Intellyx White Paper

Challenges of Microservices Architectures

Jason Bloomberg

June 19, 2015

The Architecture Buzzword du Jour

Move aside, big data. Sorry, devops. Cloud computing? You're yesterday's news. Today, the buzziest buzzword du jour is *microservices*.

Given the rather rocky road that SOA and Web Services took over the last decade, it's somewhat surprising that people are ready to take another go at the notion of *service*. But ready we are – in spite of the fact that just as with Web Services, architecture is the central challenge.

In some ways, microservices architecture is "SOA done right" – but there's more to it, as we now have the principles of cloud architecture (in particular, elasticity and automated recovery from failure) as well as another hot new buzzword, *containers*.

To understand microservices architecture, therefore, we need to do more than brush off our old SOA books. We're not simply repeating history, hoping to get it right this time. In fact, we're actually moving forward into new territory, as we finally hammer out how all these new buzzworthy trends actually fit together.

Oh yes – let's not forget to add another buzzword to the list: Internet of Things (IoT). It turns out that microservice architecture is especially well suited to the IoT. You shouldn't be surprised!

Defining Microservices

I like to define a *microservice* as a *parsimonious, cohesive unit of execution*. By *parsimonious* I mean that each microservice should be as small as it should be – but no smaller. In each iteration, refactor your microservices, either to trim them down or split them into smaller microservices. Rinse and repeat until you're done.

By *cohesive* I mean that we should design each microservice to do only one thing and to do very well. Cohesion has been a software engineering best practice since the Smalltalk days, of course – only many people lost their way during the Web Services era, as they crammed dozens of operations into each WSDL file. It's time to get back to basics.

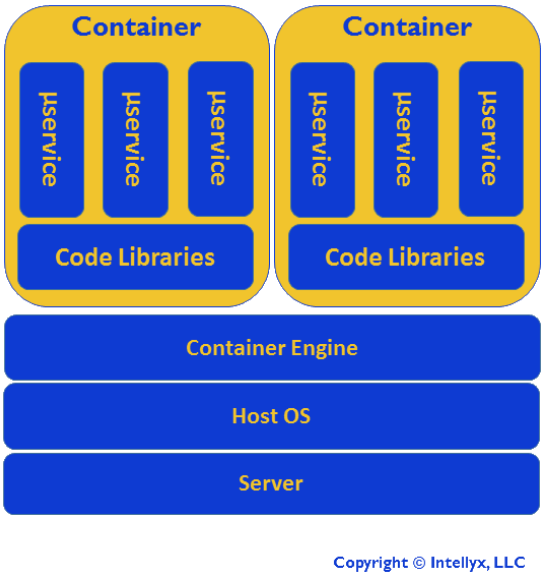
By *unit of execution* I mean that microservices contain everything from the operating system, platform, framework, runtime and dependencies, packaged together, as shown below.

The Components of a Microservice

Just which bits of the runtime, cache, or operating system you should include in a microservice (or µservice, for you fans of Greek) depends upon the purpose of the microservice as well as its environment.

In many cases, that environment is a container – although microservices don't require containers, and containers don't require microservices either, for that matter. But it's no coincidence that microservices fit very nicely into containers.

Containers support lightweight, rapid scalability and elasticity. They also help to keep track of the various bits and pieces that make up each microservice. The figure below illustrates how microservices fit into a container environment.

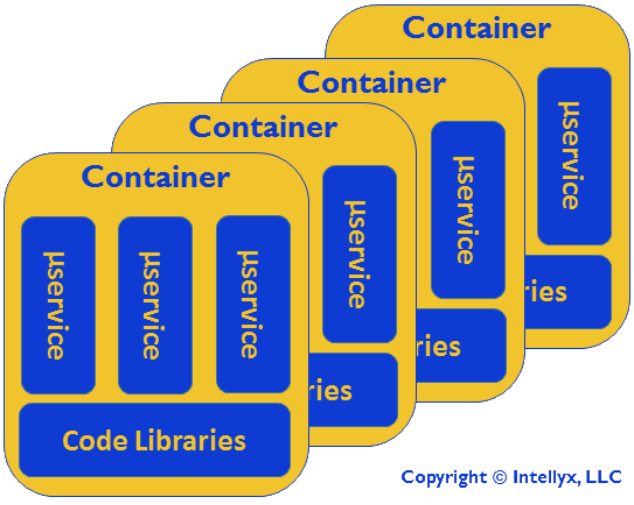


Microservices in a Container Environment

In the illustration above, there are three microservices per container, but it could easily be as few as one or as many as you like. The advantage to putting more than one microservice in a container is that you can leverage shared code libraries across those microservices – and the container will automatically manage those libraries for you. The downside, however, is that you introduce a level of coupling among the microservices in the container. Essentially, you must create and update them as a group.

In addition to managing the shared code libraries, containers also give you rapid autoscaling – far faster than the traditional VM-based autoscaling familiar from standard cloud environments. Think milliseconds instead of tens of seconds: fast enough for rapid, automated scale ups and scale downs as part of the normal behavior of the overall application.

The figure below illustrates this autoscaling:



Container-Based Autoscaling of Microservices

In the figure above, treat each of the containers as essentially identical, where the number can go up or down as needed, based upon the performance demands of the broader app. As a result, there are a (flexible) number of identical microservice instances for each microservice, one instance of each microservice per container.

Remember, however, that microservices have their own caches. If a microservice is keeping track of something in its cache and the microservice goes away or a request is routed to a different microservice instance, what happens to the cached information?

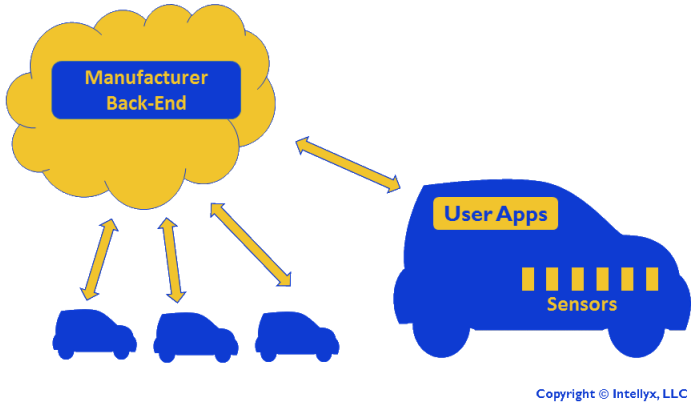
Welcome to one of the primary microservice pitfalls: dealing with state information. Caching within each microservice instance is local to that microservice instance. Resist the temptation to use those caches for anything else.

Furthermore, storing state information in the persistence tier (in other words, the underlying data stores that support all the microservices) doesn't scale well. When there's a reason to persist state information, for example, if you want to store session state for a user so they can pick up where they left off, then you'll need to store it in a persistent data store somewhere.

Otherwise you should maintain state information in the message interactions between microservices and between the microservices and everything else they interact with. (As it happens, maintaining state information in messages is why REST is *representational state transfer*, but that's another story.)

The IoT Example

It's finally time for our IoT example that illustrates microservices in action. In this hypothetical scenario an automobile manufacturer rolls out one or more connected car models. Each car has several user apps (GPS, etc.), as well as several sensors that upload telemetry data to the manufacturer's cloud, as shown in the figure below.



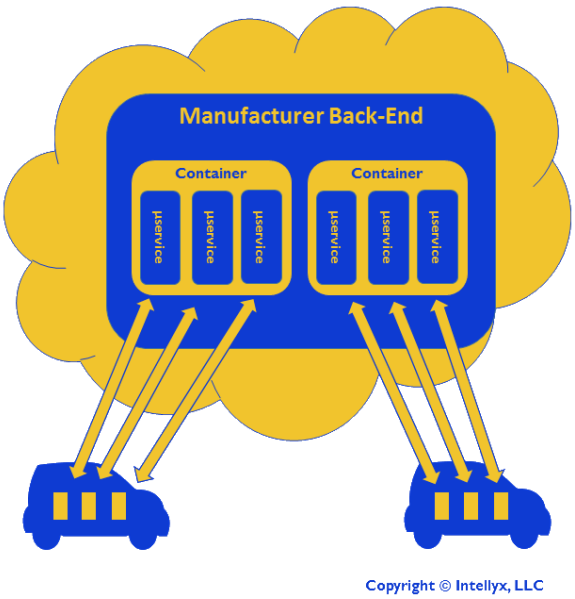
Connected Car Scenario

The central question in this scenario is how best to architect the manufacturer's back end application, both for scalability as well as for agility, as the manufacturer will want to update software on a frequent basis. Key questions: does every car share the same SaaS app? And how practical is multitenancy in this scenario?

With a traditional application (even if it has an n-tier architecture), scaling the manufacturer back-end will be challenging. Simply moving it to the cloud as a single SaaS app helps with scalability somewhat, but we still have problems if the manufacturer wants to update different parts of the app to support different auto models, for example.

Building a multitenant SaaS app sounds like it might address these issues, but in this situation, we still have scalability or agility problems, depending upon how granular our tenants are. VM-based multitenancy may work for hundreds or even thousands of tenants, but one tenant per car is pushing the limit of traditional VMs.

That's where containers come into the story. If we use containers and microservices, we have the luxury of instantiating one container (or in reality, one container autoscaling group) per car, as shown in the figure below – an example of container-based multitenancy.



IoT Scenario with Containers and Microservices

Furthermore, we can even go so far as having one microservice per individual sensor. Yes, modern container technology can rise to this challenge, even for hundreds of thousands of cars. And yet, whether we *can* and whether we *should*, however, are two different questions.

Clearly, if we have one microservice per sensor, then we will have a boatload of individual microservices leading to inevitable management challenges. Furthermore, we will need to be careful about our communications link between the automobiles and the cloud, as that connection has limited bandwidth and is relatively expensive, as it will likely depend upon a 4G link (or better).

Therefore, if we do decide our architecture requires so many microservices, we will need an intelligent integration infrastructure like the SnapLogic platform that manages the telemetry, perhaps by bundling separate messages and compressing them, or some other technique for optimizing our message traffic.

Regardless of how we handle this problem, however, the secret is to abstract the integration technology so that we don't have to worry about it when we build, manage, and update our microservices. Traditional ESBs are clearly not up to the task.

That being said, is there a good reason to have one microservice per sensor in the first place? Perhaps we might have one microservice per car to handle telemetry from all sensors (as well as interactions from the user apps on the car, perhaps).

Or we may decide to have one microservice handling telemetry from many cars, perhaps dividing up traffic from groups of cars to different microservices in an autoscaling group following a sharding scalability model.

The answer to this question, of course, is *it depends*. The appropriate architectural decision must take into account many factors, including the size and frequency of the telemetry and what sort of data processing each microservice is supposed to handle.

Within this context, however, keep in mind the principles of parsimony and cohesion: make each microservice as small as it can be, and have each microservice do one thing well. Other things being equal, therefore, choose the one microservice per sensor option over any of the other options.

The Intellyx Take: Shifting the Burden

By paring down the capabilities of each individual microservice, we externalize the responsibility for many aspects of our application: scalability, management, and integration in particular.

The container infrastructure (as well as the cloud environment it generally belongs in) will handle the scalability for the microservices. Management – the full spectrum of application and infrastructure management – becomes increasingly important, as we could potentially have vast numbers of microservices working together.

And finally, we've raised the bar on integration as well. Poor integration practices in an environment of so many microservices could easily bog down our network, open up security holes, or create other issues.

Using integration technology that is itself constructed from microservices as well as being inherently cloud friendly, like the elastic integration from SnapLogic, is also critically important.

Abstracting the underlying message interactions without introducing bottlenecks, therefore, is an important task that teams should handle proactively. And remember as well that we should be managing state information in messages too – another task that falls to the integration infrastructure to handle in a seamless, lightweight manner.

Don't fall for the microservices shell game. True, each microservice is simple to build and deploy, but microservices architectures can be every bit as complex and difficult to get right as SOA has always been. Organizations that do get their microservices architecture right, however, will be well-positioned for whatever the modern digital, IoT world can throw at them.