Exploring Microservices

14 Questions Answered by Experts

Randy Shoup is a consulting CTO and former director of engineering for cloud computing at Google. He was previously Chief Engineer and Distinguished Architect at eBay.

Andrew Phillips is VP of product management for XebiaLabs, the leading provider of software for Continuous Delivery and DevOps. He contributes to a number of open source projects including Apache jclouds, the leading cloud library.

1 Q: How much do Microservice breakdown an n-tiers architecture? Is it just something complementary to what we are doing?

Randy:

Microservice architectures are more of an ecosystem of cooperating independent services than a strict tiered model.

A Microservice implements one functional area and one area only, so imagine taking a vertical slice through all of the tiers for a single functional area, and that is a candidate for a Microservice. Typically, being a service, a Microservice will span the application and persistence tiers.

We have found it to be an anti-pattern to have shared persistence between Microservices — each should have its own (logical) persistence of just what it needs, with no secret back doors for anyone else to circumvent the interface.

Andrew:

In order to avoid a potential confusion I've seen come up when talking about Microservices: when Randy talks about "one functional area", I'm pretty sure that means in the context of a business-relevant service. I.e. not a single, centralized "database access service" through which every type of user-relevant interaction must be funneled. This is exactly the kind of single point of contention that a Microservice architecture attempts to avoid (and is one that characterized many attempts at SOA implementations, in my experience).

Within each Microservice, you may have multiple "tiers", such as a persistence component, a web component, a business logic/service layer etc. It's just that these tiers are encapsulated within the context of one particular business function, and so tend to be smaller and simpler that e.g. the persistence tier of a monolithic applications that must handle all types of data for a very large number of business functions with potentially very different characteristics.

2 Q: If you encourage redundancy of data, then how do you ensure data integrity when a service call fails? Sometimes a business activity must be completed as a transaction across multiple micro services?

Randy:

Whether you are in a Microservice architecture or not, it is best practice to have a single (logical) system that is the system of record for any piece of data, e.g., the user's address or the state of an order. Any other system that remembers that piece of data is operating on a (cached) copy, which might be out of date with the system of record.

The typical way to propagate data changes between different systems is with some sort of guaranteed event system — the system of record produces events, which other systems consume.

Within the bounds of a Microservice, transactions are straightforward to achieve. So the implicit — more difficult — question here is what to do when a business activity spans multiple functional areas, and therefore multiple Microservice requests. Again, a guaranteed event system is a great way to decouple the functional areas, while ensuring that the business activity ultimately completes successfully.

Pat Helland's "Life Beyond Distributed Transactions: An Apostate's Opinion" is a wonderful read about this approach.

Andrew:

Just to add to Randy's comment: since each Microservice will keep a local copy of whatever information it needs, it should not be difficult to ensure the Microservice's data is internally consistent. If the change needs to be propagated to the "source" system of record for that particular dataset (e.g. a user updating their address in their user profile), it is also worth considering whether you can move the responsibility for making that change to the user-facing system.

I.e. the user-facing system makes two Microservice calls, one to the system of record and then (either after a propagation delay or after forcing the second system to refresh its user profile cache for that user) to the second system that needs the updated record. This moves the responsibility of coordination to the most user-facing component, which can make that system more complicated, but makes it much easier to make the "right" decision for that particular application if e.g. one of the calls cannot be completed.

3 Q: If we are modeling the real Microservices world, do we end up with for instance the concept of an 'incomplete order´ in the code?

Randy:

Without knowing the details of your particular use case, it's hard to suggest how to model your problem. If creating an order spans multiple Microservices over an extended period of time, then maintaining a state machine on that order with various intermediate states is a common approach. But don't make things more complicated than they need to be — it might be most straightforward to have an Order Microservice which does everything related to orders. It may be able to do everything itself to create an order, or it may do its work by orchestrating calls to a bunch of other Microservices below.

Andrew:

As Randy points out: If the successful creation of an order is a single business-relevant function, then it would seem like a very good candidate for a Microservice in the first place! If other services (e.g. the shipping service or payment service or whatever) are required to complete all the steps to getting an order out of the door, then indeed the notion of an "order state machine" can help. For example, the payment service could broadcast that a payment for <some ID> was successfully completed, and the order service could listed for that event and update the state for that order to "payment processed."

4 Q: Are Microservices considered more of an architectural strategy? Or an design/implementation style for SOA?

Randy:

Both, I suppose. A Microservice approach has strong architectural implications, in that it keeps components small, and avoids shared persistence.

At the same time, Microservices are SOA done properly — where each service is simple, composable, and self-contained. "Microservice" is a new name for the old concept of encapsulation.

Andrew:

A tricky question, because by now I think there are so many flavors of definitions for both Microservices and SOA, that the answer could be pretty much anything.

In my experience, and irrespective of what we call them, the distinguishing characteristic is that SOA architectures/implementations tended to focus on avoiding duplication of technical functions, so that e.g. instead of multiple applications handling addresses in their own way, you would have one address service that could answer all your address-related questions. This created many points of contention and made it very hard to e.g. move fast with any business service that required a change to the way address were handled, because every other consumer of the address service would need to be tested too.

A Microservice approach tries to rectify this by trying to make business services independent, allowing for some degree of functional duplication (e.g. multiple Microservices maintaining their own copies of address data, cached and possible transformed based on the data from the "owner" of the data) to achieve this.

In short, I would say that, in practice, both SOA and Microservices are architectural styles that emphasize independent services, but have quite a different focus on the types of service that need to be independent: SOA focused on independent technical systems, Microservices focuses on independent business systems.

5 Q: Regarding managing the graph of service dependencies, is there a generally accepted paradigm to help manage that complexity? An API gateway, an enterprise service bus, or something else, for example?

Randy:

In a successful Microservice architecture, most services use several other services, and most are consumed by more than one client service. Looked at globally, there are a lot of dependencies flying around, and it can seem complex.

But let's be clear about what we are trying to achieve by "managing that complexity." If we want to standardize on protocols, formats, security characteristics, etc., a common set of libraries will do the trick. If we want to understand and diagnose what is happening at runtime, a monitoring system should do what we need. If we want to make sure we understand the implications of changes to one of the Microservices, then we only need to consider that Microservice's clients.

It's rare to need to view the entire ecosystem globally (I've actually never needed to do it myself). The way to think about it is that each Microservice should only need to think about its in- and out- dependencies.

Andrew:

To paraphrase what Randy said: you need to understand local dependencies - any real-world Microservice architecture is generally too complex to understand globally, and often exhibits emergent behavior that can be very interesting to study, but is hard to understand.

Making sense of local dependencies applies to both "delivery time" and runtime: runtime visualization is often well taken care of by monitoring tools, especially message tracking tools in environments that use event propagation as an asynchronous communication mechanism between services (see e.g. http://techblog.netflix.com/2015/02/a-microscope-on-microservices.html). Standardized, instrumented protocols really help here since they can provide insight into pretty much any part of the landscape, independent of e.g. which language the services themselves are written in.

At "delivery time", the main challenge is to ensure the "local environment" for the service(s) you are trying to test or deploy is consistent: if you're about to push service A into an environment and you know service A needs to communicate with two APIs of particular versions (you could also say "with two other services", but in general it makes more sense to talk about APIs that are provided or consumed), then it makes sense to a) be able to describe that dependency b) to have tooling that can validate that dependency or, better still, c) automatically ensure that dependency will be fulfilled by e.g. deploying the versions of those services exposing the required APIs to your target environment alongside your app.

Notice that, as Randy said, this is very much in the spirit of understanding the local environment: it does not make sense to try to put together a system that lets you define an "entire test environment", with all the services within it, since such a global definition is very hard to scale and keep up-to-date.

6 Q: Since accessing the database violates the API, how would you send bulk-data from your Microservices to a data warehouse?

Randy:

In a successful Microservice architecture, there is no cheating — you can't go behind the published interface and read or write in the Microservice's persistent storage. But there's nothing wrong with having a bulk data transfer as part of the supported interface, and you may want to use a more efficient transfer protocol than onesey-twosey HTTP calls. Just make sure you treat that bulk-transfer part of the interface with as much care and thought as the "front door."

Andrew:

As Randy said, there is nothing in the "rules of Microservice architecture" (assuming there were some kind of consensus about those ;-)) forbidding bulk data transfer calls from being part of the interface offered by some service. And there is also no requirement for HTTP to be The One And Only communication mechanism (as Randy mentioned during the panel, Google uses a form of RPC very heavily, rather than "vanilla" HTTP). "Mail a hard drive to my data center" might be a little tricky to codify in an interface, but that's about it ;-)

To give an example: Amazon Glacier (a type of storage API for very cheap bulk storage) offers async API calls that may take up to 24h to complete, because at the backend a robot needs to retrieve tapes from storage etc.

7 Q: When you say small changes...does that mean development/tested/deployed to prod because products/features are not small?

Randy:

I am increasingly convinced that Microservices, Continuous Delivery, Agile practices, and DevOps are all facets of a broader approach to creating great software. They all enable and support one another, and are stronger together than apart.

One of the great benefits of Microservices is that individual changes are bounded and easily understood, so it is easier to adopt a process of continuous integration or continuous delivery. So now you can break down that larger project or feature into many small steps, each of which can be understood, implemented, tested, and deployed individually.

Andrew:

I would agree that projects tend not to be "small", but the idea that Features Are Big is precisely one of those preconceptions that is often overturned in successful Microservice/Agile/CD environments. In my experience, we tend to think of features and releases being Big because of the overhead involved in getting them to production: if it takes 25 people in a conference room a whole weekend to deploy something to production, you will not do it simply to change one line of code. But if it's possible to push a feature through the pipeline from idea to prod quickly and without need for manual intervention, features can be small - it's just a matter of training yourself to think of them that way (see also http://blog.xebialabs.com/2014/08/27/continuous-delivery-real-world-move-thinking-releases-cd/).

I heard an example a while ago that I thought explained this well: if you're used to Western eating habits, the idea of not having your "own", large main dish is strange. In many other cultures, it's normal to have lots of smaller portions that are shared between everyone. Once you experience this, you can suddenly conceive of a meal as consisting of many smaller dishes rather than a very few large ones...and learning to think of features in a different way is similar.

8 Q: How do you define a Microservice and then Microservice architecture?

Randy:

Adrian Cockcroft has the best definition I have seen of a Microservice architecture: "A loosely-coupled service-oriented architecture with bounded contexts". I'll add that a Microservice is small, simple, and composable.

Andrew:

I'll just throw the description used in Martin Fowler's article on the topic (http://martinfowler.com/articles/microservices.html) into the mix here: "a way of designing software applications as suites of independently deployable services." One aspect of this that has come up in discussions I've had the past is that Microservices shouldn't have dependencies, and I think we need to be careful with that: "independently deployable" isn't the same as "independent." So while you should certainly be able to boot a Microservice and then have "something" running, it's perfectly acceptable, in my view, for a Microservice to require other Microservices to be available to provide a "useful user experience".

In other words, standing up a Microservice should allow you to test its interface, and it should work and respond something sensible. You may well need to boot up multiple, related Microservices in order to be able to run through realistic user interaction scenarios, though.

9 Q: What would you choose as a secure and practical way to communicate between Microservices to ensure their independency while keeping a reasonable performance?

Randy:

I've seen lots of successful technologies, from proprietary (Google) to standard (most other places). Most common for synchronous request / response is probably HTTP / REST / JSON. Async techniques are less standardized..

Andrew:

I think Randy has already made what I would consider the most important point: there is no One Best Choice, but whatever you choose, try to standardize on the protocols used as much as possible, as this makes management and monitoring of your overall landscape much easier. This is worth bearing in mind when considering reactive/async communication styles since, as Randy said, there aren't yet any obvious standards there - different languages and frameworks tends to use their own messaging formats and transports.

10 Q: To me, Microservices sound perfectly suitable for Test driven development (is it actually mostly the preferred pattern in developing Microservices)?

Randy:

Microservices lend themselves very well to TFD / TDD, because the Microservice is a natural testing boundary, and is simple enough to be tested pretty completely. It's my preferred way to develop. Google has a particularly good culture and practice around automated testing.

Andrew:

To me, this gets back to Randy's earlier observation that Agile/CD/TDD/Microservices are all "different parts of the same elephant", so to speak. Microservices should allow you to make small, relatively isolated changes, which in turn means that it should be easy to test whether those changes work as expected. Since Microservices work very much on the principle of exposing their functionality via published interfaces/APIs, testing Microservices also lends itself well to automation.

What becomes more challenging is figuring out how to test entire user scenarios that rely on multiple Microservices communicating with each other - not necessarily because the testing itself is different from "traditional" functional testing, but because you need to coordinate more moving parts to get a test environment running in the first place. On the other hand, if you can instrument the various calls/events/messages to and between services appropriately (see the earlier question about which communication mechanisms to use), tracking down the causes for test failures can become a lot easier.

11 Q: Would you consider a governance structure or micro services library to keep the services "clean"? Also taking into account SLA's of services?

Randy:

"Governance" has two implications — the first I think works well and the second not so much.

The first implication, which I like, is the standardization of good practice. As you suggest, this is best achieved through having a (supported) set of common libraries for building a Microservice. Make it easy to do the right thing, and more work to do the wrong thing. The NetflixOSS set of projects is a great place to look for battle-tested libraries for Microservices built on the JVM; we leveraged them extensively at KIXEYE for the "service chassis" we used to build Microservices. Ultimately we were able to go from no code to a running service in AWS in 15 minutes.

The second implication, which I have not seen work well, is top-down control. This approach to the world tends to make whatever enforcement arm you design the development / deployment bottleneck, and runs counter to the "autonomy and accountability" philosophy of Microservices. If you have a few experienced people who can help other developers avoid pitfalls, use them to build libraries rather than be approvers and governors — encode their experience in actual code!

On SLAs, I am a big fan. The most effective Microservice organizations make explicit contracts (SLAs) with their clients about what the client can expect from the service. If you send me 10K requests per second of size 10KB, you can expect latencies of X at the 99%ile, Y at 99.9%ile, etc. Both client and provider of a Microservice should agree on what they are up for.

Andrew:

Definitely agree with Randy that SLAs, i.e. the notion of a "contract" that your service must fulfill that goes beyond the interface itself to actually include non-functional aspects such as performance, is critical. Trying to reason even about the "local neighborhood" of a Microservice is very difficult if each service has unpredictable or unknown behavior patterns, and makes it hard to provide predictability about what really matters: an excellent experience for the end users of your services.

12 Q: Very granular services may result in information flow redirected to people/service registry/QA to cover multi service use cases. What is your take on that?

Andrew:

If you mean that granular services require e.g. QA to spend more time testing "multi-service use cases", then I would say that yes, that is probably correct. But I think the assumption that this causes difficulties is not necessarily correct: from a user perspective, the experience should be very much the same, so the testing should also be the same.

Yes, getting a test environment up and running will require more moving parts, but since each part is simpler that does not mean that such an environment needs to be harder to set up, manage and configure. Also, the fact that all interactions should happen via interfaces/APIs makes many tests easier to automate, and (see the answer to a previous question) instrumentation of the messages/communication between services can make failure determination quite a lot easier.

13 Q: How do you know that all the major companies use Microservices - is there a paper that published this?

Randy:

If you search around, you can find resources that discuss many of the uses of services by large companies: e.g., Amazon, Twitter, and Netflix. You can check out Netflix's code on github at NetflixOSS.

Andrew:

"All the major companies" is obviously a very broad claim, and there will certainly be some that don't use Microservices, or only in a minor way. What I think really matters, though, is that enough companies across all kinds of industries have been able to address some critical challenges by moving to a Microservice architecture for it to be an approach you should at least consider (see also this recent presentation by Adrian Cockcroft: http://www.slideshare.net/adriancockcroft/monktoberfest-fast-delivery)

One thing I should add at this point that I think can't be repeated often enough: don't do Microservices for the sake of doing Microservices. Like any other change to your application architecture, processes and organization, it carries risks, so you need to feel that the undoubted benefits that it can deliver are actually relevant for you.

If you are in a situation where it's practically impossible to make even small changes to your system, and there is an actual business need to make changes more and more frequently, or if you have hit limits regarding the scalability of your monolithic applications and you know you will need to scale quite a lot further, then it really does make sense for you to consider Microservices.

14 Q: What is the difference between a webservice and a Microservice?

Randy:

They are orthogonal concepts. A web service is a service exposed through a web protocol like HTTP. A Microservice is a service that is small, simple, and composable. Many Microservices are, in practice, web services. But only some web services are Microservices.

Andrew:

...And just to emphasize one thing Randy said: not every Microservice has to be a webservice. The choice of communication protocol for your Microservices will depend on what they do. See earlier answers that discuss this in more detail.

About XebiaLabs

XebiaLabs gives you the tools you need to deliver higher quality software, faster, at scale. Our solutions for DevOps and Continuous Delivery bring speed, clarity, and simplicity to your software delivery practice. Founded in 2008 and headquartered in Boston, XebiaLabs has a worldwide network of sales offices and partners.

For more information, please visit www.xebialabs.com.

XebiaLabs products are designed to help DevOps professionals achieve true Continuous Delivery at scale.

XL Release lets you manage, control, automate and visualize Continuous Delivery pipelines.

XL Deploy lets you automate your application deployments so releases can occur in a repeatable, standard and efficient way.

XL Test is the first test management tool that allows you to centrally control and report on all tests relevant to your application's quality.

→ Try Free Community Editions