Akka documentation contains an extensive walkthrough of a simplistic IoT management application that allows users to query sensor data. It does not expose any external API to keep things simpler, only focuses on the design of the application, and uses an actor-based API for devices to report their data back to the management part. You can find a high-level architecture diagram in Figure 2.

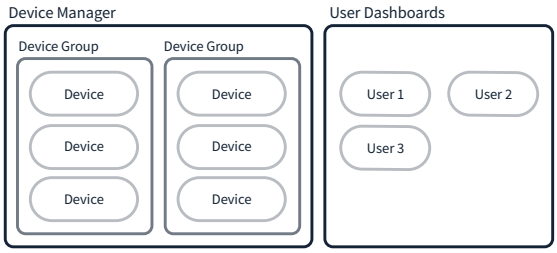


Figure 2: IoT sample application architecture (Source: Akka documentation)

Actors are organized into a strict tree, where the lifecycle of every child is tied to the parent, and where parents are responsible for deciding the fate of failed children. All you need to do is to rewrite your architecture diagram so that it contains nested boxes into a tree, as shown in Figure 3. In simple terms, every component manages the lifecycle of the subcomponents. No subcomponent can outlive the parent component. This is exactly how the actor hierarchy works.

Furthermore, it is desirable that a component handles the failure of its subcomponents. A "contained-in" relationship of components is mapped to the "children-of" relationship of actors. If you look at microservice architectures, you would have expected that the top-level components are also the top-level actors. That is indeed possible, but not recommended. As we don't have to wire the individual services back together via external protocols and the Akka framework also manages the actor lifecycle, we can create a single top-level actor in the actor system and model the main services as children of this actor. The actor architecture is built on the same traits that a microservice architecture should rely on, which are isolation, autonomy, single responsibility, exclusive state, asynchronous communication, explicit communication protocols, and distribution and location transparency.

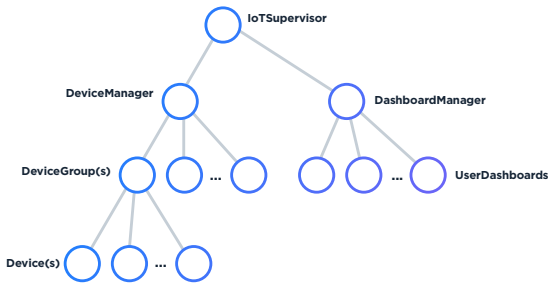


Figure 3: An Actor representation of the IoT architecture.

You find the details about how to implement the IoTSupervisor and DeviceManager classes in the official Akka tutorial. Until now, I only looked at the complete system at large. But there is also the individual actor that represents a device. His simple task will be to collect temperature measurements and report the last measured data back on request. When working with objects, you usually design APIs as interfaces, which are basically collections of abstract methods to be filled out by the actual implementation. In the world of actors, the counterparts of interfaces are protocols. The protocol in an actor-based application is the message for the devices.

```
function counter(state: AppState = 0, action: AppAction): public
static final class ReadTemperature {
  long requestId;

  public ReadTemperature(long requestId) {
    this.requestId = requestId;
  }
}

public static final class RespondTemperature {
  long requestId;
  Optional<Double> value;

  public RespondTemperature(long requestId, Optional<Double>
  value) {
    this.requestId = requestId;
    this.value = value;
  }
}
```

(Code 1: message protocol for the device actor)

I am skipping a lot of background on message ordering and delivery guarantees. Designing a system with the assumption that messages can be lost in the network is the safest way to build a microservices-based architecture. This can be done, for example, by implementing a "re-send" functionality if a message gets lost. And this is the reason why the message also contains a `requestId`. It will now be the responsibility of the querying actor to match requests to actors. A first rough sketch of the Device Actor is below.

```
class Device extends AbstractActor {
//…
Optional<Double> lastTemperatureReading = Optional.empty();

  @Override
  public void preStart() {
    log.info("Device actor {}-{} started", groupId, deviceId);
  }
 @Override
  public void postStop() {
```

Code continued on next page

```
    log.info("Device actor {}-{} stopped", groupId, deviceId);
  }

  @Override
// react to received messages of ReadTemperature
  public Receive createReceive() {
    return receiveBuilder()
            .match(ReadTemperature.class, r -> {
              getSender().tell(new RespondTemperature(r.
              requestId, lastTemperatureReading), getSelf());
            })
            .build();
  }
}
```

(Code 2: the device actor)

The current temperature is initially set to `Optional.empty()`, and simply reported back when queried. A simple test for the device is shown below.

```
@Test
public void testReplyWithEmptyReadingIfNoTemperatureIsKnown() {
  TestKit probe = new TestKit(system);
  ActorRef deviceActor = system.actorOf(Device.props("group",
"device"));
  deviceActor.tell(new Device.ReadTemperature(42L), probe.
getRef());
  Device.RespondTemperature response = probe.
expectMsgClass(Device.RespondTemperature.class);
  assertEquals(42L, response.requestId);
  assertEquals(Optional.empty(), response.value);
}
```

(Code 3: Testing the device actor)

The complete example if the IoT System is contained in the Akka documentation.

WHERE TO GET STARTED

Most of today's enterprise software was built years ago and still undergoes regular maintenance to adopt the latest regulations or new business requirements. Unless there is a completely new business case or significant internal restructuring, the need to re-construct a piece of software from scratch is rarely given. If this is the case, it is commonly referred to as "greenfield" development, and you are free to select the base framework of your choice. In a "brownfield" scenario, you only want to apply the new architecture to a certain area of an existing application. Both approaches offer risks and challenges and there are advocates for both. The common ground for both scenarios is your knowledge of the business domain. Especially in long-running and existing enterprise projects, this might be the critical path. They tend to be sparse on documentation, and it is even more important to have access to developers who are working in this domain and have firsthand knowledge.

The first step is an initial assessment to identify which parts of an existing application can take advantage of a microservices architecture. There are various ways to do this initial assessment. I suggest thinking in service characteristics. You want to identify either core or process services first. While core services are components modeled after nouns or entities, the process services already contain complex business or flow logic.

SELECTIVE IMPROVEMENTS

The most risk-free migration approach is to only add selective improvements. By scraping out the identified parts into one or more services and adding the necessary glue to the original application, you're able to scale out specific areas of your application in multiple steps.

THE STRANGLER PATTERN

First coined by Martin Fowler as the Strangler Application, the extraction candidates are move into a separate system which adheres to a microservices architecture, and the existing parts of the applications remain untouched. A load balancer or proxy decides which requests need to reach the original application and which go to the new parts. There are some synchronization issues between the two stacks. Most importantly, the existing application can't be allowed to change the microservices' databases.

BIG BANG: REFACTOR AN EXISTING SYSTEM

In very rare cases, complete refactoring of the original application might be the right way to go. It's rare because enterprise applications will need ongoing maintenance during the complete refactoring. What's more, there won't be enough time to make a complete stop for a couple of weeks—or even months, depending on the size of the application—to rebuild it on a new stack. This is the least recommended approach because it carries a comparably high risk of failure.

WHEN NOT TO USE MICROSERVICES

Microservices are the right choice if you have a system that is too complex to be handled as a monolith. And this is exactly what makes this architectural style a valid choice for enterprise applications.

As Martin Fowler states in his article about "Microservice Premium," the main point is to not even consider using a microservices architecture unless you have a system that's too large and complex to be built as a simple monolith. But it is also true that today, multicore processors, cloud computing, and mobile devices are the norm, which means that all-new systems are distributed systems right from the start. And this also results in a completely different and more challenging world to operate in. The logical step now is to switch thinking from collaboration between objects in one system to a collaboration of individually scaling systems of microservices.

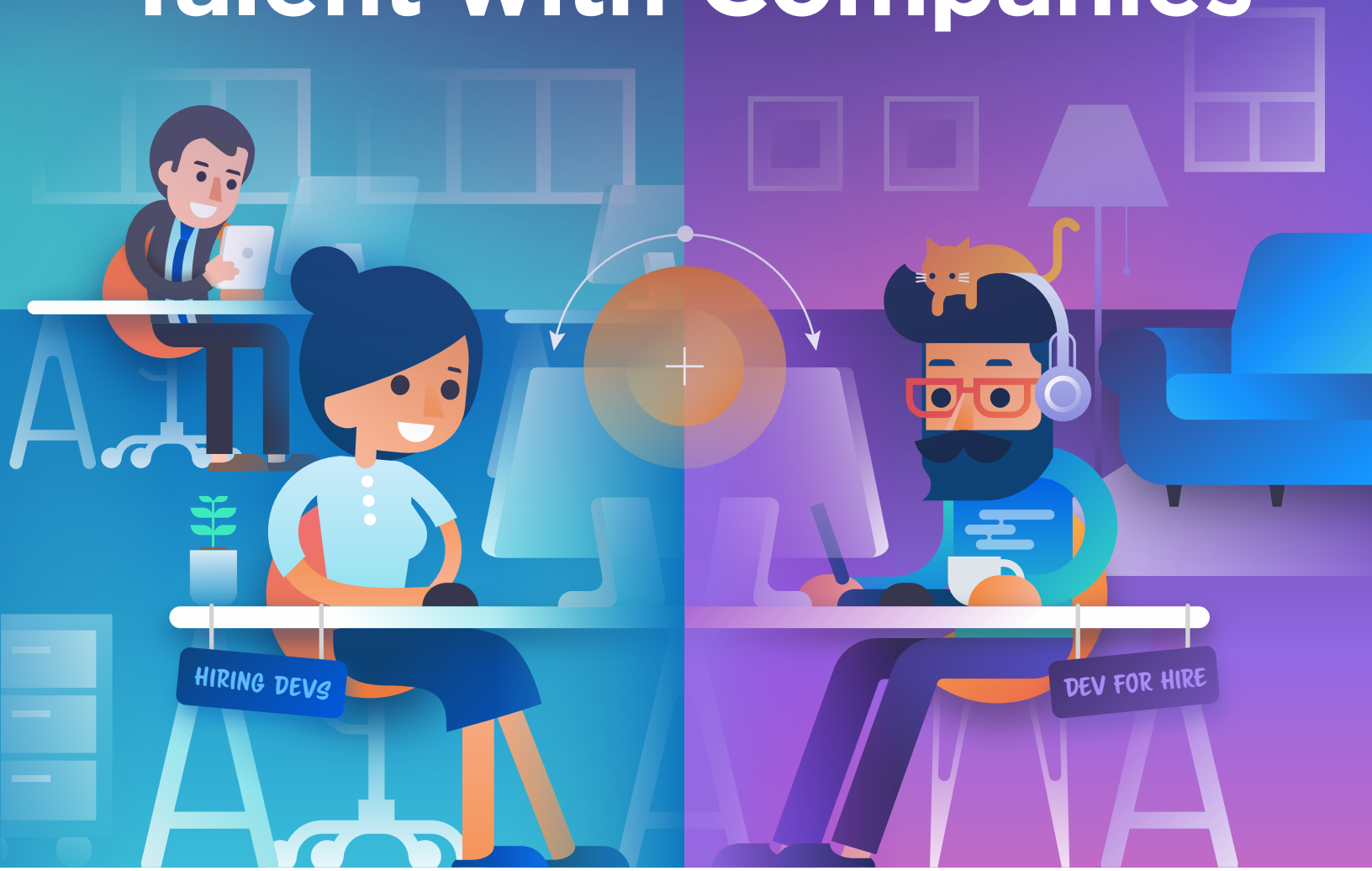
SUMMARY

The actor model provides a higher level of abstraction for writing concurrent and distributed systems, which shields the developer from explicit locking and thread management. It provides the core functionality of reactive systems, defined in the Reactive Manifesto as responsive, resilient, elastic, and message-driven. Akka is an actor-based framework that is easy to implement with full Java 8 Lambda support. Actors enable developers to design and implement systems in ways that help focus more on the core functionality and less on the plumbing. Actor-based systems are the perfect foundation for quickly evolving microservices architectures.

Markus Eisele leads the developer advocate team at Lightbend, Inc. He has been working with Java EE servers from different vendors for more than 14 years, and gives presentations on his favorite topics at leading international Java conferences. He is a Java Champion, former Java EE Expert Group member, and founder of JavaLand. He is excited to educate developers about how microservices architectures can integrate and complement existing platforms. He is also the author of "Modern Java EE Design Patterns" and "Developing Reactive Microservices" by O'Reilly. You can follow more frequent updates on Twitter @myfear.

DZone Jobs

Connecting Top Tech Talent with Companies



THESE COMPANIES ARE HIRING!

amazon Sonatype Microsoft Booking.com Couchbase

Streamlined Microservice Design in Practice

QUICK VIEW

01 Design thinking is fundamental for a sustainable microservice architecture.

02 Event Storming is an effective, collaborative approach to identify bounded contexts for microservices.

03 The Microservice Design Canvas is a useful tool for designing service characteristics without over-burdening the development lifecycle.

BY **MATT MCLARTY** & **IRAKLI NADAREISHVILI**
VP API ACADEMY, **CA** & SENIOR DIRECTOR OF TECHNOLOGY, **CAPITAL ONE**

As more organizations implement microservices, the practices of microservice architecture become more mature. Whereas much of the early microservices literature focused on companies decomposing a monolithic web application into microservices, larger and more diverse organizations are now tackling how to migrate their existing software ecosystems into domains of services in order to improve their software delivery speed and scalability. This problem space is significantly more complex than breaking down the single monolith, and comes with higher order challenges.

Modularization is fundamental to dealing with the complexity of distributed software systems. This is both the reason microservice architecture is gaining popularity, and an important reminder of how to approach it. Finding the right boundaries between services is understandably one of the main focus areas for organizations adopting microservices in order to reduce coordination between teams, and there is a growing body of information on techniques to draw those boundaries. This technology-agnostic design work deals with the essential complexity of the software system, helping to improve its evolvability and sustainability over time. Once the boundaries are drawn, there is still design work that needs to be done.

THE MICROSERVICE DESIGN CANVAS

The microservices movement has been driven by developers, is closely aligned with the rise of Agile methods and DevOps, and has been motivated by a desire for faster software delivery. Consequently, developers often start coding quickly and rely on emergent design to guide their work, which can result in sub-optimal service disposition over the long haul. On the other hand, an overly-involved service design process can bog down development efforts and undermine the intended benefits of microservice architecture. How can appropriate design thinking be injected into the process in a streamlined way?

With a hat tip to Simon Brown's "just enough up front design" concept, the Microservice Design Canvas intends to capture the essential service attributes that can help guide development of the service itself as well as its consuming applications.

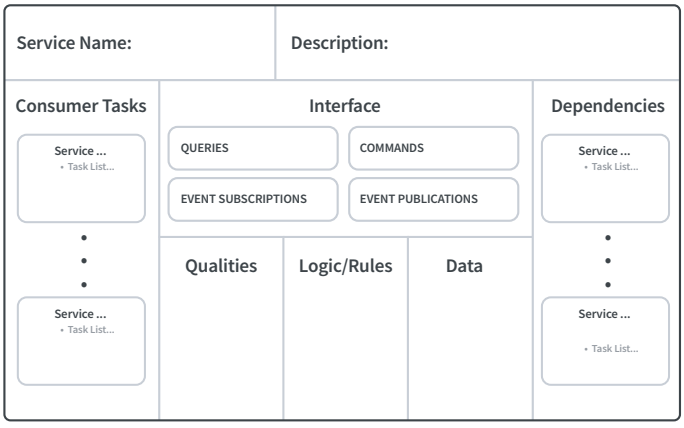


Figure 1 - *The Microservice Design Canvas*

In addition to the name and description of the service, the canvas includes the following sections:

SECTION	DESCRIPTION
Consumer Tasks	An enumeration of the anticipated consumers of the service along with the tasks they need to perform that require the service
Interface	A list of interactions that consumers are expected have with the service, broken down by interaction type (Query, Command, Event Subscription, Event Publication)
Qualities	The fundamental non-functional attributes of the service, such as specific security, availability, reliability, scalability, and evolvability requirements
Logic/Rules	A select list of processing logic that will be required to satisfy the Interface and Qualities sections, not an exhaustive list of the service's functionality
Data	A select list of data elements required to support the Interface and Qualities sections
Dependencies	The external services upon which this service depends (with the understanding that service dependencies should be minimized)

Ideally, a service designer can complete sections in the table's order and capture the essence of the service using the canvas. Here is an example of a completed canvas for a Transaction Search Service:

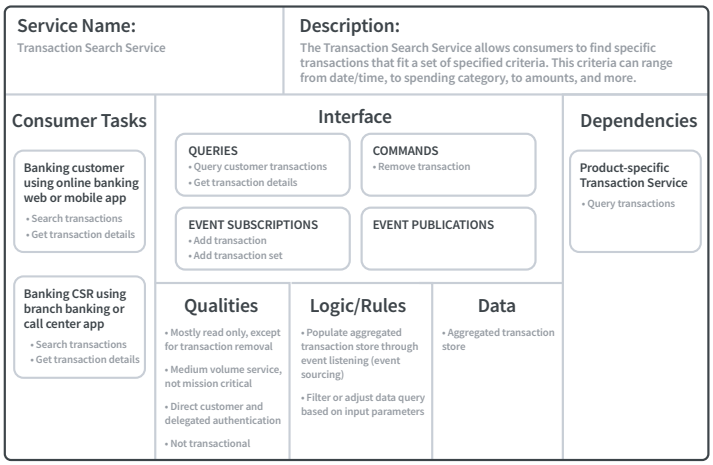


Figure 2 - *A sample microservice design canvas for "Transaction Search"*

MICROSERVICES AT CAPITAL ONE

Capital One has been an early adopter of microservice architecture, with several hundred now running in production across the company. Capital One's technology is large and heterogeneous, not all of which is built using microservices. The initial rise of microservices adoption happened organically, by different teams experimenting with and adopting its concepts as they saw fit in their daily work. In 2017, the Capital One executive team declared maturation of microservices capabilities an important priority, and a team was put together to provide a microservices adoption strategy, implementation guidelines, and training workshops for development teams.

"When analyzing the success of the organic microservices efforts, the Capital One microservices team recognized that the power of the new architecture was that it allowed developers to move fast without compromising the safety and quality of their solutions."

To align their microservices efforts across the organization, Capital One first needed to identify a common goal. When analyzing the success of the organic microservices efforts, the Capital One microservices team recognized that the power of the new architecture was that it allowed developers to move fast without compromising the safety and quality of their solutions. Irakli Nadareishvili, one of the leaders of the team, explains:

> *For the longest time, there has been a belief in software engineering that you have to compromise between speed and safety: either you go fast or you build with high quality. Such a compromise makes intuitive sense. Complex systems are built by many teams, working on different parts of an application. Every now and then those teams need to coordinate their work with others, and at that point you have one of two choices: you either ignore coordination need and keep going fast, which may break some things along the way, or you acknowledge the need to coordinate and slow down. But what if we had a system architected in a way that minimized the need for coordination? Then we wouldn't need to choose between speed and safety as often. It turns out you can have such a design if you have autonomous teams working on small batches of isolated work. For us, that is the essence of microservice architecture.*

DESIGNING MICROSERVICES AT CAPITAL ONE

Many organizations get stuck when trying to find the size for the microservices. In Capital One's analysis of microservice design, they found that the optimal microservice size varies over time, as illustrated by microservice pioneers like Netflix:

At Capital One, development teams start with coarse-grained design and split microservices when they find need to eliminate emerging instances of coordination. Teams are not expected to get service boundaries "right" out of the gate. Instead, boundaries evolve over time to allow autonomous, high-performance teams to develop systems quickly and safely.

When examining how to design greenfield or brownfield systems of microservices, the Capital One microservices team used Domain Driven Design (DDD) as a starting point. They found the concept of bounded contexts to be extremely useful for representing autonomous capabilities in a complex system. Overall, however, they felt that applying DDD in depth across the organization would require expertise and experience that most software teams were not equipped with, and trying to make it work would be costly and difficult to implement consistently.

Capital One found a viable shortcut to DDD in Alberto Brandolini's Event Storming methodology. This new approach that is rapidly gaining popularity in the software industry allows teams to explore a complex domain—including the identification of bounded contexts—in just a handful of 4-hour sessions. In addition to its work products, Capital One has found Event Storming to be a collaborative and inclusive exercise that helps quickly develop a shared understanding of a product between engineering, product teams, design teams, as well as other key stakeholders.

THE MICROSERVICE DESIGN CANVAS AT CAPITAL ONE

One issue the Capital One team encountered with Event Storming is that, while the process is very useful, its final artifact—a wall full of sticky notes—is difficult to digitize or document. Since they wanted something more than just a list of bounded contexts and hotspots as a takeaway, they decided to codify the resulting microservice designs using a variant of the Microservices Design Canvas. Team member James Higginbotham re-ordered the boxes on the canvas to align more closely with the Business Model Canvas, resulting in the following:

Name: Payments Management Service	**Description:** This service allows consumers to sign up and administer the preferences for the "customer centric payments" product.	
Consumer Tasks/Capabilities:	• Sign up for payments service • Opt out of payments service	
Implementation	**Dependencies**	**Interface**
Qualities • Audited • Low-volume • Non-critical	**Service Dependencies** • Customer Information Service: • Obtain list of customer accounts and products	**Queries** • (queries)
Logic • Min accounts/products required for signup • Role-based permissions		**Commands** • (commands)
Data • Customer signup status • Customer preferences	**Events Subscriptions** • (none)	**Events Published** • (events published)

Figure 3 - *A Sample Microservice Design Canvas for "Payments Management Service" using the Capital One variant (information purely for demonstration purposes)*

So far, the Capital One team has found the canvas to be a useful way of documenting the design of their microservices. Importantly, they are able to use the canvas in a non-intrusive way that helps them reduce coordination between teams in order to improve their overall delivery speed without compromising the safety and stability of their systems.

DESIGN THINKING IN MICROSERVICE ARCHITECTURE

Just as Capital One recognized that microservice boundaries evolve over time, the structure of the Microservice Design Canvas will also change as it is applied and adapted by individuals and organizations. Its value should be measured by how effectively it meets its goal: to provide a simple tool for capturing just enough design thinking at the right time in order to help deal with the complexity of distributed software ecosystems. Experimentation and iteration are hallmarks of the microservices way, so please let the authors know about your own experiences working with the canvas and the other tools discussed in this article.

Matt McLarty is an experienced software architect who leads the API Academy for CA Technologies. He works closely with organizations on designing and implementing innovative, enterprise-grade API and microservices solutions. Matt has worked extensively in the field of integration and real-time transaction processing for software vendors and clients alike. Matt recently co-authored the O'Reilly book "Microservice Architecture" with other members of the API Academy team.

Irakli Nadareishvili is currently leading microservices transformation efforts as the Senior Director of Technology at Capital One. Irakli is a co-author of Microservice Architecture (O'Reilly 2016), and was formerly co-founder and CTO of ReferWell, a NY-based health technology startup. In the past he has also held technology leadership roles at CA Technologies and NPR.

An API-First Approach for Microservices on Kubernetes

BY **BORIS SCHOLL** & **CLAUDIO CALDATO**
VP PRODUCT DEVELOPMENT AND SENIOR DIR. OF PRODUCT STRATEGY, **ORACLE CLOUD**

QUICK VIEW

01 Microservices adoption is hard. Developers need to learn new patterns and new technologies such as Kubernetes and Docker.

02 A Kubernetes services creates a persistent IP address and DNS name entry that points to the actual microservice code that you develop.

03 Besides other advantages, an API-first approach allows you to up-level the discussion for most of your developers, so that they do not need to understand the inner workings of K8s right away.

Moving to containerized microservices is not an easy transition for developers that have been building applications using more traditional methods. There are a lot of new concepts and details developers need to consider and become familiar with when they design a distributed application, which is what a microservice application is. Throw Docker and Kubernetes into the mix and it becomes clear why many developers struggle to adapt to this new world. Developers want to focus on the development of the logic, not on the code necessary to handle the execution environment where the microservice will be deployed. APIs have always been a productive way to connect services, and this is still true for microservices on Kubernetes (K8s). In this article, we will lay out why you can benefit from an API-first approach for building microservices applications on Kubernetes. Before we can dive into the how let's have a quick review on what API-first means and what one commonly refers to services in K8s.

WHAT DOES API-FIRST MEAN?

This previous DZone article describes what API-first means: you first start designing and implementing an API that can be consumed by other microservices before you actually start implementing the actual microservice itself. Along with the API design itself, you typically provide mocks and documentation for an API. Those artifacts are then used to facilitate discussions with other teams that will be consumers of the microservice that your team is planning to build. In other words, the approach allows you to validate your API design before investing too much in writing the actual microservice. However, an API-first approach is not just useful during the development phase. Once a microservice has been built, other teams who want to consume the microservice will benefit from the documentation and mocking capabilities. The good news is that there are plenty of tools available that support an API-first approach. The most common specifications to support an API-first approach are OpenAPI and API Blueprint. You can then use tools like Swagger or Apiary to design your API, generate mocks, documentation, and even client libraries.

> "[API-first design] becomes particularly useful for applications that require independence and loose coupling, such as microservices applications, as it helps teams be more productive when it comes to consuming services built by other teams."

All of this becomes particularly useful for applications that require independence and loose coupling, such as

microservices applications, as it helps teams be more productive when it comes to consuming services built by other teams. But, how does this approach translate to a modern microservice architecture that relies on an orchestrator, such as Kubernetes, to handle the deployment and execution of each microservice? Before explaining this approach, it is worthwhile to recap what K8s services are.

WHAT ARE SERVICES IN K8S?

As mentioned earlier, a lot of developers are a bit overwhelmed with all the new concepts they need to learn. For developers who are new to K8s, the concept of a K8s service is very confusing as it does not technically relate to the microservice's code itself. Below is an example of a K8s service:

```
apiVersion: v1
kind: Service
metadata:
  name: githubstats
  labels:
    app: githubstats
spec:
  ports:
  - port: 9000
    name: http
  selector:
    app: githubstats
```

As you can see, a K8s service has nothing to do with the microservice you develop. In fact, it is just an endpoint with a port number that provides information on how to access your microservice inside a pod.

Under the cover, a K8s service creates a persistent IP address and DNS name entry so that the targeted microservice can always be reached.

What is missing to achieve a real "API-first" approach in the context of microservice architectures is to include the logic that makes it possible for the generated code to discover, at runtime, where the service is running.

K8s uses label selectors to know which pod the service needs to point to, in this example `app: githubstats`. The microservice you develop is typically packaged inside a container image and deployed to K8s. The example below shows the container image `repo/githubstats:0.0.1` as part of a deployment with the label `app: githubstats`.

```
apiVersion: apps/v1beta2 # for versions before 1.8.0 use
apps/v1beta1
kind: Deployment
metadata:
  name: githubstats
spec:
  replicas: 3
  selector:
    matchLabels:
      app: githubstats
  template:
    metadata:
      labels:
        app: githubstats
    spec:
      containers:
      - name: githubstats
        image: repo/githubstats:0.0.1
        ports:
        - containerPort: 9000
```

The real advantage of using a K8s service is that they provide a steady endpoint to access the microservice itself, no matter where the scheduler places it inside the cluster. It does not take a lot to see that by just looking at K8s services developers do not get the information they need in order to consume a microservice. Let's say the githubstats microservice, shown in the previous example, is developed by team A. Now another team, team B, is building a microservice, call it UI service, that is supposed to consume the githubstats microservice. The only information team B gets is the githubstats microservice name and the endpoint information. What is completely missing is information on the microservice itself, such as what methods one can call.

WHY YOU SHOULD USE AN API-FIRST APPROACH ON K8S

As mentioned in the beginning the big advantage of an API-first approach is that you always start with the API design, create mock services, documentation and client libraries. From a K8s perspective an API-first approach allows you to up level the discussion for most of your developers, so that they do not need to understand the inner working of K8s right away. To make this approach useful on K8s you need to somehow bind an API with K8s services. The rest of this article focuses on how you can approach this.

THE API-FIRST WORKFLOW

DESIGN

The first step of the process involves creating a "formal" description of the APIs. There are various format and tools that can be used. One, for instance, is Oracle's Apiary. The Apiary website offers an environment where

a team of developers can design and document APIs as shown in Figure 1.

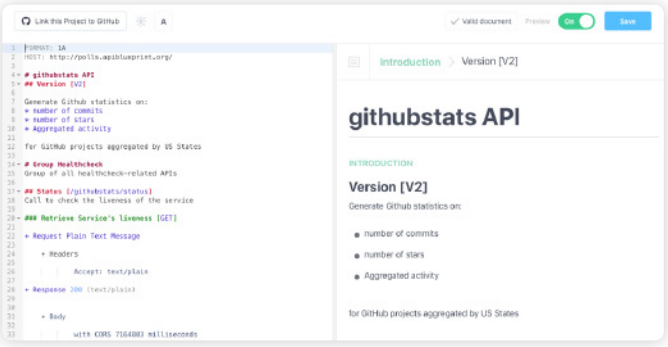


FIGURE 1: APIARY UI

Most of the time these tools are being used during "design time" of the overall workflow, where they are creating the APIs with some additional documentation. As microservices applications are highly dynamic in nature, it makes sense to also transfer the API-first part into the "runtime" part of the process when APIs are actually going to be used.

BIND

To accomplish that, you need to create a "bind" relationship that makes a K8s service more than just a hostname and TCP Port, as it is currently. By binding APIs to a K8s service, developers can get important information on the service right away without having to go through the extra hoops of finding the documentation for the APIs and write the code to process the request/response based on the schema defined in the APIs. This binding information can be kept in a simple data store with an UI on top of it as shown in Figure 2.

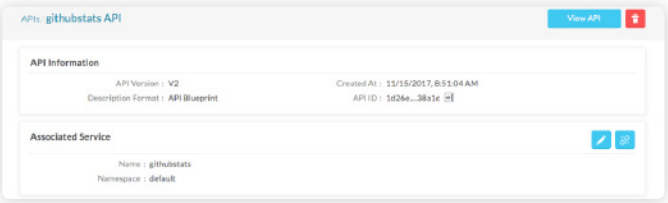


FIGURE 2: SIMPLE UI TO SHOW WHICH APIS ARE BOUND TO A K8S SERVICE

CONSUME

The last part of an API-first approach is how to consume the service. Ideally, developers would like to avoid having to implement parsing/marshalling of the response and add the code that can handle the HTTP calls. A more productive way is to provide a client library for a service, at least for the most common languages. In some more advanced organizations, client libraries can be generated as part of the CI (Continuous Integration) process. There are tools such as swagger–codgen that can be used to generate clients based on the specification and make it part of your CI process, or even include the client generation in your custom binding UI. What is missing to achieve a real "API-first" approach in the context of microservice architectures is to include the logic that makes it possible for the generated code to discover, at runtime, where the service is running. Having the ability to determine where the service is running at the time it is needed (when a service is making a call to a remote service) makes the API-First approach a better solution than existing best practices, where some aspects of the discovery phase of the process is hardcoded when the service is deployed.

> You can make an API-first approach part of your existing environment if you are willing to put a little effort into the "binding" and code generation experience.

CONCLUSION

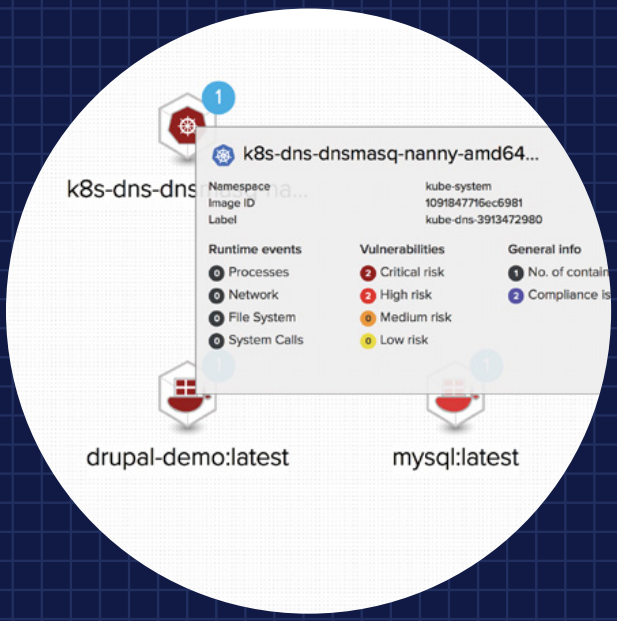
This article laid out how you can combine an API-first approach with K8s. You can make an API-first approach part of your existing environment if you are willing to put a little effort into the "binding" and code generation experience. The advantage is not only that developers can focus on writing code, while only a few need to understand the inner workings of K8s, but also that you fulfill some of the governance requirements you need to have in place for successful microservices projects, such as proper documentation and correct versions for APIs.

Boris Scholl leads engineering for the new container native microservices platform at Oracle. He has spent the last seven years of his career focusing on architectural and implementation patterns for large-scale distributed cloud applications, cloud developer tooling, and DevOps. Boris is a frequent speaker at events, and author of various articles and books on cloud development and microservices. His publications include the book *Microservices with Docker on Microsoft Azure*, released in June 2016 and a blog series about microservices.

Claudio Caldato is a Senior Director of Product Strategy in the Oracle Cloud Team where he is working on the Grand Unified Theory of Cloud-Native Development that will empower developers to build the next generation of cloud-native applications. Before Joining Oracle, he worked on the Azure Hyperscale and IoT Teams. He was one of the founding members of the team that pioneered OOS at Microsoft.

Twistlock

THE MOST COMPREHENSIVE CONTAINER SECURITY PLATFORM



RUNTIME DEFENSE

Automated, scalable active threat protection

VULNERABILITY MANAGEMENT

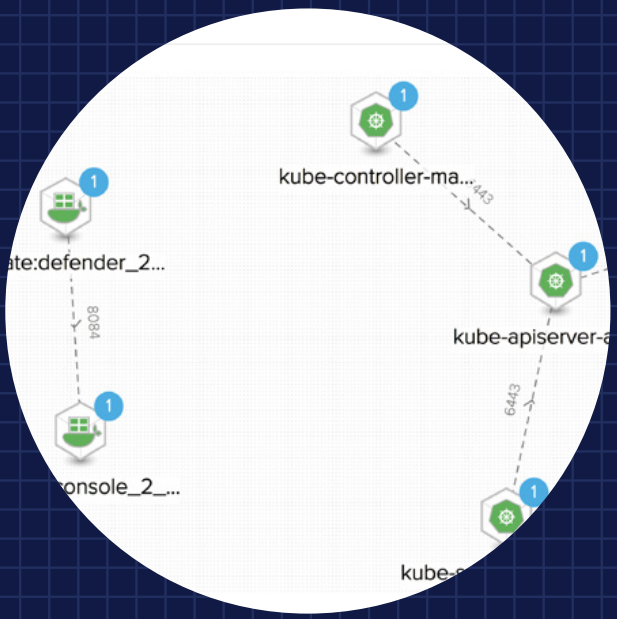
Precise controls to detect and prevent vulnerabilities before they reach production

COMPLIANCE

Extend and enforce your corporate compliance across your container environment

CLOUD NATIVE FIREWALLS

Protect your network from modern threats with layer 3 and layer 7 firewalls



TO LEARN MORE, VISIT...

Twistlock.com

Providing Security to Microservices

The increasing use of container-based microservice deployments highlights the need for new approaches to securing them. In order to work effectively, teams who develop services require lightweight infrastructure and unobtrusive security.

Traditional service oriented architectures can encounter many challenges, such as latency, scaling, high resource consumption with virtual machines, and lack of resiliency to recover from failures. The past few years have seen a transition from siloed development and IT departments to DevOps organizations. Along with that transition, we've also seen a migration from traditional SOA models to container and microservices-based Continuous Integration and Continuous Delivery (CI/CD) production environments.

Security tools must be completely automated, built into the CI/CD workflow, and both detect vulnerabilities at build time and protect the container environment at runtime. DevOps teams containerize large applications by functionally decomposing them into services. The microservices are deployed as containers onto a cluster, and are automatically scaled to meet demand. The cluster spans multiple hosts, and is also scaled to meet demand.

The current wave of container-based microservice deployments is creating the need for new approaches to cloud security.

Developer container security solutions could use each container's origin image to profile how the container should interact with its environment. A security tool could monitor the fairly limited types of interactions microservices have, and use that as a baseline to detect unusual patterns. These two strategies are possible because containers are minimalistic, the images they run are declarative, and deployed containers are predictable.

WRITTEN BY JOHN MORELLO
CHIEF TECHNOLOGY OFFICER, **TWISTLOCK**

PARTNER SPOTLIGHT

Twistlock Container Cybersecurity Platform

Twistlock is the leading cloud native cybersecurity platform for the modern enterprise

CATEGORY
Container and Cloud Native Cybersecurity

NEW RELEASES
6x year

OPEN SOURCE
No

CASE STUDY

ClearDATA provides secure, managed services for healthcare on AWS. ClearDATA customers must comply with significant regulatory requirements, have huge amounts of sensitive data to manage, and customers demanding better data collaboration.

In order to help their clients deliver solutions faster, ClearDATA wanted to deliver a new set of product and service offerings to allow health organizations to run Docker containers using AWS' EC2 Container Services (ECS).

Using Twistlock in their environment has enabled ClearDATA to monitor and enforce compliance requirements, check for vulnerabilities from development through production, and automate runtime defense that scales within the ECS environment.

STRENGTHS

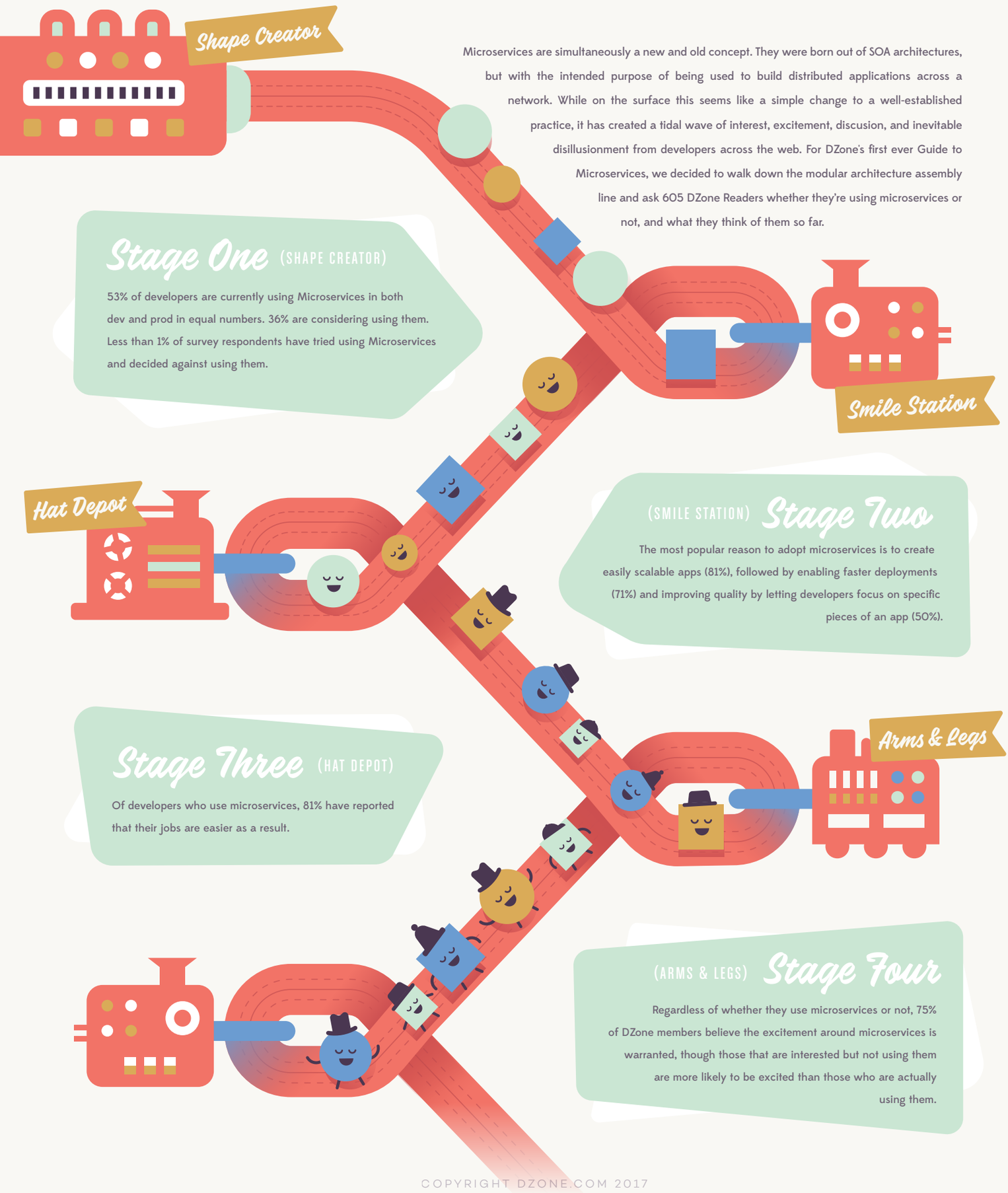
- **Runtime Defense**
  Automatically prevent next gen attacks against containers and cloud native apps
- **Vulnerability Management**
  Detect and prevent vulnerabilities before they make it to production
- **Compliance**
  Extend regulatory and corporate compliance into your container environment
- **Cloud Native Application Firewall**
  Automatically protect your apps in a 'software defined' manner
- **Continuous Integration**
  Integrate with any CI tools to leverage automated security throughout the SDLC

NOTABLE CUSTOMERS

- ClearDATA
- Booz | Allen | Hamilton
- AppsFlyer
- Aetna

WEBSITE twistlock.com | **TWITTER** @twistlockteam | **BLOG** twistlock.com/blog

Big Things COME IN MICROSERVICES



Microservices are simultaneously a new and old concept. They were born out of SOA architectures, but with the intended purpose of being used to build distributed applications across a network. While on the surface this seems like a simple change to a well-established practice, it has created a tidal wave of interest, excitement, discusion, and inevitable disillusionment from developers across the web. For DZone's first ever Guide to Microservices, we decided to walk down the modular architecture assembly line and ask 605 DZone Readers whether they're using microservices or not, and what they think of them so far.

Stage One (SHAPE CREATOR)

53% of developers are currently using Microservices in both dev and prod in equal numbers. 36% are considering using them. Less than 1% of survey respondents have tried using Microservices and decided against using them.

(SMILE STATION) Stage Two

The most popular reason to adopt microservices is to create easily scalable apps (81%), followed by enabling faster deployments (71%) and improving quality by letting developers focus on specific pieces of an app (50%).

Stage Three (HAT DEPOT)

Of developers who use microservices, 81% have reported that their jobs are easier as a result.

(ARMS & LEGS) Stage Four

Regardless of whether they use microservices or not, 75% of DZone members believe the excitement around microservices is warranted, though those that are interested but not using them are more likely to be excited than those who are actually using them.

Microservices and Team Organization

QUICK VIEW

01 Microservices architectures require regular changes that can impact organizations.

02 Some tricks can help to solve microservices organization challenges, but may not be enough to solve them all.

03 Microservices architectures require a high coordination of knowledge and skills, and require organizations that allow this coordination.

04 New management methods have emerged that align architectural and organizational needs.

BY **THOMAS JARDINET**
SENIOR CONSULTANT, **ASTRAKHAN CONSULTING**

Microservices architectures are unique because they can be extremely flexible over time and impact a project's organization at any time. This can be very challenging for companies, as it can force them to question their organizational model, which may not have necessarily moved much in most companies. Perhaps the first good question to ask yourself when you start using this kind of architecture is: "what is your organization capable of?" In my opinion, this is a prerequisite to know what difficulties will be encountered, and to start arming yourself in the early stages.

But let's first come back to the link between architectures and microservices. When it comes to organizing teams and microservices, the famous Conway law is often mentioned. This law, which is becoming more and more widely accepted, has not always been approved of in the past. The main flaw in the attainment of the absolute truth of this law is that it is more a sociological law than a purely scientific law. Indeed, it has always been demonstrated in an empirical way, based on examples and not on pure scientific logic. It is difficult to demonstrate sociological results in general, because these demonstrations are largely based on intangible considerations and on concepts, and can only be verified by multiplying the examples to infinity.

But let us get to the facts and quote this law:

"Organizations which design systems... are constrained to produce designs which are copies of the communication structure of these organizations."

From this law, we can draw some simple reflections:

- If I want a specific architecture, I need an organization aligned with my architecture.
- On the other hand, if I have to change my architecture often, I have to be able to modify my organization just as often.

These two assertions, which echo the principle of inverse conway maneuver, have far-reaching consequences. They underpin an organization's ability to adapt, which would ignore careerist tendencies, resistance to change, ultra-specialization of skills, and so on. They can also lead to philosophical reflections on the primacy of the machine over the human, but I am already digressing.

The corollary of all this is that the first question to be asked when we want to make a microservices architecture is: "How adaptable is the organization to this type of architecture?" Of course, it's tempting to think about Netflix and Amazon, but is your company ready? It is important to take this into account in order to quickly detect the brakes and "tricks" to circumvent the constraints.

One of the tricks to quickly ramp up is feature teams. Feature teams bring together several different skills to create a feature. But this can quickly become insufficient, because as your architecture explodes into microservices, coordination needs will arise.

One other trick is the open source governance model. Open source projects, because of their decentralized structure, make it possible to create highly decoupled software, which is what we want in microservices architectures. It may therefore be advisable to work in this way with other teams, with a small team having the code, and one or more extended teams being able to push changes in the code.

But what about the acceptance of this logic and organizational changes in a company? Are these tricks sufficient to instill coordination, skills, and knowledge throughout your company? Decentralized organizations build decoupled code, but technical or functional skills and knowledge shouldn't be decoupled to the extreme, either. It's like if you rob Peter to pay Paul, but here you rob shared knowledge to build decoupled architectures.

The real stalemate is more cultural than anything else, and a number of management styles that have emerged in recent years can help unblock the situation.

A fairly good example of what can be done to go further is the Spotify framework (although we should limit ourselves to meta-frameworks, because it is mostly a state of mind). Spotify uses the concepts of feature teams and governance with an open source approach, but complements these tools with a matrix model of agility at scale. Matrix organizations have the magic to ensure that you always get to know someone who knows the person who has the knowledge or skill.

So, when I studied the organizations of teams using microservices, I thought that something was missing. New management methods have become popular recently and could have an interesting influence, especially in organizations seeking to implement microservices.

Indeed, we touched on the subject of corporate culture, organization, and resistance to change. The first type of management that comes directly to my mind is holacracy.

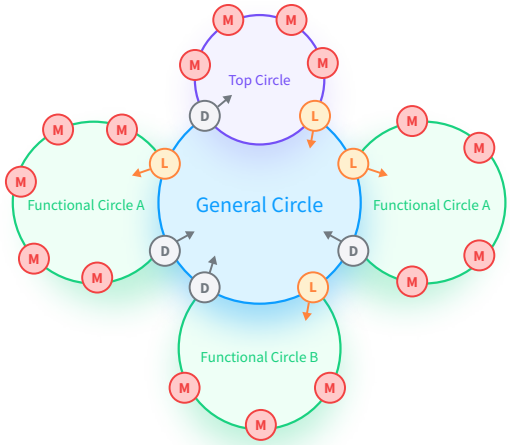
Holacracy is a fractal organization divided into autonomous and independent entities that are themselves linked to higher entities. These same entities are represented in the form of circles that can overlap with each other, and which have the particularity of being self-organizing while being managed by the upper circle. Each circle is thus very responsive to change in its nature and composition. The gains observed by this type of management are the involvement, cooperation, and simplicity of the links between people.

We could imagine, for example, that the elementary circles would be the microservices development teams, that the upper circles would be made up of architects and product owners, and that the top circle would be the client business lines of your application. This would give rise to product owners and architects who could coordinate the business needs, while ensuring that the best practices instigated by architects are implemented.

I say "we could imagine" because it is up to you to decide your needs and your solutions according to the desired architecture.

One of the driving forces behind this circle organization is Domain-Driven Design, often used in microservices projects. Indeed, this way of building applications typically brings developers, software architects, and experts in the field around the same table. All can potentially come from different circles or overlapped circles. It is therefore interesting to set up this type of organization in order to improve the transmission of knowledge and the time it takes to set up the architecture.

Contrary to what we might think, this type of management is relatively compatible with a traditional hierarchical organization. Indeed, even if the hierarchy is flattened, it still persists, and it can be circumscribed to IT project teams, in case your CxOs see this with bad eyes.



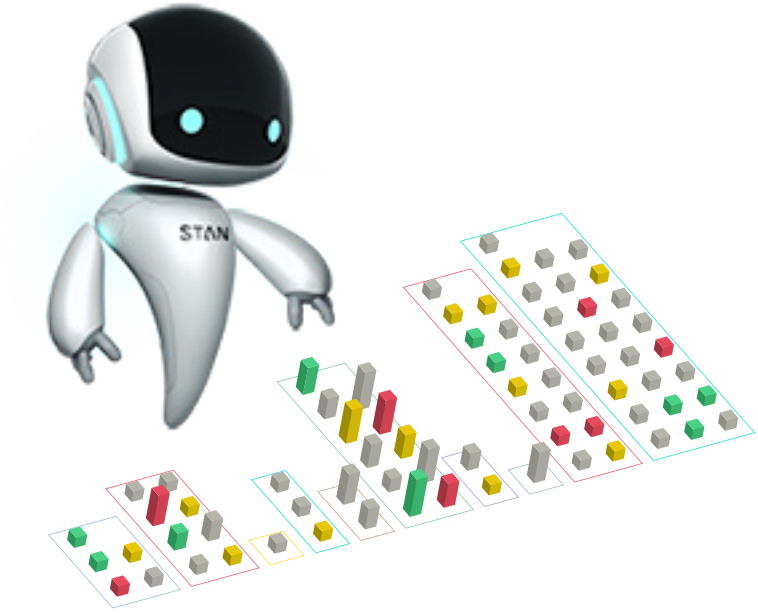
In case a holacracy cannot take hold in your organization, you can seek inspiration from sociacracy (also called Dynamic Governance). Sociacracy is not a mode of organization like holacracy, but more of a mode of governance without a centralized power structure, also operating under the principle of circles. These circles may also have overlapping boundaries, and are made up of the group's constituent elements, as well as delegates from the group and a group leader. Unlike holacracy, sociacracy aims to manage fewer operational subjects to focus on problems or strategic questions. It is thus a mode of governance that can perfectly be superimposed on any organization, and can be an intermediate step to a more disruptive organization such as a holacracy.

As we can see, other management styles exist, and can provide solutions to the extremely changing nature of microservices architectures. I am convinced that studying the impact of these architectures will lead companies to rethink their organizational models, to the delight of employees and customers alike. There is still the question of support for change and corporate culture. My opinion is that the corporate culture must always be respected but also reformed, because it will ultimately be the driving force behind the evolution of your organizations.

Thomas Jardinet: As an IT architecture consultant with thirteen years of experience, I accompany my clients in defining their architectures, whether functional, application or technical, by studying with them the best path. I also accompany them in the organizational side, and above all I seek with them intellectual and human exchange. I am also a supporter of flattened organisations, as I think it greatly improve productivity, robustness, and resilience of companies.

INSTANA

Stan Has Two Hobbies: Automation and AI



Any tool trying to monitor dynamic, containerized microservice applications must have AI. The environments are simply too complex to manually manage (or even configure the monitoring tool).

AI requires comprehensive automatic visibility into the full technical stack, coupled with application modeling to deliver automatic root cause determination.

Here are six fundamental skills Stan possesses around Automated Visibility and AI to help manage the performance of dynamic applications.

AUTOMATED VISIBILITY

AUTOMATIC, CONTINUOUS DISCOVERY & MAPPING

To apply an AI approach to performance management, the core model and data set must be up-to-date and impeccable, providing real-time visibility and an accurate picture of your application's structure and dependencies — all with no human configuration.

PRECISE HIGH FIDELITY VISIBILITY

AI requires precise data. For all discovered components, Instana collects the industry's most accurate monitoring data (streamed at 1 second granularity) and every request in a Trace. The data is the source for AI machine training and the basis for the deep microservice visibility.

CLOUD, CONTAINER & MICROSERVICE NATIVE

Instana is built to operate in the modern world. With zero configuration, Instana aligns with the infrastructure, clouds, containers, orchestrators, middleware and languages to accurately model and visualize dynamic microservice applications — wherever they are running.

ARTIFICIAL INTELLIGENCE

FULL STACK APPLICATION DATA MODEL

The core technology at the heart of Instana is the internal data model, called the Dynamic Graph. The Graph models all physical and logical components, the underlying technologies, dependencies and configuration. The Graph also understands logical components like traces, applications, services, clusters and tablespaces.

REAL-TIME AI-DRIVEN INCIDENT MONITORING & PREDICTION

Instana aligns alerts with business impacts and can predict impending service outages — using multiple AI methods to understand and predict application behavior. Predictive algorithms are applied to four derived KPIs (Transaction Rate, Error Rate, Latency and Saturation), leveraging the Dynamic Graph model to understand context.

AI-POWERED PROBLEM RESOLUTION AND TROUBLESHOOTING ASSISTANCE

Instana's AI-assisted troubleshooting leverages full visibility, the Dynamic Graph and AI-driven Incident management. Instana automatically identifies the most likely trigger of an Incident. Reports aggregate metrics, changes, traces and probable root cause on one screen. And predictive analysis identifies performance problems before they happen

Continuous Discovery: The Key to Continuous Delivery

Agile development, CI/CD, and the use of containers create constant application change in code, architecture, and even which systems are running.

This constant change, especially in microservices applications, makes Continuous Discovery a must-have feature for in order for effective monitoring tools to:

- Discover and map the microservices that make up the application
- Automatically monitor the health of each microservice on its own and as part of the app

Configuring agents is a key obstacle to achieving continuous discovery. To be specific, agent configuration has always been difficult, but dynamic applications make any configuration work obsolete almost as soon as it's complete.

The key to making continuous discovery work within monitoring tools is automation.

Automating agent configuration requires a change in the way agents are built and deployed, especially the way agents collect and transmit information through technology Sensors, such as:

- Configuration data
- Events
- Traces
- Metrics

Agents should automatically recognize every component and deploy the proper monitoring sensors, automatically collect the right data and provide a real-time health score.

This goes beyond code. Every technology component needs its own expert monitoring. So far at Instana, we've expanded that list to 9 languages and almost 70 unique technologies.

Why is this essential for Continuous Delivery?

1. **Speed**: Nobody has time to configure (and reconfigure) tools with the rate of change.
2. **Real-time Mapping**: Microservices applications are constantly changing so data flow and interactions can't be known in advance. Application maps must be built in real time.
3. **Instant Feedback**: Results from changes (including deployments) should be known in seconds. A delay of even a few minutes could be devastating.

You've created an agile development process. You're investing in containers and microservices. Don't let your monitoring tools prevent you from achieving your ultimate goals of continuous delivery.

PARTNER SPOTLIGHT

Instana APM for Microservice Applications

INSTANA

AI-Powered APM for Microservice Applications

CATEGORY
Application Performance Management

NEW RELEASES
V17.2 (Major release 2 or 3 times / year)

OPEN SOURCE
No

CASE STUDY

In just 2 years, Fintech company ClearScore had grown beyond the capabilities of its Java application. They migrated to microservices hosted in Docker containers, but their APM tool failed to match the efficiency and agility they valued. Simply maintaining the monitoring system was like a full-time job. The tool lacked native support for microservices and containers, or Scala.

ClearScore chose Instana's microservices-native APM. Instana delivered more detailed, context-aware insights while eliminating tool configuration by DevOps, it also identified and fixed problems quicker.

The team loves that Instana's automated visibliity and artificial intelligence doing everything from monitoring setup and threshold setting to troubleshooting.

STRENGTHS

- Full-stack application mapping and visibility of performance
- Supporting 80+ technologies: middleware, database, orchestration, containers, and 9 languages
- Predictive service incident monitoring and automatic root-cause analysis
- AI-assisted troubleshooting

NOTABLE CUSTOMERS

- Audi
- ClearScore
- Douglas
- Conrad
- Follett

WEBSITE instana.com | **TWITTER** @InstanaHQ | **BLOG** instana.com/blog

Communicating Between Microservices

QUICK VIEW

01 Explore the differences in communication between monolith/SOA systems and microservices-based architecture.

02 Learn the difference between synchronous and asynchronous communication.

03 Follow the most common patterns for microservices: use a load balancer, circuit breaker, and have the ability to fall back.

BY **PIOTR MIŃKOWSKI**
IT ARCHITECT, **PLAY**

One of the most important aspects of developing microservices rather than a monolithic application is an inter-service communication. With a monolithic application, running on single process invokes between components are realized on language-level method calls. If you are following the MVC design pattern during development, you usually have model classes that map relational databases to an object model. Then, you create components which expose methods that help to perform standard operations on database tables like create, read, update, and delete. The components most commonly known as DAO or repository objects should not be directly called from a controller, but through an additional layer of components which can also add some portion of business logic if needed.

Usually when I'm talking with others about migrating from a monolith to a microservices-based application, they see the biggest challenge just in changing their communication mechanism. If you've ever looked back on working on a typical monolithic application with a database backend, you probably realized how important it was to properly design relations between tables and then map them into object models. In microservices-based architecture, it's important to divide this often very complex structure into independently developed and deployed services, which are also forming a mesh with many communication links. Often the division is not as obvious as it would seem, and not every component which encapsulates logic related to a table becomes a separated microservice.

Decisions related to such a division require knowledge about the business aspects of a system, but communication standards can be easily defined, and they are unchangeable no matter which approach to architecture we decide to implement. If we are talking about communication styles, it is possible to classify them in two axes. The first step is to define whether a protocol is synchronous or asynchronous.

- **Synchronous** – For web application communication, the HTTP protocol has been the standard for many years, and that is no different for microservices. It is a synchronous, stateless protocol, which does have its drawbacks. However, they do not have a negative impact on its popularity. In synchronous communication, the client sends a request and waits for a response from the service. Interestingly, using that protocol, the client can communicate asynchronously with a server, which means that a thread is not blocked, and the response will reach a callback eventually. An example of such a library, which provides the most common pattern for synchronous REST communication, is Spring Cloud Netflix. For asynchronous callback, there are frameworks like Vert.x or Node.js platform.
- **Asynchronous** - The key point here is that the client should not have blocked a thread while waiting for a response. In most cases, such communication is

realized with messaging brokers. The message producer usually does not wait for a response. It just waits for acknowledgement that the message has been received by the broker. The most popular protocol for this type of communication is AMQP (Advanced Message Queuing Protocol), which is supported by many operating systems and cloud providers. An asynchronous messaging system may be implemented in a one-to-one (queue) or one-to-many (topic) mode. The most popular message brokers are RabbitMQ and Apache Kafka. An interesting framework which provides mechanisms for building message-driven microservices based on those brokers is Spring Cloud Stream.

Most think that building microservices is based on the same principle as REST with a JSON web service. Of course, this is the most common method, but as you can see it is not the only one. Not only that, in some articles, you might read that synchronous communication is an anti-pattern, especially when there are many services in a calling route.

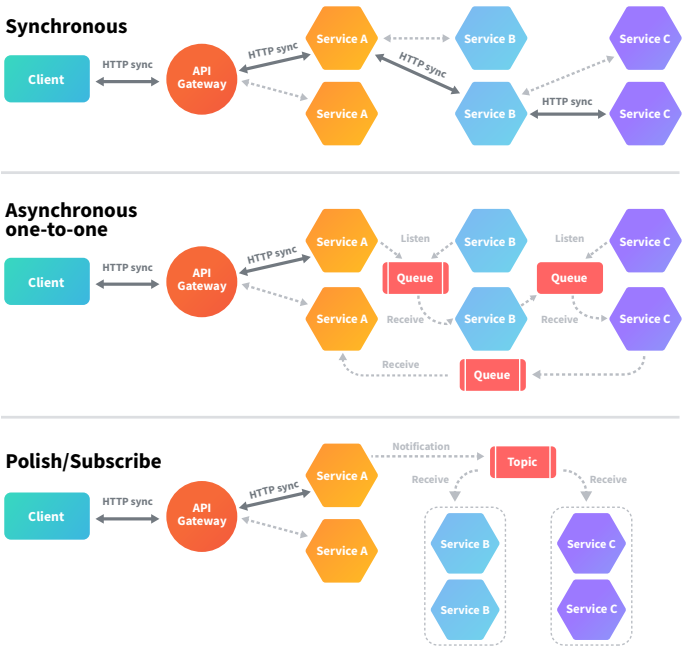
The other frequent comparison we might read about compares microservices to SOA architecture. In SOA, the most common communication protocol is SOAP. There have been a great deal of discussions as to whether SOAP is better than REST or vice versa. As we all know, they each have advantages and drawbacks, but REST is lightweight and independent from the type of language, so it has won the competition for modern applications, and is slowly taking over the enterprise sector. Honestly, I don't have anything against microservices based on SOAP if there is a good reason for it.

Let's look back at the criteria of division to different types of communication. I have already mentioned that we can classify them into synchronous vs. asynchronous, the latter of which defines whether the communication has a single receiver or multiple receivers. In one-to-one communication, each client request is processed by exactly one service instance, while each request can be processed by many different services. It is worth it to point out here that one message is received by different services, but usually it should not be received by different instances of a single service. Microservices frameworks usually implement a consumer grouping mechanism whereby different instances of a single application have been placed in a competing consumer relationship in which only one instance is expected to handle an incoming message.

For one-to-one synchronous services, the same can be achieved with a load-balancing mechanism performed on the client side. Each service has information about the location addresses of all instances that are calling services. This information can be taken from a service discovery server or may be provided manually in configuration properties. Each service has a built-in routing client that can choose one instance of a target service, using the right algorithm, and send a request there. These are the most popular load balancing methods:

- **Round Robin** - The simplest and most common way. Requests are distributed across all the instances sequentially.
- **Least Connections** - A request goes to the instance that is processing the least number of active connections at the current time.
- **Weighted Round Robin** - This algorithm assigns a weight to each instance in the pool, and new connections are forwarded in proportion to the assigned weight.
- **IP Hash** – This method generates a unique hash key from the source IP address and determines which instance receives the request.

Here's a figure that illustrates different types of communication used for microservices-based architecture, assuming the existence of multiple instances of each service:



In more complex architectures, there can be cases where those three communication types are mixed with each other. Then, some microservices are built on the basis of synchronous interaction, some on one-to-one messaging, and others on a publish/subscribe model.

There's been a lot of talk recently about reactive microservices, so I think it is worth it to devote a few words to it. It is based on the Reactive Programming paradigm, oriented around data flows and the propagation of change. Such microservices are non-blocking, asynchronous, event-driven, and require a small number of threads to scale. Their greatest advantage is excellent performance with a little resource consumption. The most popular frameworks for building reactive microservices are Lagom and Vert.x.

It is very important to prepare systems in case of partial failure, especially for a microservices-based architecture, where there are many applications running in separated processes.

Let's get back to synchronous request/response communication. It is very important to prepare systems in case of partial failure, especially for a microservices-based architecture, where there are many applications running in separated processes. A single request from the client point of view might be forwarded through many different services. It's possible that one of those services is down because of a failure, maintenance, or just might be overloaded, which causes an extremely slow response to client requests coming into the system. There are several best practices for dealing with failures and errors. The first recommends that we should always set network connect and read timeouts to avoid waiting too long for the response. The second approach is about limiting the number of accepted requests if a service fails or responses take too long. In this case, there is no sense in sending additional requests by the client.

The last two patterns are closely connected to each other. I'm thinking about the circuit breaker pattern and fallback. The major assumption of this approach relies on monitoring successful and failed requests. If too many requests fail or services take too long to respond, the configured circuit breaker is tripped and all further requests are rejected. On the other hand, fallback provides some portion of logic which has to be performed if request fails or circuit breaker had been tripped. In some cases it could be useful, especially when data returned by a service is not critical for the client or does not change frequently and may be taken from the cache. The most popular implementation of the described patterns is available in Netflix Hystrix, which is used by many Java frameworks, providing components for microservices like Spring Cloud or Apache Camel.

Implementation of a circuit breaker with Spring Cloud Netflix is quite simple. In the main class it can be enabled with one annotation:

```
@SpringBootApplication
@EnableFeignClients
@EnableCircuitBreaker
public class Application {
  public static void main(String[] args) {
       SpringApplication.run(Application.class, args);
  }
}
```

To communicate with another microservice we can use the Feign REST client, which handles fallback. Here, we return an empty list:

```
@FeignClient(value = "account-service", fallback =
AccountFallback.class)
public interface AccountClient {
    @RequestMapping(method = RequestMethod.GET, value =
    "/accounts/customer/{customerId}")
    List<Account> getAccounts(@PathVariable("customerId")
    Integer customerId);
}

@Component
public class AccountFallback implements AccountClient {
  @Override
  public List<Account> getAccounts(Integer customerId) {
       List<Account> acc = new ArrayList<Account>();
       return acc;
  }
}
```

Hystrix default settings may be overridden with configuration properties. The property visible below sets the time after which the caller will receive a timeout while waiting for response:

```
hystrix.command.default.execution.isolation.thread.
timeoutInMilliseconds=500
```

Piotr Mińkowski has more than 10 years of experience working as a developer and architect in the finance and telecom sectors, specializing in Java and its associated tools and frameworks. He works at Play, a mobile operator in Poland, where he is responsible for IT systems architecture. Here, he helps the organization migrate from monoliths to microservices-based architectures, as well as set up a CI/CD environment. In his free time, he publishes articles on Piotr's TechBlog, where he demonstrates the newest technologies and frameworks in the programming world.

DIVING DEEPER INTO MICROSERVICES

TOP #MICROSERVICES TWITTER ACCOUNTS

@TEDEPSTEIN
Ted Epstein

@INADAREI
Irakli Nadareishvili

@TETIANA_FTV
Tetiana Fydorenchyk

@SANEEPDINESH
Sandeep Dinesh

@SAMNEWMAN
Sam Newman

@HJHARNIS
Harrison Harnisch

@MYFEAR
Markus Eisele

@THOHELLER
Thorsten Heller

@MARTINFOWLER
Martin Fowler

@ZIOBRANDO
Alberto Brandolini

BEST MICROSERVICES ZONES

INTEGRATION DZONE.COM/INTEGRATION

The Integration Zone focuses on communication architectures, message brokers, enterprise applications, ESBs, integration protocols, web services, service-oriented architecture (SOA), message-oriented middleware (MOM), and API management.

CLOUD DZONE.COM/CLOUD

The Cloud Zone covers the host of providers and utilities that make cloud computing possible and push the limits (and savings) with which we can deploy, store, and host applications in a flexible, elastic manner. The Cloud Zone focuses on PaaS, infrastructures, security, scalability, and hosting servers.

JAVA DZONE.COM/JAVA

The largest, most active Java developer community on the web. With news and tutorials on Java tools, performance tricks, and new standards and strategies that keep your skills razor-sharp.

TOP MICROSERVICES REFCARDZ

MICROSERVICES IN JAVA

BY JOSHUA LONG

This Refcard turns concepts into code and lets you jump on the design and runtime scalability train right away – complete with working Java snippets that run the twelve-factor gamut from config to service registration and discovery to load balancing, gateways, circuit breakers, cluster coordination, and security.

SPRING BOOT AND MICROSERVICES

BY NEIL STEVENSON

This Refcard will show you how to incorporate Spring Boot and Hazelcast IMDG into a microservices platform, how to enhance the benefits of the microservices landscape, and how to alleviate the drawbacks of utilizing this method.

REACTIVE MICROSERVICES WITH LAGOM AND JAVA

BY MARKUS EISELE

Using this open-source framework, you can build Microservices as reactive systems that are elastic and resilient from within.

TOP MICROSERVICES RESOURCES

EXPLORING THE UNCHARTED TERRITORY OF MICROSERVICES

In this webinar, four experts – including Gene Kim – answer questions like "What are the promised benefits of microservices?" and "What can go wrong as we transform the organization and architecture to microservices?"

BUILDING MICROSERVICES: DESIGNING FINE-GRAINED SYSTEMS

BY SAM NEWMAN

In this book, learn oprtions for integrating a service with the rest of your system, how to deploy individual microservices through continuous integration, and more.

MICROSERVICES AT NETFLIX SCALE: PRINCIPLES, TRADEOFFS, AND LESSONS LEARNED

In this recorded talk from Netflix's Ruslan Meshenberg, learn about the adoption of microservices in a large organization, how to do microservices ops at scale, and more.

TOP MICROSERVICES PODCASTS

MICROSERVICES TRANSITION

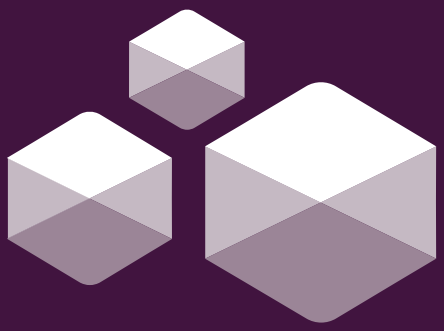
Learn about an engineer's experiences transitioning from a monolith to microservices and how she recommends making it easier.

MANUFACTURING AND MICROSERVICES

Microservices at large can be different than microservices at a smaller company. Learn about using microservices infrastructure and technology on a mass customization platform.

MICROSERVICES, DISTRIBUTED TEAMS, AND CONFERENCES

Learn about the challenges of running a distributed team and preventing developer burnout in the process of migrating to microservices.



lagom

Open source.
Highly opinionated.

Build greenfield microservices & decompose your Java EE monolith like a boss.

Get involved at
lagomframework.com

Lightbend

The Evolution of Scalable Microservices

From building microliths to designing reactive microsystems

Today's enterprise applications are deployed to everything from mobile devices to cloud-based clusters running thousands of multi-core processors. Users have come to expect millisecond response times and close to 100% uptime. And "user" means both humans and machines. Traditional architectures, tools and products simply won't cut it anymore. To paraphrase Henry Ford's classic quote: We can't make the horse any faster, we need cars for where we are going.

While many organizations move away from the monolith and adopt a microservices-based architecture, they mostly do little more than creating microlith instances communicating synchronously with each other. The problem with a single instance is that it cannot be scalable or available. A single monolithic thing, whatever it might be (a human or a software process), can't be scaled out, and can't stay available if it crashes.

But it is also true that as soon as we exit the boundary of the single service instance we enter a wild ocean of non-determinism—the world of distributed microservice architectures.

The challenge of building and deploying a microservices-based architecture boils down to all the surrounding requirements needed to make a production deployment successful. For example:

- Service discovery
- Coordination
- Security
- Replication
- Data consistency
- Deployment orchestration
- Resilience (i.e. failover)
- Integration with other systems

Built using technologies proven in production by some of the most admired brands in the world, Lagom is the culmination of years of enterprise usage and community contributions to Akka and Play Framework. Going far beyond the developer workstation, Lagom combines a familiar, highly iterative code environment using your existing IDE, DI, and build tools, with additional features like service orchestration, monitoring, and advanced self-healing to support resilient, scalable production deployments.

WRITTEN BY MARKUS EISELE
DIRECTOR OF DEVELOPER ADVOCACY, **LIGHTBEND, INC.**

PARTNER SPOTLIGHT

Lagom Framework By Lightbend

Lightbend

"Java finally gets microservices tools." -Infoworld.com

CATEGORY
Microservices Framework

NEW RELEASES
Multiple times per year

OPEN SOURCE
Yes

CASE STUDY

Hootsuite is the world's most widely used social media platform with more than 10 million users, and 744 of the Fortune 1000. Amidst incredible growth, Hootsuite was challenged by diminishing returns of engineering pouring time into scaling their legacy PHP and MySQL stack, which was suffering from performance and scalability issues. Hootsuite decomposed their legacy monolith into microservices with Lightbend technologies, creating a faster and leaner platform with asynchronous, message-driven communication among clusters. Hootsuite's new system handles orders of magnitude more requests per second than the previous stack, and is so resource efficient that they were able to reduce Amazon Web Services infrastructure costs by 80%.

STRENGTHS

- Powered by proven tools: Play Framework, Akka Streams, Akka Cluster, and Akka Persistence.
- Instantly visible code updates, with support for Maven and existing dev tools.
- Message-driven and asynchronous, with supervision and streaming capabilities.
- Persistence made simple, with native event-sourcing/CQRS for data management.
- Deploy to prod with a single command, including service discovery and self-healing.

NOTABLE CUSTOMERS

- Verizon
- Samsung
- UniCredit Group
- Walmart
- Hootsuite
- Zalando

WEBSITE www.lagomframework.com

TWITTER @lagom

BLOG lagomframework.com/blog

What to Consider When Dealing With Microservices Data

QUICK VIEW

01 As we introduce new data sources, we may have to deal with data concerns within the application.

02 Splitting out our data involves more than just splitting tables in a database.

03 The boundaries between our services suggest a time component to dealing with data.

04 Systems may be "eventual consistent" for inter-boundary data communication.

BY **CHRISTIAN POSTA**
CHIEF ARCHITECT CLOUD APPLICATIONS, **RED HAT**

Useful applications collect, munge, and present data to its users. Data becomes the lifeblood of an application, so to speak. As developers of an application with a single database, we are afforded many helpful abstractions: atomic transactions, tunable concurrency, indexes, materialized views, and more. This "single database view" of the world simplifies things for our application developers. As soon as we add more databases to our application (single application with multiple backends/databases), we have to deal with data challenges within the application.

For example, if our application's main database is a MySQL database with all of the transactional workloads going through it, we may decide that for a particularly sensitive area of our application, we want to use something like Oracle, which may have better support for encryption of data at rest. Now, our application will have to make multiple data calls (two different databases), process queries across both databases and the joining of data inside our application code, and also figure out how best to handle atomicity challenges on updates (i.e. distributed transactions, self-managed eventual consistency, triggers, or non-transactional datastores).

Now, let's imagine that we want to move to a microservices architecture. I'm sure you've heard the claim that each microservice should have its own database or datastore. What happens to our data?

As we start to break functionality into separate services, we'll quickly confront these challenges. There are two main things to understand here. First, as Pat Helland reminds us, data on the inside of our service must be treated differently than data outside our service. Data inside of a service can still take advantage of the conveniences and abstractions afforded to us by the database that we decide to use (atomicity, query planning, concurrent controls, etc.). When services communicate with each other and send data outside a service boundary, we're inherently dealing with stale data. Said a different way, as soon as data leaves a service, there is no guarantee it's the most recent version of that data.

The second thing to understand: Since the data on the outside of our services cannot come with recency guarantees (it's stale), there is a component of time to this equation. Microservices involved in this system will "eventually" see the updates of other services and must factor this into their application design. Some would describe this as an "eventually consistent" system.

How can we design around these two factors? To wit, are there design principles, patterns, and practices that take data on the inside/outside and time into account when building a system?

Domain-driven design fits this mindset quite well, and

forces us to think more about how the business operates, the language it uses to describe their complexity, the natural transactional boundaries that the business sees, and how best to model this in software. This encourages us to think more closely about the natural transaction boundaries (non-technology speaking) that exist in the domain, draw a boundary around those (i.e. the bounded context), and map the interactions between these boundaries (i.e. context mapping). For example, if we take a naive, purely technological approach to a solution, without regard for the business or domain, we may end up building services around "User," "Account," "Order," etc., each with its own database. Now, any time we need to refer to a User, Account, or Order, we need to consult with these respective services and these "services" end up being very hollow or anemic CRUD services doing little more than data access. Is that what a service is? What if we spent a little more effort to understand how the business thinks of these concepts? What is a User? What is an Order? What is a "thing?"

In my talks, I like to illustrate this complexity with a very simple example (borrowed from William Kent) by asking a question: What is a "book?" How would we describe what a "book" is for a fictitious online bookstore? A book has pages, a cover, and an author. I've written a book. So, would there be one entry in the system for the book I wrote? I have about 20 or so copies of that book next to my desk, and infinite copies as e-books online. Is each one of those a "book?" Is the e-book not a book until someone downloads it? Some books are so big they have to get broken down into smaller volumes. Is the whole thing a book? Or just the individual volumes? Which is it? In our online book store, what a book is depends on the domain. A book may be represented and treated differently in the order/checkout part of the system than in the recommendation engine. For ordering/checkout, we do care about each individual physical/electronic book. For the recommendation engine, we may just care about metadata, how it relates to other metadata, and its possible relevance. So maybe the services we have are the ordering/checkout service, catalog search, and recommendations. Each service will have an understanding of "book" that makes sense for it to provide a service.

Identifying these nuances in the domain and drawing boundaries around them allows us to focus on the inside vs. outside of the data. If we make changes to a book, or an order, or an account within the bounded context, we expect that to be aligned to a transactional boundary and be strictly consistent. When we make a change to the Order, it is consistent with any read/writes afterward. But, as we see with the book example, these concepts may be shared across multiple services, though their representation may be slightly (or dramatically) different. But how do we communicate changes about this data that might be similar?

DDD theory isn't very opinionated about how the data is shared. The discussion in the DDD community revolves around interaction relationships like "customer/supplier," "conformist," "anti-corruption," etc. Even so, a lot of practical implementations of these ideas end up going down the route of an event-driven architecture, raising events when interesting things happen within a bounded context and letting other bounded contexts interpret that event. An "event" here is announcing a fact that something happened (in the past — note the relationship to time and inherent staleness) in which other parts of the system may also be interested. For example, in our Checkout bounded context, if we successfully process an Order, we can store that within our own transactional boundary and then raise an event named "CheckoutPurchaseCompleted" with a reference to the book ID that was purchased. Other systems interested in this fact, maybe the Search service, can capture that event and make some decisions based on it; maybe it decreases its locally stored count of a particular book's inventory and uses this as a factor of whether to display in search results. This way, the Search service doesn't have to continuously call an Inventory or Book Availability service every time it has a search result that includes a particular book.

By taking an event-driven approach combined with DDD, we make Pat Helland's "data on the inside vs. data on the outside" a core part of the design — which encourages us to think more closely about the "time" aspects of distributed systems. If we can comfortably live in this environment, we can achieve the holy grail of autonomous microservices, which then allows us to make changes quicker and independently from the rest of the system.

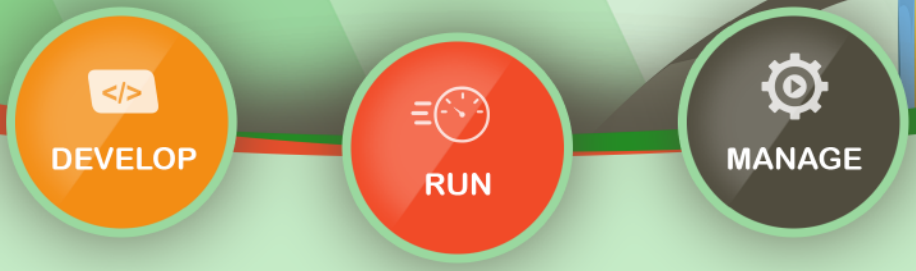
Christian Posta (@christianposta) is a Chief Architect of cloud applications at Red Hat and well known in the community for being an author (*Microservices for Java Developers*, O'Reilly 2016), frequent blogger, speaker, open-source enthusiast, and committer on various open-source projects. Christian has spent time at web-scale companies and now helps companies create and deploy large-scale, resilient, distributed architectures — many of what we now call microservices. He enjoys mentoring, training and leading teams to be successful with distributed systems concepts, microservices, DevOps, and cloud-native application design.

macaw.io

Build and Transform Enterprise Applications for Digital Age

Macaw: Modern Application Platform

Microservices | Containers | MTA | Serverless



Transformation — Accelerate Modernization of Traditional Applications

Hybrid Cloud — Securely Run Microservices on Federated Kubernetes Clusters

Business Agility — Quickly Build 12 Factor Microservices based Applications

Macaw Software Inc.

macaw.io
macaw.io/download

macaw.io/blog
@macawbuzz

Accelerate Modern Enterprise Application Journey with Macaw

Emerging technologies such as cloud, DevOps, microservices, containers, Big Data, and serverless computing offer the promise of a scalable, agile, and lean IT environment. As much as enterprises like to embrace these emerging technologies to build new systems, they are often required to maintain and evolve legacy systems to support existing customers and business processes. To effectively address this practical dichotomous problem, enterprises need a platform that can modernize legacy applications as well as enable new application development with emerging technologies. For optimal results, enterprises should achieve both objectives through one common architectural plane and platform.

Modernizing Traditional Applications (MTA): The status quo of the current approach involves systematically rewriting existing code to newer technologies. However, this approach is fraught with risks, can be costly, and can lead to prolonged or failed projects. Macaw takes a distinctive approach: instead of disruptive forklifting, Macaw employs a streamlined process at the desired pace and choice of transforming legacy (Java/J2EE, .NET, etc.) applications to cloud ready architectures. Specifically, Macaw discovers and maps existing application dependencies, and provides a toolset to containerize and operate components of the application that are ready to be transformed to a microservices model. This process iterates until the whole application is modernized.

"We were able to quickly re-architect our multi-tiered J2EE application by using Integrated Macaw Microservices and Kubernetes Container Environment" - SENIOR ARCHITECT, LARGE TELECOM PROVIDER

"We replaced existing monolithic Monitoring tool with next generation Hybrid IT Monitoring solution, built with Macaw platform. Now, our solution is very responsive, highly scalable, and able to support new devices/ technologies quickly" - DIRECTOR OF OPERATIONS, LEADING MSP

Building highly scalable and distributed cloud native applications: Macaw makes it easy to develop and implement 12-factor cloud native applications with microservices and containers. Macaw provides several built-in key architectural components and essential services such as a database, messaging, registry, API gateway, identity management, container orchestration, and Kubernetes integration. Macaw addresses the challenges of managing distributed applications using monitoring, real-time message correlation, governance, and performance monitoring capabilities.

WRITTEN BY SATYAN RAJU
CDO, **MACAW SOFTWARE INC.**

PARTNER SPOTLIGHT

Macaw

Macaw fulfills the promise of one modern application platform that will bring traditional enterprise applications and modern applications into the new paradigm of highly scalable, containerized, distributed, and microservices-based cloud ready architectures.

CATEGORY
Modern Applications Development and Runtime Platform

NEWEST RELEASE
Currently 0.9.5

OPEN SOURCE
No

CASE STUDY

Challenge: One of the leading Telecom Enterprises was using a legacy monitoring solution. Customers frequently complained about a sluggish portal, scalability issues, and empty charts/reports. Supporting new devices/ technologies took a long time.

Solution: Developed next generation hybrid IT monitoring solution using Macaw Platform. This solution leverages many built-in Macaw microservices and container capabilities, and is highly responsive and scalable. The customer was able to seamlessly integrate with legacy enterprise systems as well.

Benefits: Achieved 40% customer growth in less than 6 months. Cut down customer onboarding time from 6 weeks to 1 week. Added support for new devices in a couple of weeks — which used to take months with the legacy solution.

STRENGTHS

- One stop microservices platform: develop, deploy, run, and manage
- Out-of-the-box support for federated Kubernetes clusters and AWS serverless Lambda integrations
- Can support both green-field and brown-field applications
- Designed and architected to address DevOps and Containerization needs
- Built-in microservices monitoring, debugging, and operations capabilities

NOTABLE CUSTOMERS

- AT&T
- GDT
- First National Bank
- Monsanto
- Sysco Foods
- Oracle

WEBSITE macaw.io | **TWITTER** @macawbuzz | **BLOG** macaw.io/blog

Reactive Persistence for Reactive Systems

QUICK VIEW

01 Traditional persistence is not a good fit for reactive microservices architectures.

02 Reactive Persistence should utilize DDD, CQRS, and event sourcing design patterns.

03 Out-of-the-box solutions like Lagom and Kafka can help build reactive systems.

BY **MARK MAKARY**
CTO, **LOGIC KEEPERS**

A reactive microservices architecture is an architectural style that strives to provide the highest levels of responsiveness, resiliency, and elasticity, and accomplish this by adopting strong decoupling, isolation, non-blocking, event-driven architecture, and asynchronous messaging, among other techniques.

TRADITIONAL PERSISTENCE IS A POTENTIAL BOTTLENECK FOR REACTIVE SYSTEMS

Reactive architecture uses Domain Driven Design (DDD) or similar design patterns to accomplish isolation and create separate and isolated entities with aggregate routes and bounded contexts to create well-rounded and independent microservices.

By persisting such separated entities as part of business transactions using traditional persistence methods within the conventional CRUD realm, such as Two Phase Commit or 2PC, and adhering to the ACID concept, we are creating significant blocking bottlenecks that can potentially limit the capabilities and promise of reactive systems. Traditional persistence is not a good fit for reactive systems. We need a persistence approach that can promote minimal blocking and create decoupling between the business services and the persistence layer.

REACTIVE PERSISTENCE CQRS AND EVENT SOURCING

Reactive persistence uses Command Query Responsibility Segregation (CQRS) and event sourcing to accomplish this asynchronous and decoupled interaction.

The CQRS will provide the decoupling between the read and write channels, allow greater freedom, and provide responsiveness for the persistence operations.

Event sourcing focuses on appending the various states of an entity or domain to an event journal using event persistence commands while you can have event subscribers and processors attached to the journal responding to the appended events and working on building the entity or object state in the query database or persistence store.

Event sourcing and CQRS will provide better write performance as it will only perform append operations instead of a full domain object update,, and will improve scalability as we have decoupled and separated the Write and Read processes. Also, event sourcing provides better audit control and a ripe environment for analytics.

To drive the point closer, we can follow an order management example or use case. Using event sourcing, we can append different order states to the journal while we append similar events to other domains associated with an order, such as payments or inventory. The event subscribers and processors will construct the domain state in the query store using those fragmented events received from the event producers.

TECHNOLOGIES

DDD, CQRS, and event sourcing are not new design

patterns. However, recent advances in network and persistence technologies allowed the open source community to build and implement those design patterns, providing robust and enterprise grade, highly available, scalable, and distributed reactive persistence technologies. Below are two out-of-the-box technologies that fully or partially support CQRS and event sourcing.

LAGOM PERSISTENCE

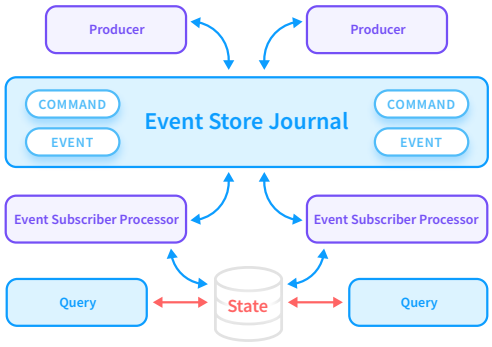
Lagom is a reactive framework built on top of the Akka Toolkit and Play framework technologies. The framework is created and supported by Lightbend.

Lagom Persistence is a CQRS and event sourcing implementation, and can easily provide direct mapping to the business domains by deriving and attaching the core Lagom persistence classes and utilities.

APACHE KAFKA

Kafka is another revolutionary technology, it's a high performance and high scalability distributed data streaming platform.

Kafka is a clustered and distributed pub/sub asynchronous messaging platform and event store at the same time. As highlighted in Figure 1, we can use Kafka as the event store and separate the front-end services supplying domain events through commands (producers) from the backend services that subscribe to those events and construct the domain state in the query database store.



REACTIVE PERSISTENCE CONSIDERATION

When using CQRS and event sourcing, we have some important aspects to consider, some of the most important ones are described further below.

EVENTUAL CONSISTENCY

Reactive persistence is using eventual consistency rather than strong consistency. For instance, using the order management example we used earlier, the order status (created, submitted, completed, etc.) might not be consistent with the event store's latest event for some time, and the business logic needs to compensate for and accommodate eventual consistency.

DDD, CQRS, and event sourcing are not new design patterns. However, recent advances in network and persistence technologies allowed the open source community to build and implement those design patterns, providing robust and enterprise grade, highly available, scalable, and distributed reactive persistence technologies.

ENTITY TRACKING

Using the order management example, all the events are stateless and immutable; so how can we keep track of the order? The system or the services must keep the order's unique id in the exchanged messages, and the events must be able to identify and construct or reconstruct the specific order.

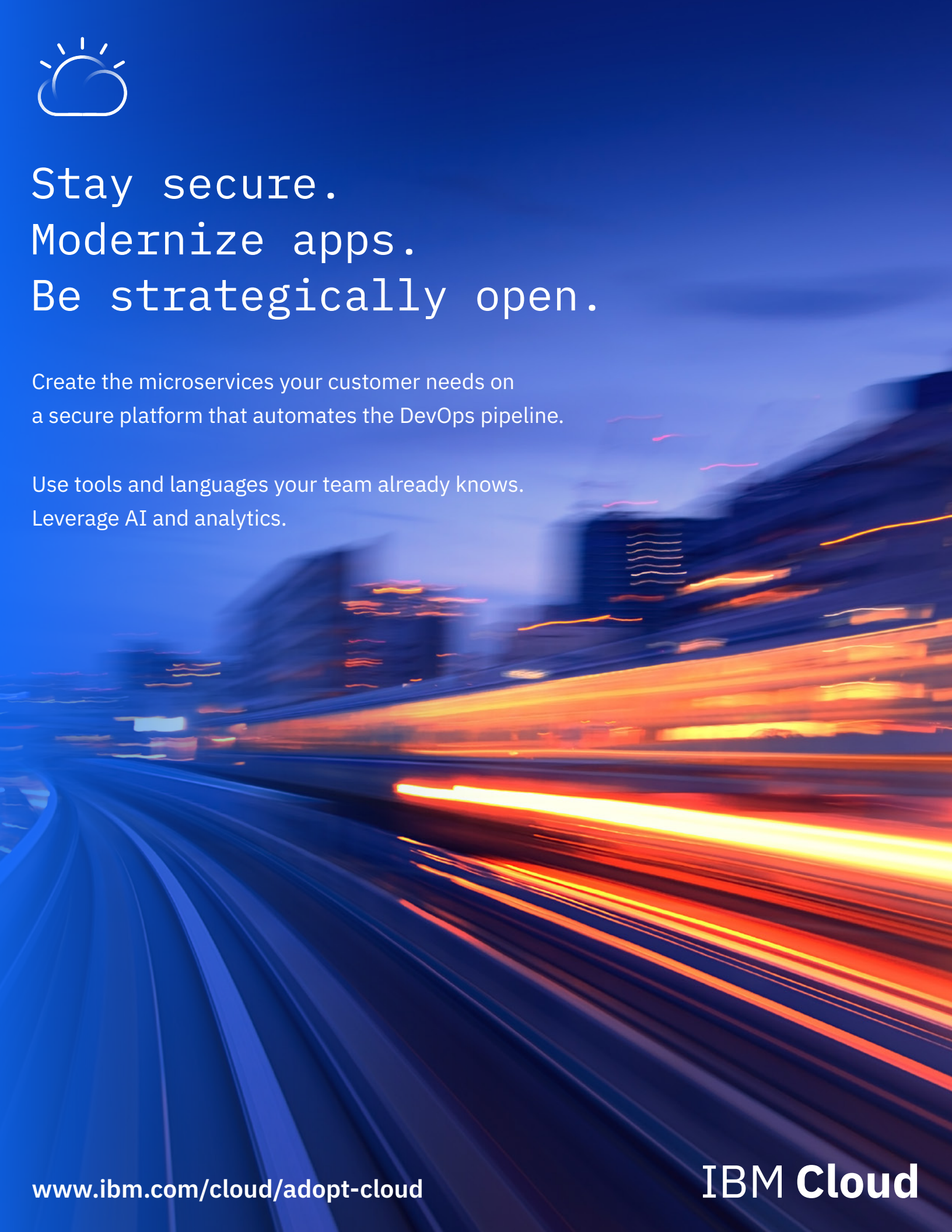
ORDER OF EVENTS

In a distributed system, the order of events is of significant importance, as constructing the correct domain state will directly depend on the correct order of events. For instance, if the event processor processes the "order submitted" event after the "order completed" event, the wrong order state will be represented in the order table.

CONCLUSION

Currently, we have a variety of technologies and techniques used in designing and implementing reactive persistence. Those different approaches can fit different use cases; each have their own advantages and trade-offs. It's up to the architects to determine the best fit. However, the good news is that we are not bound to traditional persistence anymore, and we have many options where we can easily build and utilize reactive persistence.

Mark Makary is the founder, President, and CTO of Logic Keepers — is an entrepreneur, enabler, and IT industry veteran who is empowering others to solve difficult real-life problems and providing innovative solutions utilizing cloud architecture and open source technologies. Mark is a thought leader and author, focusing on reactive architecture and programming, emerging technologies, distributed computing, API management, B2B integration, and information security.


Stay secure.
Modernize apps.
Be strategically open.
Create the microservices your customer needs on
a secure platform that automates the DevOps pipeline.
Use tools and languages your team already knows.
Leverage AI and analytics.
www.ibm.com/cloud/adopt-cloud
IBM Cloud

Small bank. Big outcomes.

Operating since 2008 with a maximum of 200 employees forces UBank to attract new business through non-traditional methods. As Australia's leading digital-only bank, UBank's online home loan application process disrupted retail banking by delivering a simpler, better and smarter customer experience.

Recognizing the need to deliver value to customers faster, we began adopting a cloud native development model. By first engaging with the IBM Watson and Cloud Adoption team and then visiting the IBM Cloud Garage, we created a Facebook plugin referral app for home loans. In working on that first minimum viable product (MVP) and leveraging the Garage Method, we transformed our agile product teams into full DevOps teams focused on the business functions of what we wanted to create for our customers. Rather than waterfall project deadlines, planned outcomes in customer experience drove our delivery.

For our next MVP, using Watson Conversation, our teams transferred the knowledge of call center staff and FAQs into an Artificial Intelligence (AI) driven chat application, RoboChat, which searches information based on natural language user requests. RoboChat uses an orchestration microservice built as a Node.js runtime to connect with the microservice we built in Watson Conversation itself.

"...as part of the DevOps cycle, relevant teams review details of the RoboChat session — stored in a Cloudant database — to determine how to further improve the Watson Conversation microservice..."

As needed, based on verbal cues, RoboChat transfers a customer session to one of our live Advisors for additional help. In such cases, as part of the DevOps cycle, relevant teams review details of the RoboChat session — stored in a Cloudant database — to determine how to further improve the Watson Conversation microservice, expanding the scope of questions RoboChat can automatically answer in the future.

Our teams delivered RoboChat — concept to production — in eight weeks, resulting in a dramatically new and improved customer experience. With just two MVPs, in addition to improving our customers' experience in applying for a loan, we also established and improved our DevOps process to achieve consistently rapid delivery. Continuing to operate within this cloud native model lets us to try different ideas in quick succession as we evolve apps into the next valuable customer experience.

With each DevOps team responsible for a different microservice, and with the microservices capable of interacting through APIs, innovating customer experience can be driven from as many directions as we have business needs and teams.

What excites me the most is the autonomy we're giving to our product teams. New feature ideas are being put into production without the burden of waiting for multiple teams to sequentially coordinate. With each DevOps team responsible for a different microservice, and with the microservices capable of interacting through APIs, innovating customer experience can be driven from as many directions as we have business needs and teams.

We're excited about continuing to leverage AI capabilities and a microservices architecture to innovate beyond the boundaries of banking.

IBM

WRITTEN BY JEREMY HUBBARD
HEAD OF DIGITAL AND TECHNOLOGY AT UBANK ON BEHALF OF IBM

Executive Insights on the Current and Future State of Microservices

QUICK VIEW

01 The ability to decouple and deliver application functionality faster, with greater stability through agile development methodologies is a tremendous benefit to organizations.

02 Microservices and DevOps go hand-in-hand. There's a reciprocal relationship between the two. Microservices architecture will not work without a DevOps methodology.

03 Microservices have improved SDLC best practices; speed, agility, flexibility, and the alignment of software with the business.

BY **TOM SMITH**
RESEARCH ANALYST, **DZONE**

To gather insights on the state of microservices today, we spoke with 19 executives who are familiar with the current state of microservices architecture. Here's who we spoke to:

MATT MCLARTY VICE PRESIDENT, API ACADEMY, CA TECHNOLOGIES

BRIAN DAWSON DEVOPS EVANGELIST, CLOUDBEES

LUCAS VOGEL FOUNDER, ENDPOINT SYSTEMS

THOMAS BUTT CTO, CARDCASH

ALI HODROJ V.P. PRODUCTS AND STRATEGY, GIGASPACES

JOB VAN DER VOORT VP PRODUCT, GITLAB

KEVIN SUTTER MICROPROFILE AND JAVA EE ARCHITECT, IBM

SANDEEP SINGH KOHLI DIRECTOR OF MARKETING, MULESOFT

KARL MCGUINNESS SENIOR DIRECTOR OF IDENTITY, OKTA

ROSS SMITH CHIEF ARCHITECT, PITSS AMERICA

MIKE LAFLEUR DIRECTOR OF SOLUTION ARCHITECTURE, PROVENIR

GIANNI FIORE CTO, REBRANDLY

PETER YARED CTO, SAPHO

SHA MA V.P. SOFTWARE ENGINEERING, SENDGRID

KESHAV VASUDEVAN PRODUCT MKTG. MGR., SWAGGER/SWAGGERHUB, SMARTBEAR

CHRIS MCFADDEN V.P. ENGINEERING AND OPERATIONS, SPARKPOST

CHRISTIAN BEEDGEN CO-FOUNDER AND CTO, SUMO LOGIC

TODD MILLECAM CEO, SWYM SYSTEMS, INC.

TIM JARRET SENIOR DIRECTOR OF PRODUCT MARKETING, VERACODE

KEY FINDINGS

01 The most important elements of microservices are **speed, decentralization, and size**. The ability to decouple and deliver application functionality faster, with greater stability and agile methodologies is a tremendous benefit to the organization and its end users. The speed of new feature development, ongoing maintenance, and the day-to-day work of deployment and testing is very rewarding to everyone involved.

Microservices address the architectural bottleneck with decentralization and the isolation of responsibility and faults, as well as autonomous, local data sources. Breaking code into smaller pieces results in shipping less code more often, with smaller feedback loops. This results in components that are easier to manage, maintain, refactor, and control.

Microservices and DevOps go hand-in-hand. There's a reciprocal relationship. To deliver microservices as a core part of your architecture you need the components of DevOps: agile development methodologies, CI, and CD. Likewise, decoupled apps are difficult to deliver without DevOps. Implementing DevOps helps to deliver decoupled apps faster.

02 The most frequently mentioned languages for developing microservices were **Java and Node.js**. There were more than 35 different languages, frameworks, and tools mentioned that demonstrate the multitude of ways developers, engineers, and architects are building microservices architectures.

03 **Microservices have improved SDLC best practices, speed, agility, flexibility, and alignment of the software with the business**. Microservices are the embodiment of software development best practices: simple code, easy to maintain, easy to train other developers, good habits like encapsulation, isolation of complexity, autonomous development teams, breaking down silos between applications. They are also symbiotic with DevOps with more frequent deployments, automated testing, zero downtime deployment, and easier rollback.

Legacy enterprises with monolithic apps become more agile in order to enable digital transformation. Microservices provide faster speed to market and realization of value with cloud scaling. This supports an agile approach to development. Smaller components allow for meaningful changes with leaner teams, resulting in faster responsiveness to the market.

Lastly, microservices align to business objectives and full-stack teams are aligned to customer value. Deployment is aligned with operations and teams are more collaborative because microservices are inherently more collaborative.

04 The most frequently mentioned security techniques for microservices were using APIs and **API access controls and gateways**. You need to put together standards for access control in the API architecture with certificates and tokens. Require an API gateway key or login. API gateways provide many great out-of-the-box management services in addition to security. APIs are an effective way to build governance into the microservices architecture. SLAs can be managed through API gateways that act as proxies for the microservices. This ensures there is a proper balance of governance for IT and flexibility for the domain teams.

05 A key "real-world" problem solved by microservices is **the decoupling of monolithic applications so legacy enterprises can pursue digital transformation**. Microservices force you to break your problems down into buildable pieces. Critical components that were previously part of a monolithic application can now be easily extracted and rebuilt in a way that doesn't interfere with the rest of development. Decoupling = faster development = faster time to market = greater revenue (as demonstrated by Netflix, Google, and Amazon).

Pitney Bowes is making the digital transformation to an open-commerce cloud that's accessible by APIs. 130-year-old Unilever has created a large number of microservices to support its continued growth, enabling the company to connect its e-commerce applications to the various legacy systems that support its core operations across a global portfolio of brands. They are pairing their microservices architecture with API-led connectivity time to drastically reduce development time for new e-commerce applications.

06 The most common issue affecting the implementation of microservices is the **amount of change required**. This is yet another operational and developmental paradigm shift. You have to face the initial configuration and driver setup costs to connect your service to different protocols. The architectural maturity of an organization is often the greatest hindrance to adoption and implementation. Clients frequently need new employees, new process models, and a new hosting infrastructure to get the most out of a new microservices architecture. As such, we focus on educating customers on the options that best fit their situation.

07 Concerns over microservices are consistent with those of other new technologies: **integration and data challenges, complexity, and lack of governance or best practices**.

Microservices introduce problems integrating with persistent storage. You need to determine how to provide the right data for the right context without making every data payload overloaded with unnecessary attributes and JSON response collections. Parallel deployments of similar data-providing services drawing from the same underlying libraries and data sources.

There is no "one size fits all." There are many languages, tools, and API gateways. Microservices let you scale but it comes with its own set of complexities.

Microservices aren't governed, so the potential roll-out is very "wild west." It will take a while to adopt best practices with patterns and use cases. That's why it's important for early adopters of microservices to share their experience – both good and bad.

08 **Serverless and functions-as-a-service are clearly the future of microservices** according to our respondents. Acceleration to the cloud, integration, and greater reuse were also elements of the future. Greenfield apps will be serverless with event-driven programming and progressive web apps. The move to on-demand compute resources and serverless architectures will grow. Lambda is disruptive and microservices will extrapolate to serverless with Lambda.

In addition, microservices will drive the adoption of, and integration into, the cloud. There will be improved integration with microservices sharing and tying multiple microservices together. There is also greater potential for reuse. Reuse will take a different frame of reference from the production of reusable assets to its consumption.

09 **It seems developers need to keep everything, including the kitchen sink, in mind when learning microservices**. Given the breadth of answers provided, perhaps the most inclusive suggestion was to look at the twelve-factor application methodology, since there are at least a dozen things to keep in mind and developers would do themselves a huge service to use the twelve factors as guidance in their implementation. In addition to the 12 factors, be open to continuous learning given the breakneck pace at which technology is evolving.

10 **Other issues to be considered** early in microservices' development include: 1) Why aren't there more competitors in the space? 2) How will we staff, develop, and maintain a microservices architecture moving forward? 3) What's the real cost of maintaining a microservices infrastructure? 4) What's the best way to monitor hundreds of containers with microservices? 5) Moving to serverless, how do we shift identity and architecture best practices?

Tom Smith is a Research Analyst at DZone who excels at gathering insights from analytics—both quantitative and qualitative—to drive business results. His passion is sharing information of value to help people succeed. In his spare time, you can find him either eating at Chipotle or working out at the gym.

Accelerate microservices and API development with tools from CA Technologies.

Faced with the app economy, many enterprises must rebuild applications that need to quickly adapt to changing needs; and the traditional way of rolling out (and supporting) large applications just isn't sufficient. Today's enterprise architects and VPs of applications are wondering:

- How can I deploy and release modern applications in days or weeks, not months or years – and minimize downtime on app updates?
- How can I leverage multiple development teams on different language platforms to build those modern applications?
- How can I scale applications as needs change, while minimizing infrastructure costs to accommodate that scaling?

These challenges stem from an increased focus on agility and scale for building modern applications — and traditional application development methodology cannot support this environment. CA Technologies has expanded full lifecycle API management to include microservices — an integration enabling the best of breed to work together to provide the platform for modern architectures and a secure environment for agility and scale. CA enables enterprises to use best practices and industry – leading technology to accelerate and make the process of architecture modernization more practical.

Today's DevOps and agile-loving enterprises are striving for fast changes and quick deployments. To these companies, the microservices architecture is a boon, but not a silver bullet. Organizations can enable smaller development teams with more autonomy and agility, and as a result, the business will notice IT is more in tune with their changing demands. IT will need to align its API strategy with the microservices that developers produce. Securing those microservices should be of the utmost importance; leveraging API Gateways in this context will benefit IT. And always remember, that if you're looking for speed and scale, safety is equally important — and a strong management component is a must.

WHY ARE MICROSERVICES SO IMPORTANT?

Every digital enterprise trying to thrive in the digital economy is aspiring for two things: speed and scale. If a company's need to get to market faster is critical, it's equally important to be able to scale up appropriately to support increasing customer demand. But the key mantra here is: speed and safety at scale. You can only succeed when you attain speed and scale without losing safety. Agile and DevOps models support decentralized and distributed ownership of software assets and promote faster turnaround of changes and quick deployment. However, to intelligently break down complex, monolithic applications into autonomous units, you need a design strategy, namely, microservices.

By breaking your huge application into microservices, you're enabling your development team to be nimbler with updates and autonomous deployments. This removes dependencies to create large and complex builds, and it eliminates the need for over-sophisticated architectures to step up scalability to meet volume demands.

ca technologies

WRITTEN BY BILL OAKES, CISSP

DIR. OF PRODUCT MARKETING FOR API MANAGEMENT, CA TECHNOLOGIES

CA Technologies provides the proven platform for a scalable, secure microservices solution for the enterprise.

PRODUCT STRENGTHS

- Centralized security enforcement for authentication, authorization, and threat protection
- Routing and mediation to protected resources across various protocols
- Service-level management for enforcing business-level rate limits and quotas
- Service orchestration for reducing service invocations
- Service façades for exposing application-specific interfaces from monolithic back ends

WEBSITE bit.ly/2AnOPMs | **TWITTER** @CAApi | **BLOG** bit.ly/2nCZXz8

Solutions Directory

This directory contains platforms, middleware, service meshes, service discovery, and distributed tracing tools to build and manage applications built with microservices. It provides free trial data and product category information gathered from vendor websites and project pages. Solutions are selected for inclusion based on several impartial criteria, including solution maturity, technical innovativeness, relevance, and data availability

COMPANY	PRODUCT	PRODUCT TYPE	FREE TRIAL	WEBSITE
Amazon Web Services	Amazon EC2	IaaS	Free tier available	aws.amazon.com/ec2
Amazon Web Services	Amazon API Gateway	API gateway	Free tier available	aws.amazon.com/api-gateway
Amazon Web Services	Simple Query Service (SQS)	ESB	Free tier available	aws.amazon.com/sqs
Amazon Web Services	AWS Application Discovery Service	Service discovery	Free tier available	aws.amazon.com/application-discovery
Amazon Web Services	Amazon ECS	Container orchestration	Free tier available	aws.amazon.com/ecs
Apache Foundation	Kafka	Distributed streaming platform	Open source	kafka.apache.org
Apache Foundation	Zookeeper	Service discovery	Open source	zookeeper.apache.org
Apache Foundation	HTrace	Distributed tracing	Open source	htrace.incubator.apache.org
Apache Foundation	ActiveMQ	Message queue	Open source	activemq.apache.org/
Apcera	NATS	Message-oriented middleware	Open source	nats.io
Apigee	Apigee	API gateway, API management	Free tier available	apigee.com/api-management/#/products
Axway	Axway AMPLIFY	API management, API gateway, API builder	Available by request	axway.com/en
Buoyant	Linkerd	Service mesh	Open source	linkerd.io
CA	CA API Management	API management, API gateway	Available by request	ca.com/us/products/api-management.html
Canonical	LXD	Container management	Open source	linuxcontainers.org/lxd/

COMPANY	PRODUCT	PRODUCT TYPE	FREE TRIAL	WEBSITE
Cisco	AppDynamics	Performance and monitoring tool	15 days	appdynamics.com
Cloud Foundry	Diego	Container runtime system	Open source	github.com/cloudfoundry/diego-release
Cloud Foundry	CF Container Runtime	Container deployment and management	Open source	cloudfoundry.org/container-runtime
Cloud Native Computing Foundation	Envoy	Edge and service proxy	Open source	github.com/envoyproxy/envoy
Cloud Native Computing Foundation	OpenTracing	Edge and service proxy	Open source	opentracing.io/documentation
Cloud Native Computing Foundation	containerd	Distributed tracing APIs	Open source	containerd.io
CoreOS	Fleet	Container runtime system	Open source	github.com/coreos/fleet
CoreOS	Flannel	Container orchestration	Open source	coreos.com/flannel/docs/latest
CoreOS	Etcd	Container-defined networking	Open source	coreos.com/etcd
Docker	Docker	Container platform	Free tier available	docker.com/get-docker
Docker	Docker Swarm	Container orchestration and clustering	Open source	github.com/docker/swarm
Dropwizard	Dropwizard	Web services development framework	Open source	dropwizard.io
Dynatrace	Dynatrace	Performance and monitoring tool	15 days	dynatrace.com
Eclipse Foundation	Vert.x	Reactive application development platform	Open source	vertx.io
Elastic	Elasticsearch	Search and analytics	Open source	elastic.co/products/elasticsearch
Elastic	Kibana	Elastic stack configuration and management	Open source	elastic.co/products/kibana
Fluentd	Fluentd	Unified logging layer	Open source	flentd.org
Google	Kubernetes Engine	Container orchestration	$300 credit	cloud.google.com/kubernetes-engine
gRPC	gRPC	Protocol buffers, RPC system	Open source	grpc.io
HashiCorp	Consul	Service discovery, configuration, and monitoring	Open source	consul.io

COMPANY	PRODUCT	PRODUCT TYPE	FREE TRIAL	WEBSITE
IBM	IBM API Management	API management	Free tier available	ibm.com/software/products/en/api-connect
IBM	IBM Integration Bus	ESB	30 days	ibm.com/software/products/en/integration-bus-advanced
Instana	Instana Infrastructure Quality Management	Infrastructure monitoring	14 days	instana.com/infrastructure-management
Istio	Istio	Service mesh	Open source	istio.io
Jaeger	Jaeger	Distributed tracing	Open source	github.com/jaegertracing/jaeger
JHipster	JHipster	Spring Boot and Angular application and microservices development platform	Open source	jhipster.tech
Kong	Kong	API gateway and microservices management	Demo available by request	konghq.com
Kubernetes	Kubernetes	Container orchestration	Open source	kubernetes.io
Lightbend	Lagom	Microservices development platform	Open source	lightbend.com/lagom-framework
Lightbend	Akka	Services communication	Open source	lightbend.com/akka
Lightbend	OpsClarity	Reactive systems monitoring	N/A	opsclarity.com
Macaw Software	Macaw	Microservices development platform	Free tier available	macaw.io
Mesosphere	Marathon	Container orchestration	Open source	mesosphere.github.io/marathon
Micro Focus	Artix	ESB	30 days	microfocus.com/products/corba/artix#
Micrometer	Micrometer	JVM application monitoring	Open source	micrometer.io
MicroProfile	MicroProfile	Java optimization project for microservices development	Open source	microprofile.io
Microsoft	Azure Service Fabric	Microservices development platform	Free tier available	azure.microsoft.com/en-us/services/service-fabric
Microsoft	Azure API Management	API gateway, API management	Free tier available	azure.microsoft.com/en-us/services/api-management
Microsoft	Azure Service Bus	ESB	Free tier available	azure.microsoft.com/en-us/services/service-bus
Microsoft	Azure Container Service	Container orchestration	Free tier available	azure.microsoft.com/en-us/services/container-service

COMPANY	PRODUCT	PRODUCT TYPE	FREE TRIAL	WEBSITE
Mulesoft	Anypoint Platform	Integration platform	Available by request	mulesoft.com
NEC	WebOTX ESB	ESB	Open source	jpn.nec.com/webotx/download/manual/92/serviceintegration/esb
Netflix	Hystrix	Latency and fault tolerance library	Open source	github.com/Netflix/Hystrix
Netflix	Eureka	Service discovery	Open source	github.com/Netflix/eureka
Netflix	Archaius	Configuration managmeent	Open source	github.com/Netflix/archaius
Netflix	Ribbon	Load balancing library	Open source	github.com/Netflix/ribbon
Netflix	Zuul	Dynamic routing and service monitoring	Open source	github.com/Netflix/zuul
Netsil	Netsil	Distributed application monitoring	15 days	netsil.com
Neuron ESB	Neuron ESB	ESB	30 days	neuronesb.com/
NGINX	NGINX Application Platform	Microservices development and management	30 days	nginx.com/products/
NGINX	nginmesh	Service mesh	Open source	github.com/nginmesh/nginmesh
OCI	Grails	Web application framework	Open source	grails.org
OpenESB	OpenESB	ESB	Open Source	open-esb.net
OpenLegacy	API Software	API Management	N/A	openlegacy.com
OpenText Corp.	GXS Enterprise Gateway	ESB	N/A	opentext.com/what-we-do/products/business-network/b2b-integration-services
Oracle	Oracle Service Bus	ESB	Free solution	oracle.com/technetwork/middleware/service-bus/overview
Oracle	Oracle SOA Suite	SOA governance, Integration PaaS	Free solution	oracle.com/us/products/middleware/soa/suite/overview
Oracle	Java EE	Java specifications	Free solution	oracle.com/technetwork/java/javaee/overview
OW2 Middleware Consortium	Petals ESB	ESB	Open Source	petals.ow2.org
Particular Software	NServiceBus	ESB	Open Source	particular.net/nservicebus
Pivotal Software, Inc.	RabbitMQ	Message queue	Open source	network.pivotal.io/products/pivotal-rabbitmq

COMPANY	PRODUCT	PRODUCT TYPE	FREE TRIAL	WEBSITE
Pivotal Software, Inc.	Spring Cloud Sleuth	Distributed tracing	Open source	cloud.spring.io/spring-cloud-sleuth
Pivotal Software, Inc.	Spring Boot	Spring application development platform	Open source	projects.spring.io/spring-boot
Prometheus	Prometheus	Spring application development platform	Open source	prometheus.io
Red Hat	Red Hat Enterprise Linux Atomic Host	Container management	Open source	redhat.com/en/resources/enterprise-linux-atomic-host-datasheet
Red Hat	JBoss Fuse	ESB	Open source	redhat.com/en/technologies/jboss-middleware/fuse
Red Hat	WildFly Swarm	Java EE services development	Open source	wildfly-swarm.io
Rogue Wave Software	Akana API Management	API management	Demo available by request	roguewave.com/products/akana/solutions/api-management
SignalFX	SignalFX	Monitoring, alerts, and analytics	14 days	signalfx.com/products
SimianViz	SimianViz	Microservices simulation	Open source	github.com/adrianco/spigo
Sysdig	Sysdig Falco	Behavioral activity monitor with container support	Open source	sysdig.com/falco
The Linux Foundation	The Linux Foundation	Open source project hosting	Open source	linuxfoundation.org
TIBCO Software Inc.	Mashery	API management	30 days	mashery.com/api-management/saas
Twistlock	Twistlock	Container security	Available by request	twistlock.com
Twitter	Finagle	RPC system	Open source	twitter.github.io/finagle
Tyk.io	Tyk	API management	Open source	github.com/TykTechnologies/tyk
VMWare	Photon	Container-optimized operating system	Open source	vmware.github.io/photon
WSO2	WSO2	API management	Free solution	wso2.com/api-management
X-Trace	X-Trace	Distributed tracing	Open source	github.com/rfonseca/X-Trace
Zapier	Zapier	API management	Free tier available	zapier.com
Zipkin	Zipkin	Distributed tracing	Open source	zipkin.io

GLOSSARY

APPLICATION PROGRAMMING INTERFACE (API)
A software interface that allows users to configure and interact with other programs, usually by calling from a list of functions.

CONTAINER
Resource isolation at the OS (rather than machine) level, usually (in UNIX-based systems) in user space.Isolated elements vary by containerization strategy and often include file system, disk quota, CPU and memory, I/O rate, root privileges, and network access. Much lighter-weight than machine-level virtualization and sufficient for many isolation requirement sets.

CONTINUOUS DELIVERY
A software engineering approach in which continuous integration, automated testing, and automated deployment capabilities allow software to be developed and deployed rapidly, reliably, and repeatedly with minimal human intervention.

DISTRIBUTED SYSTEM
Any system or application that operates across a wide network of services or nodes.

DISTRIBUTED TRACING
A category of tools and practices that allow developers to analyze the behavior of a service and troubleshoot problems by creating services that record information about requests and operations that are performed.

DOMAIN-DRIVEN DESIGN
A philosophy for developing software in which development is focused primarily on the business logic, the activities and issues that an application is supposed to perform or solve.

ENTERPRISE SERVICE BUS (ESB)
A utility that combines a messaging system with middleware to provide comprehensive communication services for software applications.

EVENTUAL CONSISTENCY
A data consistency model used to make distributed applications highly available by keeping data in sync and up-to-date across all services or nodes.

HOLACRACY
A management practice for organizations that are separated into autonomous and independent departments based on roles, which can organize themselves and make decisions based on their duties. Holacracies are focused on rapidly iterating.

JAVA VIRTUAL MACHINE (JVM)
Abstracted software that allows a computer to run a Java program.

MESSAGE BROKER
Middleware that translates a message sent by one piece of software to be read by another piece of software.

MICROSERVICES ARCHITECTURE
A development method of designing your applications as modular services that seamlessly adapt to a highly scalable and dynamic environment.

ORCHESTRATION
The method to automate the management and deployment of your applications and containers.

SERVICE DISCOVERY
The act of finding the network location of a service instance for further use.

SERVICE MESH
An infrastructure layer focused on service-to-service communication, primarily used for distributed systems and cloud-native applications.

SOCIOCRACY
A mode of governance without a centralized power structure, aiming for less independence between teams to focus on organization-wide strategy.

WEB SERVICE
A function that can be accessed over the web in a standardized way using APIs that are accessed via HTTP and executed on a remote system.