Editor: **Cesare Pautasso**
University of Lugano
c.pautasso@ieee.org

Editor: **Olaf Zimmermann**
University of Applied Sciences
of Eastern Switzerland, Rapperswil
ozimmerm@hsr.ch

Microservices in Practice, Part 2

Service Integration and Sustainability

Cesare Pautasso, Olaf Zimmermann, Mike Amundsen, James Lewis, and Nicolai Josuttis

IN PART 1, we discussed microservices definitions, clarified their relation to service-oriented architecture (SOA), and covered service identification from business requirements.[1] It also became clear that once a monolith gets broken up into microservices, architects and developers must deal with many design issues common in distributed systems, as well as related organizational matters. This instalment of Insights takes it from there.

Service Composition and Communication

Cesare Pautasso: Having decomposed a monolith into a set of microservices, how do you compose them back together into user applications?

Nicolai Josuttis: First, we have to clarify how coarse-grained microservices are allowed to be. In my world, you can certainly provide services that cover composed or even complex stateful processes. For example, consider the service of a bank transfer, calling a withdrawal and a deposit (and sometimes a canceling) service. I'd assume that such a transfer service is still a useful granularity for a service and probably even a microservice, although it is composed.

Appropriate granularity is something that evolves according to requirements and experience and should not be dictated by the architectural style. In a large and complex SOA at a major telecommunications company, we had about 30 different services to request customer master data. They evolved and improved over time, and some of them were just compositions of services we already had. One important reason to implement them redundantly was performance.

Mike Amundsen: Therefore, it doesn't help (from the caller's point of view) to distinguish between pure microservices (no dependents, no side effects) and composed microservices since this is almost impossible to see at runtime in a widely distributed system. My experience tells me you will always have composition or aggregation in a healthy, resilient system. Just where it appears and whether or not other components even know which service is pure and which is composed is a matter of style. On the WWW, nobody can tell if the element on the other end is pure (for example, static content) or composed (for example, aggregated content, dependent on remote components, or data stores).

However, a key implementation tenet is to always give the appearance of pure services. In RESTful HTTP [REST stands for Representational State Transfer], one way of doing this is caching. In the reactive style, this is achieved

0740-7459/17/$33.00 © 2017 IEEE

by insisting on asynchronous messages between components—they always return immediately, even if the return is "I'll send you the results when they are ready."

So, I see the process of composing (and decomposing) services as one that reflects the growth and change of the problem space. I always assume the system I am working on is one that will inevitably change over time. The contributions I make to the system (the new components, data storage, interfaces, and so on) are—at some level—transient. That means I design them to be changeable and/or disposable. Today's monolithic user management service might be decomposed into smaller services in a few years. And those elements might be subsequently recomposed into a new, as yet unknown, monolithic component at some future date.

By taking the approach of assuming my work is not written in stone, I find composing services to be quite enjoyable, too. I might not get it right the first time, and that's fine. I'll be able to make changes to the boundaries tomorrow or next week, and so on.

James Lewis: Specifically addressing the question of how to compose services to deliver a web application, I have had a lot of success using the simple rule of "one single synchronous request per page" and then including additional page components asynchronously. It's discussed neatly in *The Art of Scalability*.[2] An example would be the *Guardian* newspaper, where article content is delivered in one round trip but additional components such as the comments are included asynchronously. This can be done using Edge Side Includes [ESI],[3] or you can replace simple placeholders client-side by slapping in the results of another request. Specifically, you just have a little bit of JavaScript that replaces elements in a DOM [document object model] with the results of a call to another resource. At ThoughtWorks, we've done this a fair few times for clients in the UK, and it works pretty well. It also gives you "seams" for replacing components delivering those asynchronous components at a later time.

> Choreographed microservices implement messaging, local state retention, and late binding of replaceable components.

Another approach that has gained traction is the Backend for Frontend pattern described in detail by Sam Newman.[4] Quite simply, you add an API specific to some slice of channels, whether that's having two APIs, one for mobile and one for desktop over your services, or even more fine-grained as Sam describes, one for iOS and one for Android. The benefit here is that you aren't constrained by change rates in APIs outside your team and can concentrate on delivering exactly what you need.

Finally, GraphQL from Facebook and the latest API platform thinking from Netflix are other approaches. GraphQL provides for a single endpoint that different channels query and whose implementation is responsible for parsing the query and aggregating data across multiple services. Netflix has gone toward an implementation based on RxJava, where aggregation is achieved via a scatter–gather approach across multiple services. It's interesting to note that organizations that are operating at the extremes of scale seem to be moving toward reactive models.

Olaf Zimmerman: Ok, let's assume these composition strategies have succeeded. How do you recommend addressing end-to-end data integrity across service interface boundaries?

James: The simplest thing to say is that you follow the same practices that we've always followed. Data should have a single master, a single source of truth for that data.

Mike: Well, many times the systems I work on do not own the data they use. The data is from a third party or a remote team or some other source that we don't control. It is certainly possible for any upstream data processor to concoct well-formed but invalid data. Hence, my approach is to make sure each component validates the data it receives and returns on the basis of that component's own local models. This, too, matches Eric Evans' DDD [domain-driven design] approach[5] that we talked about in Part 1. I don't rely on shared or canonical data models in a system—they never work as expected.

Nicolai: Indeed. If somebody asks for a strict system-level solution for

end-to-end data integrity, then I get a bit nervous. Eventual consistency[6] is a key approach in large distributed systems, which means that this question primarily has to be answered by the business logic, not technically by a protocol or an engine.

James: I agree with Nicolai here. It's simply not possible to guarantee consistency across a distributed system without sacrificing availability. This goes back to the fundamental question of centralized orchestration versus decentralized choreography: If I can, I will decompose a business process such that each participant in the process is decoupled from the other participants. It's surprising how often you can flip the semantics from one of classic orchestration to choreography. In some senses, this goes back to the origins of object-oriented programming (OOP). I love the following quote from Alan Kay: "OOP to me means only messaging, local retention and protection and hiding of state-process, and extreme late-binding of all things."[7] You wouldn't be far wrong if you thought of choreographed microservices as implementing these original properties.

Cesare: Since you mention messaging and late binding: "To ESB or not to ESB" is a concern that predates microservices; is it still relevant?

Mike: I have been able to create resilient systems that contain many microservice components and employ an enterprise service bus (ESB) as a central hub. Labeling these as "not a microservice implementation" doesn't improve anything.

Nicolai: Exactly. Labeling or relabeling never improves anything. That said, I wonder where the microservices paradigm ends. Since the early 2000s, I have categorized ESBs in three approaches:[8]

- distributed (as on the Internet, no intelligence in the infrastructure or network);
- with technical intelligence (protocol mapping, routing, logging, security) in the ESB, but conceptually still transparent; and
- with business intelligence in the ESB (mapping request and response data, composing services into integration flows, and so on).

My understanding is, the microservices community recommends the first and maybe the second, but never the third approach. Which is good.

James: At ThoughtWorks we've been, I think, pretty famously anti-ESB. In part it's a visceral response to the misuse we've seen over the years and the observation that SOA became almost synonymous with ESB. The main objection I've always had is neatly described in Martin Fowler and Jim Webber's keynote at QCon London, "Does My Bus Look Big in This?"[9] They rightly point out, I think, that many organizations were basically sold an illusion: "This will solve all your problems." Of course, ESBs won't and can't; even if you get them working (and that's a big if), all you've done is sweep things under the carpet.

That isn't to say that you don't need to be able to address the technical concerns listed by Nicolai, but these should be explicitly called out as requirements and implemented using the best tools you have available rather than defaulting to "Oh, the ESB will take care of that." I think API gateways run into the same risk at the moment.[10] Obviously, you need to have a solution for authentication, rate limiting, and other basic technical concerns, but as soon as business logic ends up in them, you will get into trouble.

Finally on this point, I think there's another pernicious issue, too—that of coordination of work. When you have these specialized bits of kit with experts that are in charge of them, they very often become competency bottlenecks with large queues of changes piling up in front of them and blocking other teams that require those changes to get work done.

Nicolai: Yes, vendors want to sell ESBs as tools, but just having one doesn't bring you SOA. So my view is that an ESB is more a concept (with different ways to implement it), and any service infrastructure always has an ESB. Even without an ESB, providers and consumers have to agree on a common protocol. And "no ESB tool" or "We only use RESTful HTTP to integrate our microservices" does not imply that things are easy or just plug-and-play.

Olaf: Moving from concepts versus products to technology, is there room for protocols other than HTTP in the microservices design space?

Nicolai: Sure, HTTP is the basic protocol in this distributed world. Even SOAP usually uses it. But we once implemented a service call over FTP. Why not, if it is appropriate?

Mike: There is definitely room for more than one application-level protocol in this world! HTTP is now getting close to 25 years old. I have a hard time believing it will be the only protocol we ever need for decades to come. CoAP [Constrained

Application Protocol],[11] MQTT [mqtt.org], and others are emerging as viable for a world where messages are smaller, memory and power are limited, and Internet connection is spotty.

And who knows what kinds of challenges we'll meet as we start to implement networked software for the interplanetary level. There are already initiatives in this area such as the Licklider Transmission Protocol[12] and a whole host of delay-tolerant communication implementations. Think what it will be like to send packets of data back and forth when it takes hours, even days for a signal to reach the intended target. I suspect we'll see a shift back to large coarse-grained, stateless messages that can be safely replayed over long distances.

> Apply the same ideas about scaling, statelessness, and interoperability to people as to code.

James: Awesome stuff to think about, Mike. I think it's in *Accelerando*[13] that Charles Stross postulated a solution to the Fermi paradox, that as a species evolves it becomes more and more dependent on low-latency, high-bandwidth communication. So, past a certain point, you end up staying at home since latency measured in light years is not appropriate for messaging!

Answering the question, I've used one-way, broadcast, and request–response all at different times and, as Nicolai intimates, on top of either HTTP or a lightweight messaging infrastructure. I already pointed out in Part 1 that to me, microservices aren't an excuse not to think about choices such as a transport protocol or a message exchange pattern (MEP) or even a DAP [domain application protocol]. Such decisions should be made by the teams implementing a particular system. A core part of using an evolutionary approach to building out systems is for the people building the system to agree on the things that are hard to change later. Which DAP or MEP to use would usually constitute one of the things that would be hard to change and therefore should be discussed and agreed early on by the team.

Sustainable Service Evolution

Cesare: How do you evolve microservices?

Olaf: In particular, how do you go about versioning?

James: When I think about versioning, I tend to think of two different things. The first is the build number—what did your build server stamp your artifact with? Then there is your interface version, which is a separate concern and which varies differently from the build number since that's a simple increment. Three things I would immediately think about in this space are applying semantic versioning to your interface, applying the Tolerant Reader[14] pattern when consuming APIs outside your control, and implementing Consumer-Driven Contracts[15] where appropriate.

Mike: As for versioning and evolution, my design approach is to build service components that can be safely evolved by humans (developers) using backward compatibility as the guiding principle. Changes to a service component should never break consumers—that means no changes to the existing interface promises, just additions. That makes it possible to rely on interface consumers that can automatically adapt without the need for human intervention. As long as the ubiquitous language stays the same, even new features can just appear for interface consumers and work fine. However, if the language changes (for example, new domain concepts are introduced), then the system needs to be designed to allow clients to ignore these new domain elements and continue to work successfully.

Nicolai: Yes, we have to understand the nature of interface versioning and what this means for any distributed system using typed interfaces over time. Even microservices can't solve the inherent problems of the requirement to be backward compatible for existing clients while applying modifications due to new requirements. Although Roy Fielding claims "Don't version" (which, by the way, was only meant as a recommendation to avoid version numbers in public APIs), versioning is a real problem, which lies in the nature of distributed interfaces. For enterprise scenarios, I strongly recommend the opposite: Always put explicit version numbers into service names and data types instead of hiding versions in a

new hostname or namespace. This is important for better maintainability. Sooner or later, microservices systems or system landscapes will also have to deal with this problem. Unfortunately, the current hype may lead to the impression that this is not the case. You will learn it the hard way!

Olaf: So semantic interface versioning still is required, and the syntactic means vary by application context. Let's look at the bigger picture now. What are your thoughts about microservice lifecycle management?

Nicolai: I see two options and criteria here. Is the system landscape rather chaotic and self-adjusting (like the Internet), or do we need to understand the relationships between specific business processes, domains, and services (which usually helps to maintain the system and avoid undesired redundancies)? In the former case, we need the strategies that work for the web. In the latter case, we need the same approaches as with any managed SOA—see, for example, *SOA in Practice: The Art of Distributed System Design*.[8] I see nothing new with microservices here (except that the current hype leads to the impression that these problems are gone).

Mike: Lifecycle management, in my experience, must provide the ability to remove obsolete components from the system without breaking the system. This is an area that, I think, doesn't get much discussion in the microservices world, and I'd like to see more of that. For example, I'm working on some experiments that would allow components that are no longer in use to simply remove themselves from the system. Effectively, dying out through disuse.

James: This is where one of the biggest shifts in thinking has occurred. One of the principles that have been rethought in the last decade is the idea of designing software so that it can be reused. Many of the problems I've seen on projects at all sorts of scale have been caused by developers or architects thinking about reuse rather than use, and I think microservices are a reaction to this in part. My feeling is that we should switch to considering replacement rather than reuse.

So, my rather glib answer to the lifecycle management question is that these things should be small enough to be thrown away rather than maintained. This reminds me of Theseus's paradox ("Is it still the same ship?")—we should be able to build systems whose component parts are replaceable over time without having to resort to full-on rewrites of very large systems. This is exactly what companies taking full advantage of this style exploit. What does lifecycle management mean when you completely rewrite components of your system every couple of months because you have learned something new?

Olaf: So, depending on the type of system landscape, the service lifecycles become shorter, and the same functionality might get resurrected in another service. In the light of these increased service dynamics, which organizational scaling strategies do you recommend on the basis of your experience?

Mike: First, it is important to acknowledge that the organization (people and processes) are part of any system you design. So, you need to apply the same ideas about scaling, statelessness, interoperability, and so forth to the people in your system as you do the code. To that end, sources like Mel Conway's "How Do Committees Invent?,"[16] Fred Brooks's *The Mythical Man-Month*,[17] and John Gall's *Systemantics*[18] have all influenced the way I think about and design organizational elements. I encourage everyone working in this field to keep these sources as handy references.

James: Love those references, Mike. To them I would add John Roberts's *The Modern Firm*,[19] Donald Reinertsen's *Principles of Product Development Flow*,[20] and Thomas Allen's *Managing the Flow of Technology*.[21]

Mike: Team size also matters. Robin Dunbar has a social theory about how group size affects overall efficiency.[22] He relates various group sizes (5, 15, 35, 50, and 150) to specific challenges to maintaining group cohesion and effectiveness. A popular version of this approach is Jeff Bezos's "Two-Pizza Team" meme.

James: It all points to the same ideas, really. Create small, autonomous teams, give them the tools to do their job (and I include training here), create a shared goal, and get out of their way.

Nicolai: Or as Kent Beck says, "Start stupid and evolve." SOA in general, even with the constraints of microservices, is an architectural paradigm, not a cookbook. You have a lot of things to decide: which basic protocol and MEPs to use, how much loose coupling, common policies, and so on. And then, to come back to your question, two key questions arise: How do you establish

collaboration, and who is responsible for cross-domain services? At least in a managed system landscape such as an enterprise, this often leads to new departments for solution managers, solution testers, and systems that provide and manage business processes.

James: I would try to avoid the creation of departments that cross boundaries like this. In fact, if I've got one rule of thumb, it's to organize your teams such that they can deliver end-to-end functionality to a set of consumers without work leaving the team boundary. As soon as work leaves your team, your lead time is beholden to others, and that's not where you want to be.

> Evolving self-adjusting systems like the Internet is different from evolving managed enterprise SOAs.

Nicolai: Well, services and teams are only a part of the whole distributed process. Somebody has to be responsible for things as a whole.

Challenges and Outlook

Olaf: Which critical success factors for or inhibitors to a broad, sustainable adoption of microservices do you see?

James: I have a few. The ability to continuously deliver software into production. The ability to easily create infrastructure on demand. The third would be organizational, which we already talked about.

Mike: A theme we explored in *Microservice Architecture*[23] is that the only constant is change. Anyone who thinks that creating a successful IT system means reaching some fixed point on a path is making a mistake. Almost always, by the time you reach your intended goal, things have changed. In fact, I contend the biggest contributor to what is sometimes called technical debt is that systems (both code and organizations) fail to properly deal with change over time.

One of the things I like about the microservice approach is that small teams working on small components can complete a round of changes in days, not months. That means you get feedback sooner. To me, sustainable change means doing it in small increments over a long period of time. And doing it in ways that will not break a running production system.

Nicolai: I absolutely agree. To amend, the key is to understand the nature of distributed systems, especially the inherent problems. This also includes understanding the difference between evolving distributed systems like the Internet and managed distributed systems as with enterprise SOA strategies. This will hopefully help to avoid the impression that microservices solve fundamental problems of distribution without a price.

Cesare: What R&D challenges should our readers work on?

Mike: From my perspective, we need much more work on tooling and design at the system level. While there are dozens of code libraries, IDEs, and frameworks available at the component level, I don't see many valuable tools at the system level. Currently, a running system is still incredibly opaque to humans. This is especially true for systems that span multiple physical locations and are operated by independent teams around the world.

I like the trend of so-called serverless computing[24]—where all the infrastructure of deploying and scaling a service is hidden from developers. But the current crop of these tools is still amazingly crude.

Nicolai: The biggest challenge may be to understand that big systems follow different rules than small systems. Unfortunately, we learn programming mostly in small systems. But programming is in essence today the maintenance and evolution of system landscapes. And there, different rules apply. For example, in a small system, you try to avoid redundancy, can deal with transactions, and are able to define a common data model. In large systems, you need redundancy and have to use compensation instead of transactions, and any attempt to have a common data model is a recipe for failure. Unfortunately, the experience with large systems is hard to teach; we can and have to do better.

James: I think there are a number of challenges. One very interesting area for me is the testing of distributed systems using tools like Jepsen [aphyr.com/tags/jepsen]. I really

like the way the tool includes the clients of the system when perturbing it. Also, I'd like to see if it was possible to identify invariants for microservices automagically, based on data such as request logs. One of the real issues teams struggle with is identifying thresholds for alerts to let them know that something is wrong. Machine learning applied in this space would be super-useful. I also like what Adrian Cockcroft is trying to do with his network simulator. Harvesting real data to produce visualizations like those in his Spigo [github.com/adrianco/spigo] tool would be fun.

Finally, I'm sure there is a load of work to be done looking at business and software architecture isomorphism and how that relates to specifics of flat versus hierarchical team structures. For instance, what's the effect of matrix management on the flow of work and resulting designs?

Mike: I also would like to see more work on improving the autonomy of running systems. The DevOps movement did a great job of automating things from the time code is checked in to the time it is deployed into production. But we have just scratched the surface of automating the way running components advertise their capabilities, enlist other components, and actually complete actions on the network.

Finally, I think there is still much work to be done to help system designers. The process of identifying and defining application domains is inadequate, in my view. We need more work on how to outline and modify context boundaries over time, and more work on how to know when it is time to change these boundaries in order to improve the speed and safety of a running system.

So, I think there are a lot of great opportunities to improve the quality of distributed network applications. I think we're still in the earliest stages of architecture for software and networks. And I'm looking forward to what's ahead.

Cesare and Olaf: Thank you for your invaluable insights and your time to have this inspiring discussion, Mike, James, and Nicolai!

References

1. C. Pautasso et al., "Microservices in Practice, Part 1: Reality Check and Service Design," *IEEE Software*, vol.

34, no. 1, 2017, pp. 91–98.
2. M.L. Abbott and M.T. Fisher, *The Art of Scalability: Scalable Web Architecture, Processes, and Organizations for the Modern Enterprise*, 2nd ed., Addison-Wesley, 2015.
3. *ESI Language Specification 1.0*, W3C note, 4 Aug. 2001; www.w3.org/TR/esi-lang.
4. S. Newman, "Pattern: Backends for Frontends," 18 Nov. 2015; samnewman.io/patterns/architectural/bff.
5. E. Evans, *Domain-Driven Design: Tackling Complexity in the Heart of Software*, Addison-Wesley, 2003.
6. P. Bailis and A. Ghodsi, "Eventual Consistency Today: Limitations, Extensions, and Beyond," *Comm. ACM*, vol. 56, no. 5, 2013, pp. 55–63.
7. A. Kay, "Clarification of 'Object-Oriented,'" *Dr. Alan Kay on the Meaning of "Object-Oriented Programming,"* 23 July 2003; www.purl.org/stefan_ram/pub/doc_kay_oop_en.
8. N. Josuttis, *SOA in Practice: The Art of Distributed System Design*, O'Reilly, 2007.
9. M. Fowler and J. Webber, "Does My Bus Look Big in This?," presentation at QCon London 2008, 6 June 2008; www.infoq.com/presentations/soa-without-esb.
10. "Overambitious API Gateways," Thoughtworks, 2016; www.thoughtworks.com/radar/platforms/overambitious-api-gateways.
11. *The Constrained Application Protocol (CoAP)*, IETF RFC 7252, June 2014; tools.ietf.org/html/rfc7252.
12. *Licklider Transmission Protocol—Specification*, IETF RFC 5326, Sept. 2008; tools.ietf.org/html/rfc5326.
13. C. Stross, *Accelerando*, Ace Books, 2005.
14. M. Fowler, "Tolerant Reader," 9 May 2011; martinfowler.com/bliki/TolerantReader.html.
15. I. Robinson, "Consumer-Driven Contracts: A Service Evolution Pattern," 12 June 2006; martinfowler.com/articles/consumerDrivenContracts.html.
16. M. Conway, "How Do Committees Invent?," *Datamation*, Apr. 1968; www.melconway.com/research/committees.html.
17. F. Brooks, *The Mythical Man-Month: Essays on Software Engineering*, 2nd ed., Addison-Wesley Professional, 1995.
18. J. Gall, *Systemantics: How Systems Work and Especially How They Fail*, Quadrangle, 1977.
19. J. Roberts, *The Modern Firm: Organizational Design for Performance and Growth*, Oxford Univ. Press, 2007.
20. D.G. Reinertsen, *The Principles of Product Development Flow: Second Generation Lean Product Development*, Celeritas, 2009.
21. T.J. Allen, *Managing the Flow of Technology: Technology Transfer and the Dissemination of Technological Information within the R&D Organization*, MIT Press, 1984.
22. R.I.M. Dunbar, "Neocortex Size as a Constraint on Group Size in Primates," *J. Human Evolution*, vol. 22, no. 6, 1992, pp. 469–493.
23. I. Nadareishvili et al., *Microservice Architecture: Aligning Principles, Practices, and Culture*, O'Reilly, 2016.
24. M. Roberts, "Serverless Architectures," 4. Aug. 2016; martinfowler.com/articles/serverless.html.

CESARE PAUTASSO is an associate professor at the University of Lugano's Faculty of Informatics. Contact him at c.pautasso@ieee.org.

OLAF ZIMMERMANN is a professor and institute partner at the University of Applied Sciences of Eastern Switzerland, Rapperswil. Contact him at ozimmerm@hsr.ch.

MIKE AMUNDSEN is the director of API architecture at the API Academy. Contact him at mca@mamund.com.

JAMES LEWIS is a principal consultant at ThoughtWorks. Contact him at jalewis@thoughtworks.com.

NICOLAI JOSUTTIS is an independent consultant. Contact him at nico@josuttis.com.