Answering questions (from Devoxx) on Microservices

Posted on Jun 22 2015

I'm at Devoxx Poland (http://devoxx.pl/) this week, and my session (http://cfp.devoxx.pl/2015/talk/QRA-8722/Principles_Of_Microservices) was first up after the keynote. I got a load of questions via the Q&A system which I couldn't answer during the talk and which require more space than Twitter allows to respond to, so I thought I'd reproduce them here along with my responses.

I've tweaked some of the questions a bit for the sake of brevity, but I hope I haven't misinterpreted anything.

Sharing Code

I had a few questions on this, including:

> *Is it OK to share some code between multiple microservices? - Mariusz*

And:

> *What is the best practice for sharing object like 'customer' between bounded contexts: model copy-paste, shared lib or just correlation Id? - Piotr Łochowski*

In general, I dislike code reuse across services, as it can easily become a source of coupling. Having a shared library for serialisation and de-serialisation of domain objects is a classic example of where the driver to code reuse can be a problem. What happens when you add a field to a domain entity? Do you have to ask all your clients to upgrade the version of the shared library they have? If you do, you loose independent deployability, the most important principle of microservices (IMHO).

Code duplication does have some obvious downsides. But I think those downsides are better than the downsides of using shared code that ends up coupling services. If using shared libraries, be careful to monitor their use, and if you are unsure on whether or not they are a good idea, I'd strongly suggest you lean towards code duplication between services instead.

Communication

> *How microservices should talk to each other? Is service bus best way to go? Can we somehow do p2p connections without loosing decoupling? - Dominik*

Don't start with technology, start with what sort of interactions you want. Is it request/response you're after? If so, synchronous or asynchronous? Or does event-based collaboration make more sense in this context? Those choices will shrink your potential solution space. If you want sync request/response, then binary RPC like Thrift or HTTP are obvious candidates. If using async request response some sort of message passing tech makes sense. Event-based collab is likely to push you towards standard messaging middleware or newer stuff like kafka. But don't let the tech drive you - lead with the use case.

> *What's your take on binary protocols? - Tomasz Kowalczewski*

I prefer textual protocols for service-to-service comms as they tend to be more hackable and they are less prescriptive (generally) in terms of what tech services need to use. But they do have a sweet spot. Protocols like Thrift (https://thrift.apache.org/) and Google's Protobuffers (https://github.com/google/protobuf) that make use of schemas/IDL make it easy to generate client and server stubs, which initially can be a huge boon in getting started. They can also be quite

lean on the wire - if the size of packet is a concern, they may be the way to go. Some of these protocols can be terrible though - Java RMI is very brittle for example and requires both client and server are Java-based systems. But start with what interactions make sense as above, then see if a binary protocol fits in that space. And if you do decide to go this route Protocol Buffers or Thrift are probably where you should start as they handle expansion changes pretty gracefully.

Size

> *How big should be microservice? - Prakash*

And:

> *Can you give some advice on how big a micro service should be? - Marcin*

42.

Seriously though, it does depend. LOC don't help us here as different languages are more or less expressive. I'd say how small a service should be for you is more down to how many services you can manage. If you are fairly mature in your automation and testing story, the cost of adding additional, smaller services might be low enough that you can have lots of them, with each one being pretty small. If everything is deployed manually on one machine with lots of manual testing, then you might be better off with a smaller number of large services to get started with.

So 'how big is too big' will vary based on your context.

Ownership

> *Do you agree with: "one microservice per team"? - Prakash*

In general, yes. Services need clear ownership models, and having a team own one or more services make sense. There are some other ownership models that can work where you cannot cleanly align a service to a single team which I discuss in more detail in Chapter 10 of the book (/books/building_microservices/), and also on this site where I discuss different ownership models (/blog/2015/04/08/capturing-patterns/).

DDD & Autonomy

> *What's the key difference or decision making fact considering domain driven model and decentralization. seem to be contradictory when defining model & types - anonymous*

Good question! I see DDD as helping define the high-level boundaries, within which teams can have a high degree of autonomy. The role of the architect/team leads is to agree the high level domain boundaries, which is where you should focus on getting a consensus view. Once you've identified the boundaries for services, let the teams own how you handle implementation within them. I touch on these ideas when discussing evolutionary architecture in Chapter 2 of the book (/books/building_microservices/).

Authorisation & Authentication

> *What is the Best way to profile one authorisation system for all Microservices? - Antanas*

Technically, you can do this using a web server local to the microservice to terminate auth (authentication and authorisation), so using a nginx module to handle API key exchange or SAML or similar. This avoids the need to reproduce auth handling code inside each service.

The problem is that we don't really have a good fit for a protocol for both user auth and service-to-service auth, so you might need to support two different protocols. The security chapter of the book goes into more depth, but this is a changing field - I'm hopeful that OpenID Connect (http://openid.net/connect/) may be a potential solution that can help unify the worlds of human and service-to-service auth.

Transactions

> *How to go about transactions within microservice world? - anonymous*

Avoid! Distributed transactions are hard to get right, and even if you do can be sources of contention & latency. Most systems at scale don't use them at all. If you're decomposing an existing system, and find a collection of concepts that really want to be within a single transaction boundary, perhaps leave them till last.

Read up on BASE systems and eventually consistency, and understand why distributed locks (which are required for distributed transactions) are a problem. To move from a monolithic system to a distributed system may mean dropping transactional integrity in some parts of your

application, which will have a knock-on effect on the behavior of your system. That in turn can lead to difficult conversations with your business. This is not easy stuff to do.

Maintenance

> *When system goes into maintenance, isn't it more difficult to handle "polyglot" micro services than an ugly monolith using the same stack of technologies? - Paweł J*

It can be. If the service is small enough, then this problem can be mitigated, as even if the tech stack is alien for the person picking it up they can probably grok it if it's a few hundred lines of code. But it is also a trade-off that each organisation needs to make. The benefits of polyglot services are that you can find the right tool for each job, embrace new tech faster, and potentially be more productive as a result. On the other hand in can increase the cost of ownership. Which choice is right for you should be based on the goals of your company, and your context. What's right for one company isn't right for all.

NodeJS

> *So we should more carefully go to nodejs? - anonymous*

Personally, I am not a fan. Operationally, many people picked up NodeJS for microservices due to its ability to handle large numbers of concurrent connections and very fast spin-up time, an itch that Go scratches for me along with other benefits I don't get from Node. That said there is no smoking gun that says to me that NodeJS is a bad choice. Debugging is difficult, error handling can be fraught, and of course there is NPM and the crazy pace of change in the build tool and general library space, but lots of people use it, and love it, including many colleagues and clients.

So please feel free to put down my dislike of nodejs to a personal preference thing!

User Interfaces

> *In what way UI could be decomposed to microservices? - anonymous*

Some argue that each microservice should expose a UI, which is then aggregated into a whole. For many reasons I think this pattern works in a very small number of cases. More often you'll either be making API calls from the API itself to pull back data/initiate operations, as this gives much more

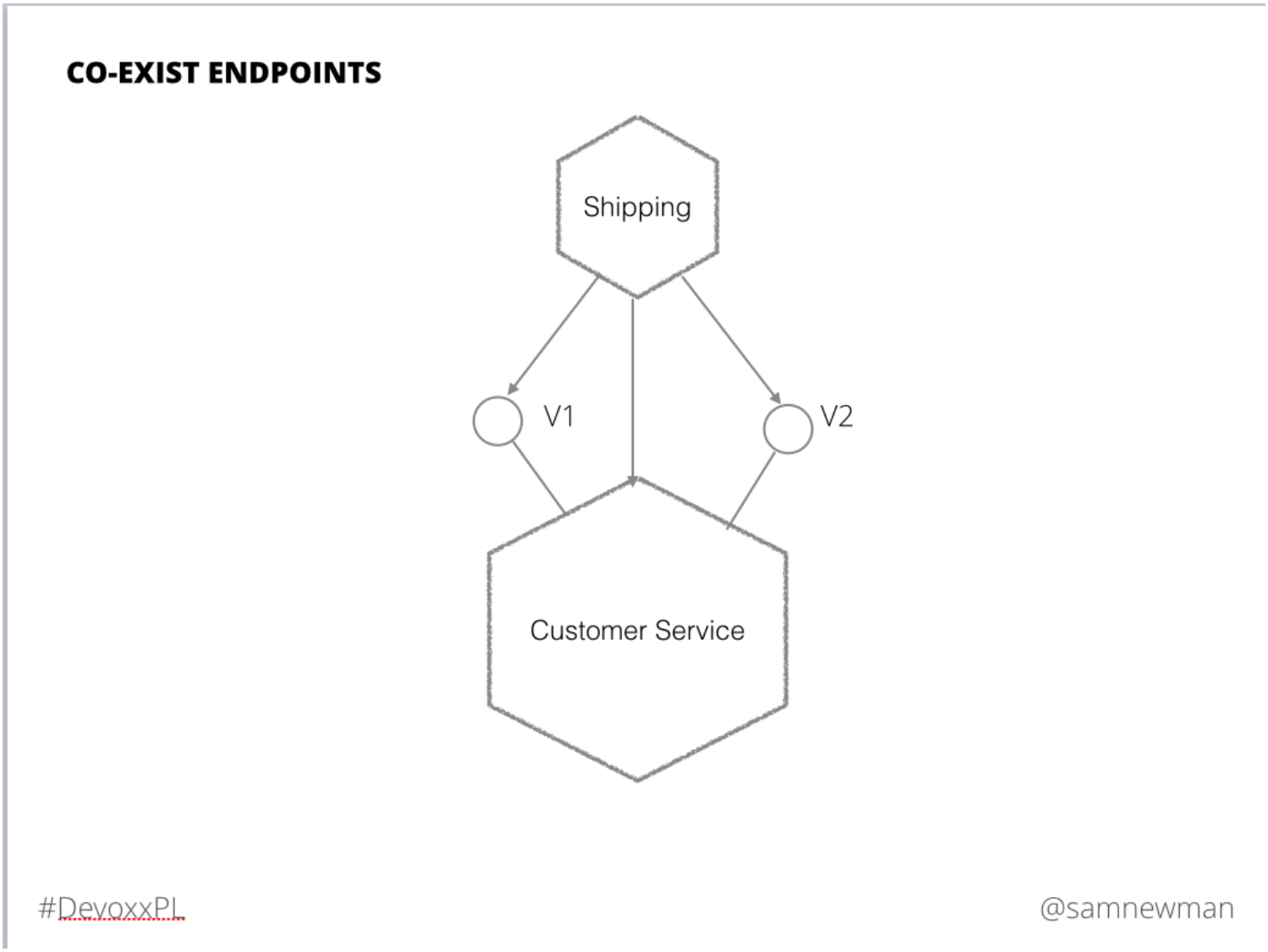
flexibility in terms of UI design.

In general, view UIs fundamentally as aggregation points. Think of your services as exposing capabilities, which you combine in different ways for different sorts of UIs. For web-based UIs this means making API calls to services to perform the required operations and return the appropriate data. There is lots of nuance here - this is unlikely to work well for mobile. Here using aggregating backends for your UIs can be required. I need to write up some patterns on this (I discuss them briefly in chapter 4 of the book) but it is an emerging space where there is no one right answer.

Versioning Endpoints.

> *Why you version your endpoints on uris (v1,v2 example) not on accept header level ? - anonymous*

Here is the slide in question:



A slide form my 'Principles Of Microservices' talk where I show co-existing endpoints

This wasn't meant to define any specific implementation, and how you implement this pattern will depend on the underlying technology. For RPC this may need a separate namespace, or even a

separate RPC server bound to a different port. For HTTP you have many options. You could do this using an accept header for example. However I *tend* to prefer having a version number explicitly in the path as it is much more obvious, and can be more straightforward to route., but I know many purists might be horrified at that thought!

Caching

> *Where to cache? Client or service itself? - anonymous*

Cache in as few places as possible to start with. Caches are an optimisiation, so treat them as such, and only add them when needed. In general, I prefer caching at the server, as this centralises the cache and can avoid the need to cache the same data in lots of places, which can be efficient. However depending on the performance characteristics you are trying to meet, and the problems you have, this may or may not be the right answer. Sometimes you might want to do both!

Do though make sure you have metadata on any entities/resources you expose via APIs to aid caching, especially if you want to enable client-side caching. HTTP has very rich semantics for example to support this.

Performance

I also got a question via Twitter which I thought I'd fold in here too...

> *microservices increase latency due to network calls. How to solve for it? - @Aravind Yarram (https://twitter.com/yaravind/status/612953741944324096)*

Well, in a way you can't. You may be replacing what were in-process method calls with calls over the network. Even if you pick lightweight protocols and perhaps even go for non-blocking async calls, it will still be much slower, potentially by several orders of magnitude. We need to mitigate this by avoiding chatty services, and local caching, but even so there are limits.

I would turn this around. How fast does your app need to be? How fast is it now? By moving to microservices the additional latency due to network calls might end up slowing the app down (assuming this call is on the critical path), but if your app is still fast enough, that's OK. Microservices give, and they take away. Introducing them may make some aspects of your application slower. It's up to you to judge if that is an issue, and if you get enough benefits from them to make things worthwhile.

→ **Back to Blog (/blog)**