**Microservices dilemma: Convert a monolithic solution or start from scratch?**

[microservices](https://developer.ibm.com/bluemix/tag/microservices/)



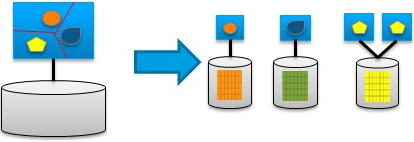
[DougRothert](https://developer.ibm.com/bluemix/author/drothert/) / November 24, 2015 / [0 comments](https://developer.ibm.com/bluemix/2015/11/24/microservices-architecture-convert-monolithic-solution-or-start-from-scratch/#respond)

The buzz around microservice architectures is clearly building. More and more people are discovering the benefits of splitting large monolithic services into smaller microservices. These microservices have a standardized interface that enables them to tolerate individual service failures and updates without catastrophically impacting the overall business solution.

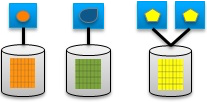
People want to know how to get started. There are excellent resources available, but you first must decide whether to build a monolithic solution (Brownfield) and convert it to a microservices architecture, or to build a microservices architecture from scratch (Greenfield). There are strong opinions on both sides of this issue, including Martin Fowler’s [MonolithFirst](http://martinfowler.com/bliki/MonolithFirst.html) and Stefan Tilkov’s [Don’t Start with a Monolith](http://martinfowler.com/articles/dont-start-monolith.html). In this post, I will distill some of their arguments, find common ground, and present my own perspective for people starting to use microservices.

**Contrasting “MonolithFirst” and “Don’t Start with a Monolith”**

There are two key arguments against starting with microservices in “MonolithFirst”: that microservices add potentially unnecessary complexity to new projects that usually need rapid evolution, and that it is extremely difficult to accurately determine the boundaries of each service when you start a project. Fowler says that if you have a working monolithic service, you can deconstruct it into microservices either by pulling it apart incrementally or all at once, noting the natural boundaries:

  
*Should I try to make a monolith first and then break it into microservices?*

“Don’t Start With a Monolith” states that it is very hard to split an existing monolith into microservices if it was not designed to be split. Tilkov points out that monolithic services are often tightly coupled with loosely defined interfaces, which makes the redesign to split them into well defined microservices extremely difficult. The best way to ensure that your microservices have well defined boundaries is to design that into the architecture at the beginning. Ideally, you are experienced with the domain and can use that knowledge to make the correct architectural decisions when you build a second version of the system:

  
*Or should I just try to build a microservice architecture without it?*

Both of these arguments center on the boundaries between services—what is the scope of each service and what interfaces define it. This is the common ground, and having these boundaries well defined is crucial for successful microservices. Fowler argues that working with an existing implementation is the best way to find this, while Tilkov states that it must be planned from the start, ideally with enough domain knowledge to avoid serious mistakes.

**Learning the most from both views**

My perspective is that real world situations and systems require aspects of both. I lean towards Tilkov’s perspective, but ask yourself some questions about the microservice architecture that you intend to create and use the answers to help take their advice more effectively.

* **Think about your existing assets.** Do you have a monolith to start working from? If the answer is no, your only option is to pursue a Greenfield approach. Take a serious look at all open source projects in your domain—successful and not— to learn what has and hasn’t worked before you build your model. This can help to define the right boundaries before investing your own time and money
* **Take a hard look at your monolith (if you have one).** What does the monolith look like both conceptually and underneath the covers? Specifically, does it have well defined boundaries between its components? Can you clearly see the scope of each component and a standardized set of interfaces that make sense? If this is the case, you may be able to refactor some of your existing monolith into a set of microservices
* **Take a hard look at those components of your monolith (if you have one).** Of your existing components, which are critical? Which are unstable? Which change frequently? If you have components that are critical, stable, and infrequently need change, don’t try to change them. Typically you want to start with an incremental replacement approach, first focusing on components that are non-critical, unstable, and prone to change. If they have well defined boundaries, they are good candidates to convert to microservices because they will benefit the most in the long run.

When weighing the points above, remember that a good microservice is one that not only does its job effectively but has a [limited blast radius](https://en.wikipedia.org/wiki/Blast_radius) when it fails, allowing the overall business solution to work.

**Changing an architecture is a journey**

As you get started with a microservice architecture, remember that adopting a new architecture style is usually an ongoing process. At first you should concentrate on the small wins that can build confidence and support a standard approach that you can repeat. Try to read as much as you can on microservices (including the resources listed below) to broaden your perspective. One of my favorite expressions is [“your mileage may vary”](https://en.wiktionary.org/wiki/your_mileage_may_vary) because it neatly expresses that individual experience is unique, and your path to microservices will be no exception.

#### 5 Things to Know about Microservices

Traditional practices are daily being challenged by innovative ideas. Software Architecture is no different. Until very recently, single big applications ("monoliths") were the de-facto standard way of building applications. The entire team worked in unison to produce a single deployable unit which encompassed all the business requirements. This blog post discusses an alternative architecture pattern, Microservices, which is increasingly being used by companies in their cloud based solutions.

**What are microservices?**

Microservices is an architecture style which prescribes building large complex software applications using many small microservices. These microservices are narrowly focused, independently deployable, loosely coupled, language agnostic services fulfilling a business capability. These multiple microservices communicate with each other using language-agnostic APIs such as REST.

These microservices are applications in themselves and are often owned by small teams. Unlike the normal practice, the team which coded the microservices is also responsible for its support.

**Why microservices?**

Business drivers like agility, better reliability, improved scalability, security has forced architects to consider new paradigms like cloud computing and microservices. The traditional monolithic applications suffer from challenges like

1. Difficult to develop: The more is the code base, the more difficult is it to understand and maintain.
2. Difficult to test: One single change can affect multiple sub units forcing much larger testing effort. The testers should also be aware of the various code inter dependencies.
3. Slower to adapt: Changing even a single aspect of the application requires the entire code base to be affected, thus making it much slower to change and error prone.

In contrast, microservices, by definition itself are smaller manageable units of business capabilities which are easy to develop, test and maintain and also faster to adapt.

**How is it related to SOA?**

Services form the core concept in both SOA and microservices but the objectives are different. While SOA services aim for enterprise wide reuse, Microservices have a much smaller goal. They are made for specific objectives so they are often called “fine grained SOA”.  Microservices tends to be focused on agility.

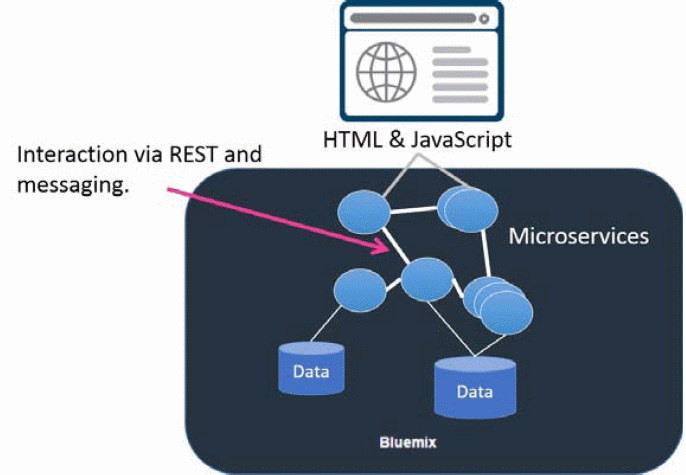
SOA is found in technical environments which have traditional in-house infrastructure, where as Microservices are being rapidly adopted in organizations relying on Cloud computing as cloud infrastructure makes it easy to deploy, manage and collaborate with other microservices.

**Bluemix is a great platform for microservices**

IBM Bluemix is an open-standards cloud platform for building, running, and managing applications. It provides developers access to IBM and third party services for integration,security, transaction, and other key functions. Providing a Platform as a Service (PaaS) environment as one of its runtimes, along with containers and virtual machines, Bluemix leverages the Cloud Foundry open source technology to accelerate new application development and DevOps methodologies. Additionally, Bluemix provides optimized and elastic workloads, enables continuous availability, and simplifies delivery and manageability of an application by providing pre-built services and hosting capabilities.

Microservices are created as individual applications on Bluemix. These multiple microservices  talk using REST or messaging services.. IBM Bluemix also provides rich DevOps capabilities which allows developers to rapidly create microservices. IBM Bluemix DevOps Services provides capabilities like

* Agile planning, through the Track & Plan service
* A Web IDE for editing and managing source control
* Source control management (SCM), through Git, Jazz SCM, or GitHub
* Automated builds and deployments, through the Delivery Pipeline service

[](https://www.ibm.com/developerworks/community/blogs/5things/resource/BLOGS_UPLOADED_IMAGES/Image7.gif)

**Will Microservices solve all our problems and bring "World Peace"?**

Microservices are not a silver bullet that will solve all our problems. Microservices has its own challenges and can’t be blindly applied to all scenarios. Few of the issues that needs to be carefully addressed are

* How to effectively partition the application?
* How to co-ordinate the changes to different microservices?
* Data replication across the different data stores

To summarize, Microservices pattern offer a fresh alternative to Monolithic application development and should be considered in Cloud based solutions. Check out the latest Redbooks publication [Microservices from Theory to Practice: Creating Applications in IBM Bluemix Using the Microservices Approach](https://www.redbooks.ibm.com/redbooks.nsf/RedpieceAbstracts/sg248275.html?Open). for more information on microservices and how to develop applications using the microservices approach in Bluemix.

[](https://www.ibm.com/developerworks/community/blogs/5things/resource/BLOGS_UPLOADED_IMAGES/shirin.gif)

**Shishir Narain** is an Open Group certified Master IT Specialist with deep skills in IBM middleware products. He works in IBM Software Services for WebSphere at India Software Lab, Bangalore. He has 16 years of experience in developing solutions for multiple clients. He has led several end-to-end IT implementations based on SOA. He holds a Master of Technology degree from Indian Institute of Technology, Kanpur.

# [Self-Contained Systems – Microservices for Dummies](https://dejanglozic.com/2016/01/04/self-contained-systems-microservices-for-dummies/)

I love the smell of the new year in the morning. As in the tweet above, there is something hopeful in starting a new blog by first updating the copyright notice at the bottom.

Nevertheless, what is currently occupying my mind is not as much how 2016 started, but how 2015 ended. As luck would have it, within weeks I had a chance to present about our production experience with micro services to two enterprises –  a mutual fund and a bank. Both were curious about them and simultanously unsure as to how they would apply in their unique situation.

In such discussions, the topic of Netflix inevitably comes up, and I am not sure this is a good thing. Sure, I enjoyed watching The Social Network like the most of us, but I am not sure it taught me much about how to become Mark Zuckerberg. Certain experiences, approaches and stories are so intensely singular that their utility as a ‘how to’ manual is quite poor.

In the case of Netflix, I have noticed two characteristics that are not very useful in most scenarios where micro services are considered:

1. Very few outfits need to scale up to a [third of all Internet traffic](http://fortune.com/2015/10/08/netflix-bandwith/)
2. Being the micro service pioneers, Netflix engineers wrote a lot of custom code that is impractical in most situations

**Maybe we don’t need 1000 microservices?**

The consequence of the fact that you don’t need to prepare to handle a third of all Internet is that the overhead equation skews towards having fewer large micro services, rather than many small ones. In fact, it is surprising what transpires when discussions go a bit deeper into the ‘why’ direction.

As you probably know, I am on the record claiming that micro services are more about future-proofing, solving organizational problems of large teams and DevOps than any particular technology. It has become apparent to us as we talk to companies that the problem of [two speed IT](http://www.mckinsey.com/Insights/High_Tech_Telecoms_Internet/Organizing_for_digital_acceleration_Making_a_two_speed_IT_operating_model_work?cid=digital-eml-alt-mip-mck-oth-1510) is more acute and pressing than replicating Netflix’s improbable feat. In the two-speed systems, companies can preserve investment and control through exposing the current system of record through APIs, while building up a cloud-based micro service system that consumes those APIs and serves Web and mobile experiences at the required speed and agility. You don’t need thousands of micro services for this.

In fact, more proof that the pragmatic pendulum has swung towards more reasonable architectures  can be found in the coining of a new phrase – [Self Contained Systems (SCS)](https://www.innoq.com/en/links/self-contained-systems-infodeck/). It is a neat workaround for the cognitive issues that may come from the term ‘micro service’. The term ‘system’ implies something larger, so even the illustrations of SCSs imply no more than a dozen or so. All the good properties of micro services are preserved (polyglot persistence, ability of teams to develop and deploy them independently, stack independence), but there is no expectation to deploy them in the hundreds and thousands. Sure, a self-contained system can in turn consist of several micro services, but at that point it becomes implementation detail.

It is probably best for companies dipping their toes into the micro service world to think about self-contained systems instead, lest they are taken down the Netflix path which will most likely be wrong for them.

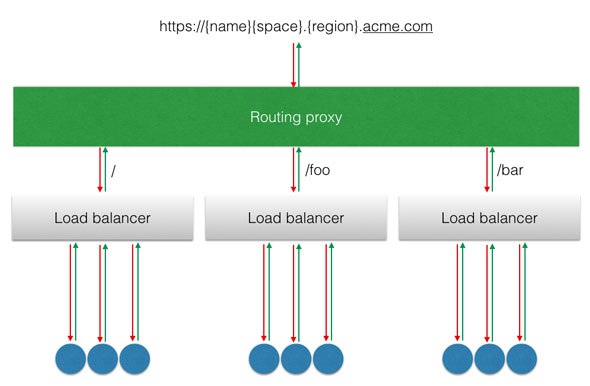
**I make my own underwear**

The point of custom code from above reminded me of a story from the days when I was still climbing the audiophile ladder and cared about hi fi companies and stories about them. In this particular story from 2011, a parent company IAG that owns illustrious British names such as Quad, Mission and Wharfedale, [describes its process](http://www.whathifi.com/news/china-behind-scenes-iag) like so:

One of the immediately notable things about IAG’s operation is the extent of what they make themselves: from new parts, to the tools to make those parts, through to all their own speaker drivers: the vast bulk of what goes into IAG products is made right there in the Shenzhen factory. They even manufacture their own wire.

*What Hi Fi: Behind the scenes at IAG*

The advice ‘start by making your own wire’ would make sense for very few companies, and the same applies to most enterprises trying to replicate Netflix’s approach. This comes into sharp focus when you start discussing concepts such as service discovery. Consider the following diagram:

[](https://dglozic.files.wordpress.com/2016/01/service-discovery.jpg)

The URL of the micro service system is composed based on spaces (dev, QA, prod) and regions. It can be easily constructed or scripted. The routing proxy partitions the path namespace, and each path acts as API and is driving a load balancer which in turn handles any number of micro service instances.

I am showing you this because we are running this successfully in production right now. If you want to, say, hit the development version of the micro service system running in US1 data centre, you would have something like [**https://foo-dev.us1.acme.com**](https://foo-dev.us1.acme.com). A script with a couple of substituted variables is all you need to hit the system you want. As for the load balancer, it comes for free in the IBM’s Cloud Foundry-based PaaS ([Bluemix](http://bluemix.net)).

If you have a Netflix-type system, you need service discovery just so that you can connect all your micro services, all flailing around at the unguessable IPs and port numbers. And since you need to build everything yourself, a simple thing such as load balancing that comes out of the box in all PaaSes and a number of IaaSes is something you need to provide.

Netflix story is unique for many reasons, and having to build everything yourself is one of those characteristics not necessary in 2016. Cloud today is all about what you can outsource to a PaaS or IaaS vs what is uniquely your value add. Building load balancers is doable but a rarely justified practice today.

**For Dummies?**

Yes, I know, forgive me for the click bait – SEO and all that. But by now you are well versed with the tongue-in-cheekness of the ‘For Dummies’ franchise. As people finding value in them are hardly dummies, so are people wisely choosing Self-Contained Systems instead of going down the path of the thousand micro services. Running a half a dozen to a dozen independently deployed, agile self-contained systems hooked up to your legacy systems may be the best choice for you right now. You can always add more, but don’t feel pressured to do so.

© Dejan Glozic, 2016

own an Acura TL 2006. It’s a great car. Every day I derive great pleasure driving it to work. It has a tight sporty suspension, precise steering, comfortable leather seats and an awesome audio system.

At the same time, I know better than to take it off-road. Its high performance tires are optimized for asphalt traction and low rolling resistance, not gravel or soil. It does not have enough clearance for rocks, or 4×4 drive required for rough terrain. If I did take it off-road, I could erroneously conclude that it is an awful car, which I know not to be true. I would have simply used it for something it was never designed to do.

I used this example to explain the concern I have seeing the evolution of the industry’s relationship with the micro-service architecture. It was just a matter of time people until people started taking their micro-service Acuras off-road and then writing how they are awful cars.

**Original success stories**

Architectures and approaches normally turn into trends because enough use cases exist to corroborate their genuine usefulness when solving a particular problem or a class of problems. Otherwise, only [architecture astronauts](http://www.joelonsoftware.com/articles/fog0000000018.html) would care. In the case of micro-services before they were trendy, enough companies built monoliths beyond their manageability. They had a real problem on their hands – a large application that fundamentally clashed with the modern ways of scaling, managing and evolving large systems in the cloud. Through some trial and error, they reinvented their properties as a loose collections of micro-services with independent scalability, life cycle and data concerns. Netfix, Groupon, Paypal, SoundCloud are just a small sample of companies running micro-services in production with success.

It is important to remember this because the trendiness of micro-services threatens to compel developers to try them out in contexts where they are not meant to be used, resulting in the projects overturned in the mud. This is bad news for all of us who derive genuine benefits from such an architecture.

**Things to avoid**

It is therefore good to try to arrive at a useful list of use cases where micro-services are not a good choice. It will keep us more honest, keep the micro-service hype at bay and prevent some failures that would sour people to an otherwise sound technical approach:

1. **Don’t start with micro-services** – this one is a no-brainer. Micro-services attempt to solve problems of scale. When you start, your app is tiny. Even if it is not, it is just you or maybe you and couple more developers. You know it intimately and can rewrite it over a weekend. The app is small enough that you can easily reason about it. There is a reason why we use the word ‘monolith’ – it implies a rock big enough that it can kill you if it falls on you. When you start, your app is more like a pebble. It takes certain amount of time and effort by a growing number of developers to even approach monolith (and therefore micro-service) territory.
2. **Don’t even think about micro-services without DevOps** – micro-services cause an explosion of moving parts. It is insane to attempt it without serious deployment and monitoring automation. You should be able to push a button and get your app deployed. In fact, you should not even do anything – committing code should get your app deployed through the commit hooks that trigger the delivery pipelines (at least in development – you still need some manual checks and balances for deploying into production).
3. **Try not to manage your own infrastructure** – micro-services often introduce multiple databases, message brokers, data caches and similar services that all need to be maintained, clustered and kept in top shape. It really helps if your first attempt at micro-services is free from such concerns. A PaaS such as Cloud Foundry or Heroku will allow you to be functional faster and with less headache than with an IaaS, providing that your micro-services are PaaS-friendly.
4. **Don’t create too many micro-services** – each new micro-service adds overhead. Cumulative overhead may outstrip the benefits of the architecture if you go crazy. It is better to err on the side of larger services and only split when they end up containing parts with conflicting demands for scaling, life cycle and/or data. Making them too small will simply [transfer complexity](http://hueniverse.com/2014/05/30/the-fallacy-of-tiny-modules/) away from the micro-services and into the service integration task.
5. **Don’t share micro-services between systems** – I listed this final point here for completeness, but it is so important that it requires to be broken into its own section.

**On micro-service sharing**

I have seen many a fiery debate about the difference between micro-services and SOA. There are many similarities (it is hard to argue that micro-service architecture, or MSA is revisiting SOA principles). More recently I have formed a fairly strong opinion that a key differentiation between MSA and SOA is that of **ambition**.

When you go back and read about the lofty goals of SOA proponents, it is easy to notice that the aim was much higher. MSA success stories didn’t attempt to reinvent the world around catalogs of reusable services, systems that are discovering those services through registries, etc. At the beginning of every MSA success story is a team that grew their simple application too fast without refactoring along the way and hit the maintainability wall.

If you carefully read ‘monolith to micro-services’ blog posts, you will notice that the end result is the same thing. Groupon team has not created a ‘catalog of social coupon services to be assembled into coupon applications’ – they rebuilt Groupon Web site. They broke the monolith into small pieces and rebuilt it again. As far as their end users are concerned, the monolith is still there – the site was rebuilt in mid-air.

Since I think that micro-services are pragmatic and sane revisiting of SOA, it is apt to assume that creating reusable micro-services is low on the list of priorities. Yes, a micro-service needs to be individually deployable and be flexible enough that it can be bound to other services dynamically (minimally through some kind of a configuration on startup). You need to be able to deploy each service to multiple logical ‘spaces’ (DEV, QA, STAGING, PROD). But each logical micro-service instance is part of a single distributed monolith, re-imagined in a cloud-friendly way.

**From a monolith to a – distributed monolith?**

Where am I going with all this? I am a bit concerned that the industry noise will ruin micro-services by taking them outside their comfort zone. Too many people are taking them to the areas where they shouldn’t, and I don’t want the inevitable backlash to overshoot. Micro-services are a solution for the Big Ball of Mud architecture, but the alternative micro-service system is still a big ball. This ball made up of many small balls, is cleaner and easier to manage, deploy, scale and evolve, and can be inflated bigger than the old ball without exploding, but it is fundamentally the same thing.

Any attempts at nano-services, trying to deploy micro-services manually, using them because they are trendy without real need, or re-using them between multiple systems will result in a disappointment we don’t really need at the moment.

Are micro-services SOA? No, and please let’s keep it that way.

**Scalable Microservices through Messaging**

Posted by [Bilgin Ibryam](http://developers.redhat.com/blog/author/bibryam/) on May 26, 2016

Microservices are everywhere nowadays, and so is the idea of using service choreography (instead of service orchestration) for microservices interactions. In this article I describe how to set up service choreography using ActiveMQ virtual topics, which also enables scalable event based service interactions.

**Service Interaction Styles**

There are two main types of service interaction: **synchronous** and **asynchronous**.

With *synchronous* interactions, the service consumer makes a request and blocks until the operation completes and a response is received. The HTTP protocol is a great example for a synchronous interaction. This type of interaction is usually associated with request/response interaction style and the HTTP protocol. (Of course, it also possible to do request/response with asynchronous requests or event messaging, via registering a callback for the result, but that is a less common use case).

With an *asynchronous* interaction style, the service consumer makes a request, but doesn’t wait for the operation to complete. As soon as the request is acknowledged as received, the service consumer moves on. This type of interaction allows publish/subscribe style of service communication — e.g. instead of a service consumer invoking an operation from another service, a producer raises an event and expects interested consumers to react.

Apart from these technical considerations, there is also another aspect to consider with service interactions: coupling and responsibility.

If service A has to interact with service B, is it the responsibility of service A to invoke service B (***orchestration***) or is it the responsibility of service B to subscribe for the right events (***choreography***)?

With service orchestration, there is a central entity (as the service A itself in our case), which has the knowledge of other services to be called. With the choreography approach, this responsibility is delegated to the individual services and they are responsible for subscribing for the “interesting” events.

To read more about this topic, checkout Chapter 4 from the excellent Building Microservices [book](http://samnewman.io/books/building_microservices). For the rest of this article, we will focus on doing service choreography using messaging.

**Service Orchestration Through Messaging**

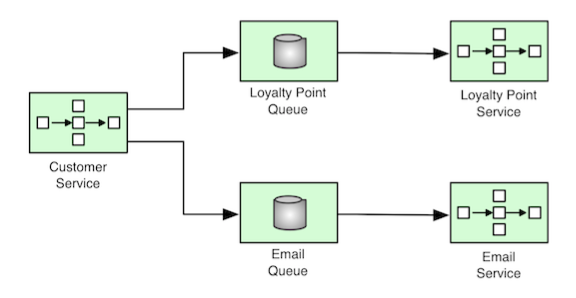
Service orchestration in messaging is achieved through queues. A queue implements load balancer semantics using the competing consumer pattern and makes sure that there is only one consumer of a message.

Let’s say there is a “Customer Service” that has to interact with “Email Service”.

The easiest way to implement this is to use a queue and let “Customer Service” send a message to “Email Queue”. If the “Customer Service” has to interact with “Loyalty Point Service”, again, “Customer Service” has to send another message — this time to “Loyalty Point Queue”.

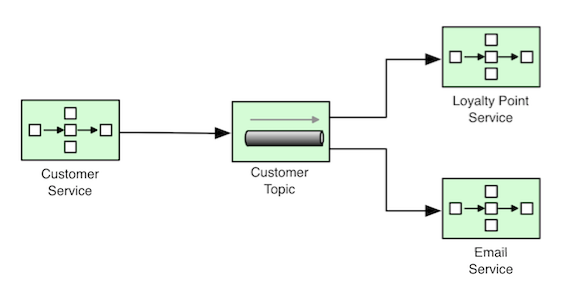
With this approach, it is the responsibility of “Customer Service” to know about “Loyalty Point Service” and “Email Service”, and subsequently send the right messages to the corresponding queues. In short, the whole interaction is orchestrated by “Customer Service”.

One advantage of using queues is that, it allows scaling of consumers easily. We could start multiple instances of “Loyalty Point Service” and “Email Service”, and the queues will do the load balancing among the consumers.

[](https://rhdevelopers.files.wordpress.com/2016/05/33131-pointtopoint.png)

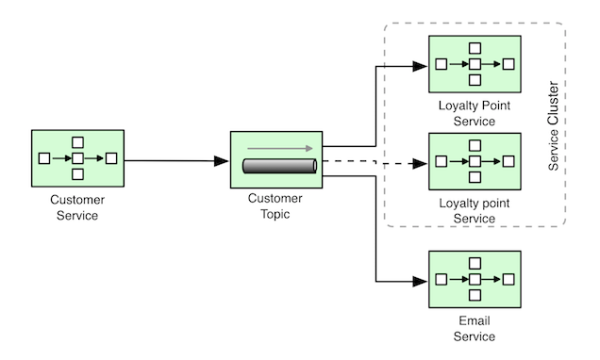
**Service Choreography Through Messaging**

With the service choreography approach, “Customer Service” doesn’t have any knowledge of “Loyalty Point Service” or “Email Service”. “Customer Service” simply emits an event to “Customer Topic”, and it is the responsibility of “Loyalty Point Service” and “Email Service” to know about the Customer event contract and subscribe to the right topic — the publish/subscribe semantics of the topics will ensure that that every event is distribute to both subscribers.

[](https://rhdevelopers.files.wordpress.com/2016/05/60659-publishsubscribe.png)

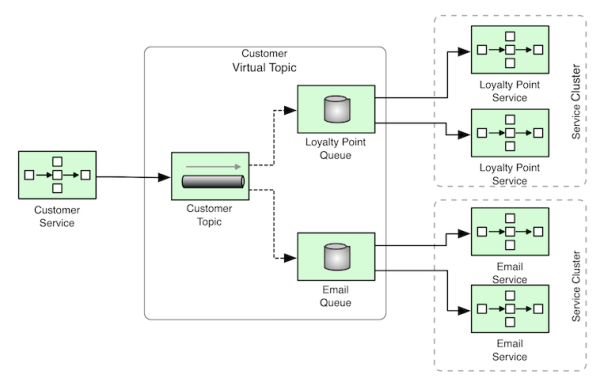
**Scaling Service Choreography**

Since topics implement publish/subscribe semantics rather than competing consumers, scaling the consumers becomes harder. If we (horizontally) scale “Loyalty Point Service” and start two instances, both instances of the service will receive the same event and there won’t be any benefit in scaling (unless the services are idempotent).

[](https://rhdevelopers.files.wordpress.com/2016/05/46b6b-scalingpublishsubscribe.png)

**ActiveMQ Virtual Topics to the Rescue**

So what we need is some kind of mixture between topic and queue semantics. We want the “Customer Service” to publish an event using publish/subscribe semantics so that all services receive the event, but we also want competing consumers, so that individual service instances can load balance events and scale.

[](https://rhdevelopers.files.wordpress.com/2016/05/61b72-scalingwithvirtualtopics.png)

There are a number of ways we could achieve that with Camel and ActiveMQ:

* The very obvious one that comes to (my) mind is to have a simple Camel route that is consuming events from “Customer Topic” and sends them to both “Loyalty Point Queue” and “Email Queue”. This is easy to implement, but every time there is a new service interested from the “Customer Service” events, we have to update the Camel routes. Also, if you run the Camel route on a separate process than the broker, there will be unnecessary networking overhead only to move messages from a topic to a set queues in the same message broker.
* An improvement to the above approach would be, to have Camel routes running in the ActiveMQ broker process using [ActiveMQ Camel plugin](http://activemq.apache.org/broker-camel-component.html). In that case, you still have to update the Camel route every time there is change to the subscribers, but the routing will happen in the broker process itself, so no networking overhead.
* And even a better solution would be to have the queues subscribed to the topic w/o any coding, but using a declarative approach using ActiveMQ [virtual topics](http://activemq.apache.org/virtual-destinations.html) (hence the whole reason for writing this article).

Virtual topics are a declarative way of subscribing queues to a topic, by simply following a naming convention — all you have to do is define or use the default naming convention for your topic and queues.

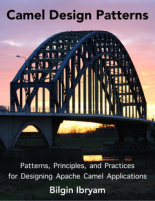
For example, if we create a topic with a name matching *VirtualTopic.>* expression, such as: *VirtualTopic.CustomerTopic*, then have the “Loyalty Point Service” consume from *Consumer.LoyaltyPoint.VirtualTopic.CustomerTopic* queue, the message broker will forward every event from *VirtualTopic.CustomerTopic* topic to *Consumer.LoyaltyPoint.VirtualTopic.CustomerTopic* queue.

Then we could scale Loyalty Point Service by starting multiple service instances, all of which consume from *Consumer.LoyaltyPoint.VirtualTopic.CustomerTopic* queue.

Similarly, later we can create a queue for the Email Service by following the same naming convention:

*Consumer.Email.VirtualTopic.CustomerTopic*. This feature allows us to simply name our topics and queues in a specific way,  and have them subscribed without any coding.

**Final thoughts**

[](https://rhdevelopers.files.wordpress.com/2016/05/476a6-medium.jpg)

This is only one of the many patterns I have described in my recently published [Camel Design Patterns book](https://www.amazon.com/dp/B01D1RERQG). Camel is quite often used with ActiveMQ, and as such, you can find also some ActiveMQ patterns in the book too.

Another way to scale microservices using choreography can be achieved through event sourcing. You can find a nice blog post describing it [here](https://www.thoughtworks.com/insights/blog/scaling-microservices-event-stream).

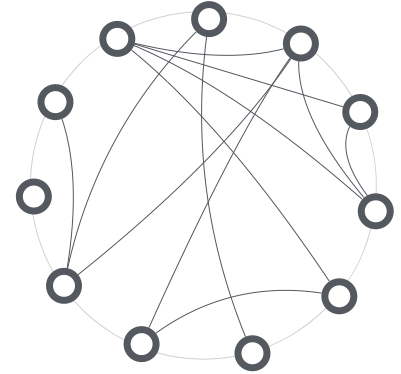
# Scaling Microservices with an Event Stream

In [Building Microservices](http://info.thoughtworks.com/building-microservices-book), Sam Newman explains why services choreography may be a more appropriate integration model for supporting complex business processes across domains.

In this article, I would like to stress the challenges of using a point-to-point integration model, and present services choreography as a foundation of a more expandable microservices architecture, where services are highly cohesive and loosely coupled.

### Services orchestration from the trenches

Last year I participated in a microservices project where code changes became harder and harder because we were making asynchronous point-to-point calls between services. The dependency graph of the eleven microservices we released to production is illustrated in the diagram below. Each node represents a service, and each edge represents a direct HTTP call from one service to another.

  
Dependency graph in a real world orchestrated microservice project

This approach introduced a great deal of complexity when implementing business transactions involving several services, so we struggled to continuously deliver changes in an efficient and safe manner.

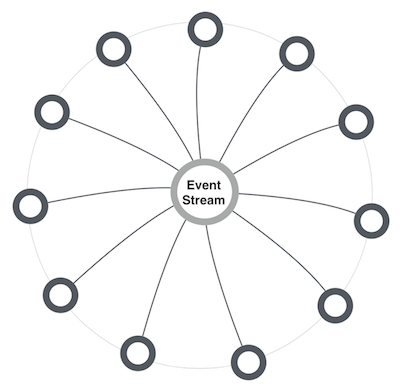
When a business transaction spanned across several services, handling a failure in any of the participating service required careful consideration: is the service that fails essential to the transaction? If it is, then the transaction had to abort and a meaningful error returned to the consumer. On the other hand, when the system could recover from the failure, we let the transaction succeed.

Our testing strategy required depending services to be stubbed, so as to be able to contract-test1 one service in isolation from the others. Each service had a stub, with its own code repository and [Go](http://www.go.cd/) pipelines. By stubbing each service, we dramatically increased the number of services to manage, and had to create dedicated environments where each service only depends on stubs for contract-testing.

For some of our most complex services, we also relied on our browser-based tests of the client application to validate their integration and make sure we had not introduced unforeseen side effects when executing a transaction against real services (and not stubbed). High services coupling prevented us from releasing them independently from one another. We ended up with a “monolithic” release process where all services are deployed simultaneously, alongside the client application.

### Services Choreography with an Event Stream

Some of the issues we faced with point-to-point integration could have been avoided by decoupling services and limiting the boundaries of all business transactions. Using a model where an upstream service publishes events, while downstream services subscribe to those events, does provide the required level of isolation. The diagram below shows a number of microservices integrated through an event stream.

  
Dependency graph illustrating the concept of a fully choreographed set of microservices

To explain the idea, let’s reuse Sam’s example of a workflow in MusicCorp2 where three services have to participate in the customer creation process. In an orchestrated approach, when a new customer registers, the customer service implements the entire customer creation workflow: saving the new customer's details and calling downstream services, so that the customer receives a welcome email and gets her first membership status points.  
   
In an event-based approach, the responsibility of what happens after a new customer gets created shifts towards the downstream services. The JavaScript code snippet below shows how a customer service3 can publish a customer\_created event:

app.post('/', function(req, res) {

customers.push(req.body);

var event = { 'type':'customer\_created', 'data':req.body };

publish(event);

res.sendStatus(200);

});

The customer service publishes the event

The boundaries of the customer creation transaction are confined to the customer service, whose responsibility is simply to add the new customer and let the world know about it. Then, any downstream service can subscribe to the stream and will asynchronously be notified when such an event is published, like the email service below.

http.listen(3001, function() {

eventStore.subscribe('customer\_created',

function(customer) {

sendWelcomeEmail(customer);

});

console.log('email service listening on \*:3001');

});

The email service subscribes to the event

### Scale with errors in mind

Handling endpoint failures, which has to be carefully handcrafted in a point-to-point situation, is now pushed to the messaging system. Therefore, the messaging system itself and the choreography of services around this component need to be built with these failures in mind.

If a service if down for a short period of time, event stores will naturally re-post missed events as soon as the subscriber service comes back up. However, if a service is down for a longer period of time, then the service must reload the history of past events and check which ones have been missed, as shown by the code snippet below.

http.listen(3001, function() {

eventStore.history('customer\_created',

function(customer) {

if(!hasWelcomeEmail(customer)) {

sendWelcomeEmail(customer);

}

});

console.log('email service listening on \*:3001');

});

The email service plays back the history of events

In order to track down errors to process an event, a common pattern is to use the event ID as correlation ID among all services participating in the workflow for that event. Hence, when an event is posted, it is logged by the upstream service with its unique ID. For instance, the customer service will log the customer\_created event with event\_id **2987**. Then, the resulting transaction in a downstream service is logged with a reference to the original event. Hence, the loyalty service will log the add\_status\_point transaction with transaction\_id =**2987**.0987.

If each downstream service logs its transaction in relation to the triggering event, it becomes easy - by aggregating logs of all your microservices - to draw a graph of all downstream transactions for an event, and ensure they all occurred successfully.

### Beyond services integration

While I strongly recommend the use of events  to avoid some pitfalls of point-to-point services integration, it also presents other noticeable benefits.

One benefit is to be able to build cross-domain data aggregates without leaking one service domain model into another. In a database-centric application, dedicated views can be used to tailor datasets for presentation purposes. However, in an event-driven world, it is simply a matter of creating a dedicated service that subscribes to the relevant events to build the aggregate and exposes it to its consumers.

Services collaboration through events leads to a much more scalable architecture, with simpler and better tailored models, that will reduce the complexity of your digital assets, increase scalability, and improve your ability to collect data for analytics purposes.

# Consumer-Driven Contracts

**Consumer-Driven Contracts**

By [Ian Robinson](http://iansrobinson.com/)

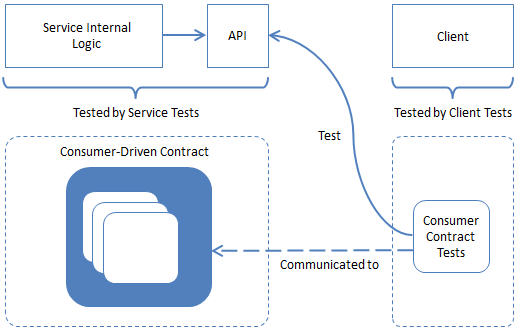
A service has several clients, each with different needs. Service owners know who their clients are, and client developers are able to communicate their expectations of the service's API to service owners.

How can a web service API reflect its clients' needs while enabling evolution and avoiding breaking clients?

Service APIs must often evolve to meet the unique needs of different clients. However, whenever a service changes to accommodate the requirements of a single client, the other client applications face the risk of breaking changes. Some service developers have tried to skirt this problem by creating service APIs that attempt to address all current and future client needs. More often than not, these efforts turn out to be futile as missed requirements are identified, and unanticipated changes are requested.

Service owners who understand how clients use their APIs are at a great advantage. They are able to create client-centric tests that identify when service changes cause clients to break. However, when the service owner creates these tests, they are often an inaccurate interpretation of how the client might use the services. Client owners can help service owners understand how their services will be used by documenting their expectations, but unless these documents are generated from code and schemas, and regenerated every time they change, they can quickly become out of date. What the parties really need are a set of automated integration tests.

Client developers write integration tests that express the client's expectations of a service API. These tests are given to the service owner, who incorporates them into the service's test suite.



The [Consumer-Driven Contract](http://martinfowler.com/articles/consumerDrivenContracts.html) pattern helps service owners create service APIs that reflect client needs; it also helps service owners evolve services without breaking existing clients. Service owners receive integration tests from each client and incorporate these tests into the service's test suite. The set of integration tests received from all existing clients represents the service's aggregate obligations with respect to its client base. The service owner is then free to change and evolve the service just so long as the existing integration tests continue to pass.

**Different types of microservices?**

Posted by [Mark Little](http://developers.redhat.com/blog/author/nmcl2001/) on April 19, 2016

I’ve been working with some of our teams recently on microservices and how we can assist our customers and communities with best practices and recommendations, whether they’re Java EE developers, Vert.x coders, writing [Node.js](http://node.js/) applications or something else. If you’ve [read any of my previous articles](https://developer.jboss.org/blogs/mark.little/2016/04/06/monoliths-and-micromonoliths) then [you’ll know](http://markclittle.blogspot.co.uk/2015/04/microservices-and-unit-of-failure.html) [I have a few thoughts](https://developer.jboss.org/blogs/mark.little/2016/04/04/when-is-a-microservice-not-a-microservice) on [microservices](http://markclittle.blogspot.co.uk/2015/02/microservices.html), and yet there are many things I still feel I need to get straight in my own head. That’s why I love talking with the teams we have, because they’re always challenging my thought processes and pushing the frontiers of where our industry needs to go.

It was during a few of these conversations that something I hadn’t realised had been bothering me started to become a little clearer. For a long time I’ve been thinking about microservices, how they relate to SOA and distributed systems, DevOps etc. As I mentioned at the start, we have a lot of [projects and products](http://developers.redhat.com/products/) which can be used to assist in the development of a (micro) service based architecture. So far, most of what I’ve been reading outside of Red Hat has been about building microservices and applications from collections of them, from scratch. It’s also true to say that has probably been the focus of much of our development work. Greenfield development; re-architecting systems and building up from scratch.

Yet microservices didn’t start out that way. If you go and read some of the original literature, especially from Netflix, the idea of microservices (or “fine grained SOA” as Adrian Cockcroft put it originally) was about taking existing systems and refactoring them into components (services) which could be independently developed, versioned and released. The idea was that you do this work of building microservices in situ (brownfield development). And whilst I’ve known and understood this, I’ve always assumed that the processes, tools, approaches etc. to building microservices in this manner were identical to greenfield. Until recently, that is.

It’s fair to say that you have to start somewhere, and some of our teams have been looking at the brownfield approach to microservices – it makes a lot of sense. It’s pragmatic because most people will be coming at them from the perspective of existing applications, just like Netflix. It doesn’t require you to boil the ocean and you can take things one step at a time. But – and this is what I hadn’t really appreciated until now – it also allows for some significant simplifications to the infrastructure you have to develop in order to *support* these microservices.

In the examples we’ve been looking at, you’ve got existing components in a monolithic application. Since the team were coming at this from a Java EE perspective, these are currently represented as multiple WAR (Web Archive) or JAR (Java Archive) files shipped in a single EAR (Enterprise Archive); however, this is an example that has relevance outside of a single programming language or framework. The aim in this example would be to separate out the individual components into microservices so that we don’t have to recreate the entire EAR each time something changes in a single WAR.

So far so good, and not really any different if you were coming at this from a greenfield perspective. What changes, or at least can be simplified, is what we need to do in the newly distributed case. In this kind of scenario we’re not trying to be a generic distributed system. We don’t necessarily need a name service (or something like it) to locate varieties of services. We don’t necessarily need SLAs and rich contract languages; in fact, because we know the API for the service(s) and we’re very specifically trying to improve the agility of the development processes, we could hard code the API into the “client” (the rest of the application). The realisation to me, which is pretty obvious in hindsight (and one that others in the team already had), is that much of this could simply be hard coded. Binding addresses, or the underlying network, could be leveraged, e.g., use REST/HTTP, with a URL for the service name/address – hence DNS for binding.

Yeah, yeah. I know this only goes so far and at some point you will need some of the complexities we have in traditional distributed systems, but we’re not there yet. The whole point of the exercise from the team was precisely this: “Where do I (a developer) start embracing microservices, and how can I do it easily without having to develop or install too much additional infrastructure to make this a success?” And I think this short cutting / hard coding / relying on the infrastructure works to a degree.

I think the team delivered on the remit. But what got me noodling on this was whether they were developing a new category of microservices? Much of what we read about microservices talks about centralised logging, event driven approaches, orchestration and deployment technologies such as Kubernetes. Yet, none of these is essential if you’re looking at microservices just to increase agility in a very specific application with pre-defined components/services. You could do it all with hard coding and a little bit more automation.

Does it constitute a new category of microservice, as I had thought originally? I don’t think so. It’s an evolutionary approach, no different really to pre-CORBA or pre-Java EE applications which were often written by hand – with hard coded addresses, interfaces, etc. Then as the complexity of distributed applications grew, developers needed more help from the infrastructure and tools. So they’re definitely microservices – as they still tick the right boxes such as making teams more agile, having independent lifecycles etc – they’re just a lot more focussed, perhaps a lot more pragmatic.

**Join** [**Red Hat Developers**](http://developers.redhat.com/products/rhel/overview/)**, a developer program for you to learn, share, and code faster – and get access to Red Hat software for your development.  The developer program and software are both free!**

# Using API keys to secure your microservice

So, you’ve followed all the talk about microservices and created one, complete with its very own REST API to be invoked by various client applications or, indeed, other services. At this point, however, you start to wonder about security – surely you don’t just want anyone calling your service…?

## Using API keys to authorise and validate calls

Like a lot of the microservices landscape, there are a number of techniques and practices that can be applied to a given scenario; security is no exception. This article is going to look at how we can use API keys to authorise and validate calls that a service receives. These keys allow us to answer the following important questions:

1. Which service is making the call?
2. Do I trust the service that is making the call?
3. Have any of the parameters, or the data, been altered?

Because API keys allow you to make sure that data hasn’t been altered, they can also be used to protect your service from other problems such as replay attacks whereby the same request is sent multiple times.

One important thing to note is that API keys will not help you with authentication; that is, the identity of the user and what they are allowed to do. That is typically handled by other security options such as [Oauth](https://en.wikipedia.org/wiki/OAuth) and [OpenID Connect](https://en.wikipedia.org/wiki/OpenID_Connect).

## What makes up an API key then?

An API key is composed of a number of component parts, each of which represents some data that needs to be sent to your service. Here’s an example key:

[Parameter data] + [service ID] + [time stamp]

where:

* Parameter data is the data being sent to your API.
* Service ID is an identifier that your API will use to recognise the service (remember that this won’t say who the service is acting on behalf of).
* Time stamp corresponds to when the request was made.

So, this answers question 1 above, we now know which service is making the API call. However, at the moment, any of these items could just be invented by a malicious client and sent to your service. We still need to answer questions 2 and 3 above: do we trust the caller, and has anything been altered?

## Using a hash to secure the microservice

The way to answer questions 2 and 3 is using something called a [Hash-based Message Authentication Code](https://en.wikipedia.org/wiki/Hash-based_message_authentication_code) or HMAC. An HMAC takes some data, creates a hash of it, and then encrypts that hash. What the hash does is ensure that the data used to generate the hash has not been changed; even changing a single byte of the data will cause a different hash to be generated. (This is why you see websites publish a hash alongside a download link to a mirror site as it allows you to check whether or not a change has been made to the file since the creator published it.)

Once we have the hash, it is then encrypted by what is known as a shared secret which is where the encryption and decryption are performed using something that was previously shared between both parties, such as a password.

Now we know how an HMAC works, how does it help us secure our microservice? What we do is generate an HMAC using the values in our API key and then transmit it along with everything else. The receiving service is then able to use the HMAC to authorise the call and validate the parameters. Going back to our questions, both are now answered with the use of a HMAC as follows:

* Do I trust the service that is making the call? The answer is yes because the shared secret is only known by the sender and receiver, no one else knows it, so it must have come from the trusted service.
* Have any of the parameters, or the data, been altered? No, none have been altered if the hash matches the API key, i.e the data + service ID + time stamp in this example.

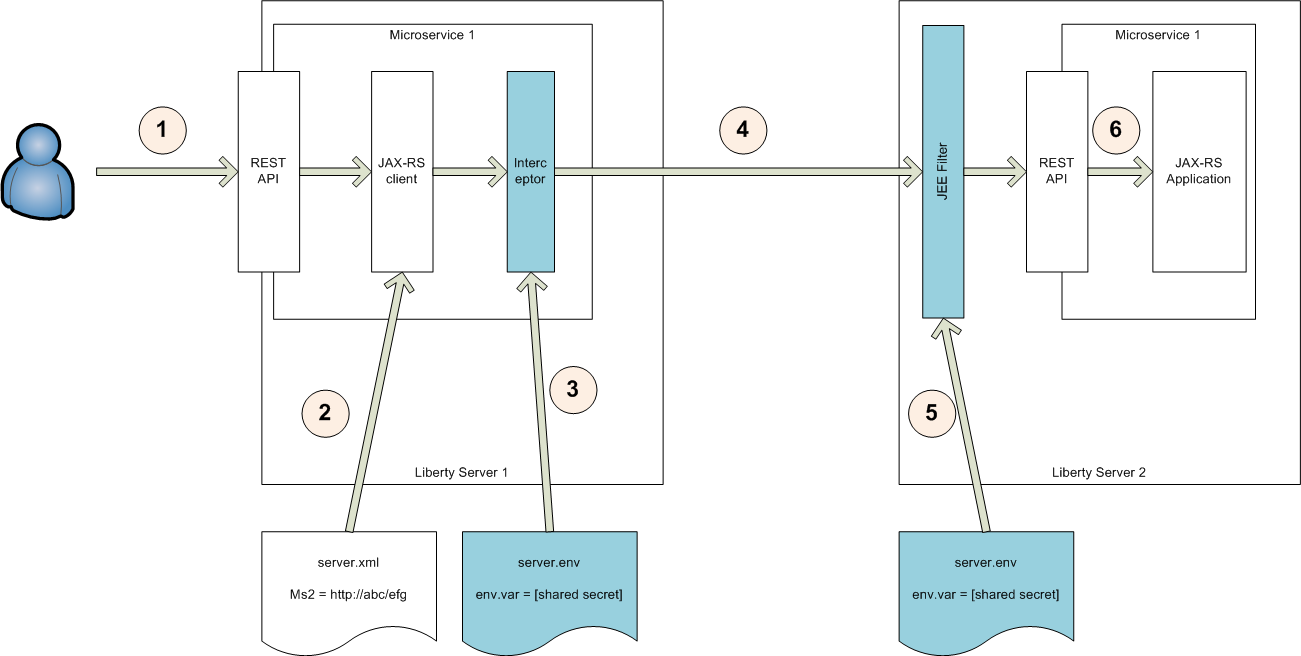
## Things to consider…

* How to share the secret. This is typically done via a second channel. So take GitHub, for example. When you want to authorise an application to access GitHub services, as part of that process you are given a shared secret to use which is specific to your app. Public/private keys obviously solve this problem but require a [public key infrastructure (PKI)](https://en.wikipedia.org/wiki/Public_key_infrastructure) to be setup, which means that the shared secret approach offers a lower configuration and maintenance overhead. Having said that, if both the client and server are part of the same PKI then you can use that.
* How to read the shared secret. Hard-coding the secret into your application needs to be avoided, otherwise it won’t be stateless, easily deployed into cloud environments, easy to change, or will just be plain visible to someone looking at your code!  
  Typically the secret is read from environment variables, which has the advantage of working in pretty much all languages (remember that a microservice architecture can be a ployglot one), but needs to be configured to only be visible from the process running the microservice, rather than all proceses on the server. You can always use more language-specific features to read the shared secret, such a system properties in a JVM, but the important thing is no hard-coding.
* Transmitting the HMAC over the wire. When an HMAC is generated you are typically left with a series of meaningless bytes. You need to think about if they are going to need to be encoded prior to transmission. Take a HTTP REST call, for example. You’re going to need to take some extra steps such as base 64 and URL-encoding before invoking the REST end-point. If, however, you are making a call over something that supports binary formats, such as a messaging system, then it may not be necessary to encode the HMAC at all.

Finally, you can add whatever additional data you want to your API; there isn’t a specification to follow, you just need to consider the three security questions posed at the start of this article: who is calling me, do I trust them, and has anything changed in transit?

## An example API key implementation

In order to illustrate the techniques talked about in this article there is a [WAS Liberty sample](https://github.com/WASdev/sample.securems) available. The diagram below shows the overall architecture of the sample, which consists of two Liberty servers, each hosting their own JAX-RS microservice. The configuration is stored in various Liberty server files such as server.xml. The numbers indicate the process flow starting with a client accessing the REST API of microservice 1 and ending with the invocation inside microservice 2. Finally, shared components have the same colour:

[](https://developer.ibm.com/wasdev/wp-content/uploads/sites/9/2015/09/SecureMS-APIKeys.png)

Let’s now walk through the flow and see what is happening at each stage, together with some code snippets.

### (1) Client makes a request

A client makes a request to the REST API of the first microservice; this may be by the actual user or another microservice. Inside the message-handling code, we create a new JAX-RS client with which we are going to invoke the second (remote) microservice. Ideally we want to create this client every time so that we can be stateless, pick up any configuration changes etc.

|  |  |
| --- | --- |
| 1  2  3  4 | @GET  @Produces(MediaType.TEXT\_PLAIN)  public String getMessage() {      Client client = ClientBuilder.newClient(); |

### (2) Creating the JAX-RS client

The client needs to know where the remote service is located (remember, no hard-coding) and so retrieves the value from the environment variable which was configured in the Liberty server.env file. The other thing that the client does is configure a JAX-RS client interceptor. This is the APIKey class and its constructor takes a service ID and the name of the environment variable storing the shared secret. This class is going to be responsible for creating the API key that will be used during the call. Splitting this out into a separate class allows us to easily add or remove the use of API keys without having to change the business logic.

|  |  |
| --- | --- |
| 1  2  3 | //register the API key generator for use with the client call  APIKey apikey = new APIKey(SERVICE\_ID, SYSPROP\_SECRET);  client.register(apikey); |

### (3) Client generates the API key

The API key class reads the shared secret from the environment variable specified in the constructor (which has its value set in the Liberty server.env file). It then creates the API key by feeding the query string parameters, service ID, and timestamp into an HMAC generator. The service ID, timestamp, and HMAC are then appended to the URL to be invoked by the JAX-RS client.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20 | /\*  \* Entry point for the client that wants to make a request to a second  \* service. It takes the original URI supplied and adds additional query string  \* parameters. These are  \*  \* 1. The service ID supplied by the client  \* 2. A timestamp of when the request was made  \* 3. A generated API key for this invocation.  \*  \* @see javax.ws.rs.client.ClientRequestFilter#filter(javax.ws.rs.client.ClientRequestContext)  \*/  @Override  public void filter(ClientRequestContext ctx) throws IOException {      String idparams = Params.serviceID.toString() + serviceID + Params.stamp.toString() + Long.toString(System.currentTimeMillis());      String apikey = ctx.getUri().getRawQuery() + idparams;      String hmac = URLEncoder.encode(digest(apikey), CHAR\_SET);      URI uri = URI.create(ctx.getUri().toString() + idparams + Params.apikey.toString() + hmac);      System.setProperty(SYSPROP\_LOGGING, &amp;amp;quot;Outgoing request url : &amp;amp;quot; + uri.toString());  ctx.setUri(uri);  } |

### (4) Invoke the API

The JAX-RS client now calls the specified service location using the interceptor modified URL which contains the API key and HMAC.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | //make the request  String log = &amp;amp;quot;Target set in JAXRS client : &amp;amp;quot; + svcurl + &amp;amp;quot;?id=1&amp;amp;amp;full=truen&amp;amp;quot;;  WebTarget target = client.target(svcurl + &amp;amp;quot;?id=1&amp;amp;amp;full=true&amp;amp;quot;);  Invocation.Builder builder = target.request(MediaType.APPLICATION\_JSON);  Response response = builder.build(&amp;amp;quot;GET&amp;amp;quot;).invoke();  String resp = response.readEntity(String.class);  response.close(); |

### (5) Authorising filter

The same class that was used to alter the outgoing client request can also be deployed as a filter. Obviously it’s not always possible to use the same class for both services. If you can, however, it makes life easier in terms of controlling what authorisation steps are involved and ensuring that all parties use the same algorithms.

The following code snippet goes through a number of stages as it authorises the request. It starts by checking that we’ve actually received what we expected in terms of data items, then goes through validating the HMAC, and finally making additional checks using the timestamp. Similar to the client interceptor in step 3, using a filter allows us to secure our service (or change that security) without changing the business logic inside the service.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30 | while(!state.equals(AuthenticationState.PASSED)) {  switch(state) {  case hasQueryString : //check that there is a query string which will contain the service ID and api key      queryString = ((HttpServletRequest) request).getQueryString(); //this is the raw version      state = (queryString == null) ? AuthenticationState.ACCESS\_DENIED : AuthenticationState.hasAPIKeyParam;  break;  case hasAPIKeyParam : //check there is an apikey parameter      pos = queryString.lastIndexOf(Params.apikey.toString());      state = (pos == -1) ? AuthenticationState.ACCESS\_DENIED : AuthenticationState.isAPIKeyValid;  break;  case isAPIKeyValid : //validate API key against all parameters (except the API key itself)      queryString = queryString.substring(0, pos); //remove API key from end of query string      String hmac = request.getParameter(Params.apikey.name());      apikey = digest(queryString);      state = !apikey.equals(hmac) ? AuthenticationState.ACCESS\_DENIED : AuthenticationState.hasKeyExpired;  break;  case hasKeyExpired : //check that key has not timed out      time = Long.parseLong(request.getParameter(Params.stamp.name()));      state = (System.currentTimeMillis() - time) &amp;amp;gt; timeoutMS ? AuthenticationState.ACCESS\_DENIED : AuthenticationState.checkReplay;  break;  case checkReplay : //simple replay check - only allows the one time use of API keys, storing time allows expired keys to be purged      Long value = usedKeys.putIfAbsent(apikey, time);      state = value != null ? AuthenticationState.ACCESS\_DENIED : AuthenticationState.PASSED;  break;  case ACCESS\_DENIED :  default :      ((HttpServletResponse)response).sendError(HttpServletResponse.SC\_FORBIDDEN);  return;  }  } |

### (6) Service process the request

Finally, the service is now invoked (assuming that it has passed the checks in the filter). Depending on your requirements, you can receive the service ID as part of the JAX-RS invocation which then allows you to do additional checking such as validating that the service is entitled to perform the requested operation.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | @GET  public String getMessage(final @QueryParam(&amp;amp;quot;serviceID&amp;amp;quot;) String svcid, final @QueryParam(&amp;amp;quot;id&amp;amp;quot;) String id) {      String msg = &amp;amp;quot;Received params : ntserviceID = &amp;amp;quot; + svcid + &amp;amp;quot;ntid = &amp;amp;quot; + id + &amp;amp;quot;nResponse to client : nt&amp;amp;quot;;  /\*  \* Optionally at this point you would check that the service  \* is entitled to make this call  \*/ |

### Extending the API key

The sample API key shows how you can trust and validate incoming requests from other microservices. This is an extensible model in that you can add as many additional data elements to the API key as fits your organisational needs. Alternatively, you can also add in extra security validation steps.

## Summary

This article has explained the principles behind allowing your microservice to be invoked by other microservices. It has described some of the important security questions that need to be answered and how API keys can help you accomplish this.

However, API keys are only one component of your security toolbox, and generally cannot be used in isolation. Additional steps and techniques need to be employed to establish the identity of the person under which the service is being invoked. Typically, your service is not going to be invoked directly by the user, but by another service acting on behalf of a user. Furthermore, once that has been done you still need to perform validation on the actual input data, such as checking that numbers are really numbers and are within acceptable ranges.

### Share this:

# Refactoring to microservices, Part 1: What to consider when migrating from a monolith

In both the Cloud and Agile programming communities, it seems like everyone is talking about microservices. Other architectural principles like REST have taken the development world by storm, and now microservices are the newest wave to crest. But what are they? And why would a Java™ developer care about them?

In this series, I'll answer those questions and explain why you'd want to migrate your applications to microservices (Part 1). I'll delve into data refactoring (Part 2), and lay out a step-by-step method to help you migrate to microservices (Part 3).

## What are microservices?

0

In [Martin Fowler and James Lewis's classic article](http://martinfowler.com/articles/microservices.html) on the subject, we have the simplest definition of a microservices architecture: "An architectural approach that consists of building systems from small services, each in their own process, communicating over lightweight protocols." A critical concept to understand is that each microservice represents one unique business function.

Fowler also refers to microservices as "service orientation done right." As Fowler and Lewis state, and many others have attested to, the enterprise computing landscape is littered with the remains of large-scale SOA projects gone bad. Microservices may help reverse that trend, but we have to understand where to apply them, and more importantly, recognize that they are especially effective in projects that aren't the neatest, coolest thing on the block.

## Why refactor to microservices?

0

Despite what the folks in Silicon Valley sometimes like to believe, not every application is a green-field. The reality is that enterprises have a lot of existing Java code, and a lot of Java developers. It's simply not economically feasible to throw away all of that Java code and start fresh with all new runtimes and programming languages.

Instead, it's better to find the good parts and reuse those in the right framework. That's why refactoring existing applications into microservices is often the best, most prudent approach to keeping your existing systems running while moving them to a more sustainable development model.

## How do you refactor to microservices?

0

So what do I mean by refactoring? Programming communities define it as "introducing a behavior-preserving code transformation." That boils down to keeping your external APIs the same while changing the way your code operates or is packaged. Refactoring to microservices would thus mean adding microservices into your application without necessarily changing what it does. You wouldn't add new functionality to your application, but you would change how it's packaged and perhaps how the API is expressed.

Refactoring to microservices is not right for every application, and you can't always do it successfully. But refactoring is worth considering when you can't throw everything away. The three basic considerations are:

* How is your application packaged (and built)?
* How does your application code function?
* How does your application interact with back-end data sources when those data sources are structured in different ways?

### Step 1. Repackaging the application

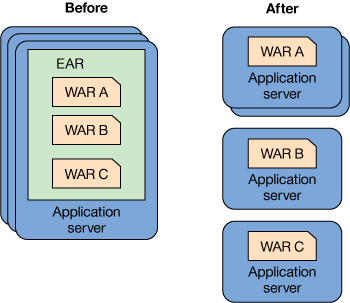
0

The best place to begin is by revisiting your Java application packaging structure and adopting some new packaging practices before you even start to change your code. In the early 2000s we all started building ever-larger EAR files to contain our logical applications. We then deployed those EARs across every WebSphere® Application Server in our farm. The problem is that this tied each piece of code in that application to the same deployment schedules and the same physical servers. Changing anything meant retesting everything, and that made any changes too expensive to consider.

But now with containers like Docker and PaaS, and lightweight Java servers like WebSphere Liberty, the economics have changed. So now you can start reconsidering the packaging. Here are three principles you need to start applying:

1. **Split up the EARs**: Instead of packaging all of your related WARs in one EAR, split them up into independent WARs. This may involve some minor changes to code, or more likely to static content, if you change application context roots to be separate.
2. **Apply "Container per service"**: Next apply the "Container per service" pattern and deploy each WAR in its own Liberty server, preferably in its own container (such as a Docker container or a Bluemix instant runtime). You can then scale containers independently.
3. **Build, deploy, and manage independently**: Once they are split, you can then manage each WAR independently through an automated DevOps pipeline (such as the IBM DevOps Pipeline Service). This is a step toward gaining the advantages of continuous delivery.

You can see the effect of applying these three principles:



### Step 2. Refactoring the code

0

Now that your deployment strategy has gotten down to the level of independent WARs, you can start looking for opportunities to refactor the WARs to even more granular levels. Here are three cases where you can find opportunities for refactoring your code to accommodate a packaging approach that packages microservices independently.

* **Case 1. Existing REST or JMS services**: This is by far the easiest case for refactoring. You may have existing services that are already compatible with a microservices architecture, or that could be made compatible. Start by untangling each REST or simple JMS service from the rest of the WAR, and then deploy each service as its own WAR. At this level, duplication of supporting JAR files is fine; this is still mostly a matter of packaging.
* **Case 2. Existing SOAP or EJB services**: If you have existing services, they were probably built following a functional approach (such as the Service Façade pattern). In this case, functionally based services design can usually be refactored into an asset-based services design. This is because in many cases, the functions in the Service Façade were originally written as CRUD (create, retrieve, update, and delete) operations on a single object. If this is true, the mapping to a RESTful interface is simple: just re-implement the EJB session bean interface or JAX-WS interface as a JAX-WS interface. You may need to convert object representations to JSON in order to do this, but that's usually not very difficult, especially where you were already using JAX-B for serializations.

In cases where it's not a simple set of CRUD operations (for instance, account transfer), then you can apply a number of different approaches for constructing RESTful services (such as building simple functional services like /accounts/transfer) that implement variants of the Command pattern.

* **Case 3. Simple Servlet/JSP interfaces**: Many Java programs are really just simple Servlet/JSP front-ends to database tables. They may not have what is referred to as a "Domain Object" Layer at all, especially if they follow design patterns like the Active Record pattern. In this case, creating a domain layer that you can then represent as a RESTful service is a good first step. Identifying your domain objects by applying Domain Driven Design will help you identify your missing domain services layer. Once you've built that (and packaged each new service in its own WAR), then you can either refactor your existing Servlet/JSP app to use the new service or you can build a whole new interface, perhaps with JavaScript, HTML5, and CSS, or maybe as a native mobile application.

### Step 3. Refactoring the data

0

Once you've built and repackaged the small services defined in the previous three cases, you'll want to turn your attention to what may be your hardest problem in adopting microservices: refactoring the data structures that your applications are built on. We'll examine this challenge more deeply in Part 2 of this series. But there are a few rules you can follow for the simplest cases:

1. **Isolated islands of data**: Begin by looking at the database tables that your code uses. If the tables used are either independent of all other tables or come in a small, isolated "island" of a few tables joined by relationships, then you can just split those out from the rest of your data design. Once you have done that, you can consider the right option for your service.

For instance, do you stay in SQL, but perhaps consider moving from a heavy-weight enterprise database like Oracle to a smaller, self-contained database like MySQL? Or do you consider a NoSQL database to replace your SQL database? The answer to that question depends on the kinds of queries you actually perform on your data. If most of the queries you do are simple queries on "primary" keys, then a key-value database or a Document Database may serve you very well. On the other hand, if you really do have complex joins that vary widely (for example, your queries are unpredictable), then staying with SQL may be your best option.

1. **Batch data updates**: If you do have only a few relationships and you decide to move your data into a NoSQL database anyway, consider whether you just need to do a batch update into your existing database. Often when you consider the relationships between tables, the relationships don't take a time factor into consideration; they may not need to be always up to date. A data dump/load approach that runs every few hours may be fine for many cases.
2. **Table denormalization**: If you have more than a few relationships to other tables, you may be able to refactor (or in DBA terms, "denormalize") your tables. Now, even discussing this can raise the hackles of many database administrators. However, if you take a step back, your team as a whole should think about why data was normalized to begin with. Often, the reason for highly normalized schemas was to reduce duplication, which was to save space, because disk space was expensive. However, that's simply not true anymore. Instead, query time is now the thing you want to optimize, and denormalization is a straightforward way to achieve that.

## Conclusion

0

Now you have a taste of what refactoring to microservices is about, and what factors to consider in choosing your approach. The good news is that refactoring your code is not as hard as you may think, and in many cases, it's actually pretty simple. If you work your way through your code looking for these (relatively) simple cases, you may find that the more complex code sections are actually few and far between.

In Part 2 of this series, we'll dive deeper into how the structure of your data can affect how you choose to introduce microservices into your applications.

**Java Microservice**

This step involves developing the Java Microservice. Keeping the scope of the blog in mind, this microservice can be downloaded and run with the following instructions

|  |  |
| --- | --- |
| Git Clone | git clone <https://github.com/sohishah/hello-microservice.git> |
| Build the project in Maven | mvn clean package |
| Run the Microservice | java -jar target/hello-microservice-1.0-SNAPSHOT.jar server hello-microservice.yaml |
| See the Microservice in action | <http://localhost:9000/java/microservice> |

Implementation details about the Microservice can be studied in the source code by loading the project into your preferred Java IDE such as Eclipse.

**Install Docker**

Before the Microservice can be run inside Docker, the Docker technology must be installed on your local machine. You can follow step-by-step Docker installation procedure at: [Docker Installation](https://docs.docker.com/mac/)

Once Docker is installed correctly, you can test your installation using the following command:

|  |
| --- |
| **docker run hello-world** |

**Create a Microservice Docker Image**

In the Docker ecosystem, there are two main concepts to understand.

* Docker container: A Docker container is a lightweight instance of a Linux based OS running on top of your host Operating System
* Docker image: Docker image represents your Application software + entire environment running inside a container

For the above microservice, the container loads the microservice image, and as part of this image it not only loads the Application Code for the microservice, but also the Java 8 environment it needs to run the microservice.

But, before you can load the microservice into Docker, you need to create a Docker image for that software. The steps to create the image are as follows:

|  |  |
| --- | --- |
| Create a directory next to your microservice project | **mkdir hello-microservice-build** |
| Copy microservice artifacts to the build directory | **cp target/hello-microservice-1.0-SNAPSHOT.jar ../hello-microservice-build**    **cp hello-microservice.yaml ../hello-microservice-build** |
| Create the **Dockerfile** file under hello-microservice-build directory | FROM java:8  ADD hello-microservice-1.0-SNAPSHOT.jar hello-microservice-1.0-SNAPSHOT.jar  ADD hello-microservice.yaml hello-microservice.yaml  CMD java -jar hello-microservice-1.0-SNAPSHOT.jar server hello-microservice.yaml  EXPOSE 9000  EXPOSE 9001 |
| From the Docker session, goto the hello-microservice-build directory and issue the command | **docker build -t hello-microservice-local .** |

**Explanation**

The Docker build process uses a file named **Dockerfile** to get its instructions about what to do when building an image. In this particular microservice, the Dockerfile instructs the Docker system to download an image called 'java:8'. This is the core infrastructure needed to run the microservice. Next it adds the microservice jar and configuration to the image. And later, it exposes the ports 9000 and 9001 to service the requests.

**docker build -t hello-microservice-local .** (is the command that processes the Dockerfile and produces the hello-microservice-local image)

**Note:** make sure this command is issued from the Docker session and not just any command line session.

Once this Java Microservice Docker image is created, it must be run inside a Docker container using the following command:

|  |
| --- |
| docker run -p 9000:9000 --name hello-microservice-local -t hello-microservice-local |
| -p:  exposes port 9000 to the host machine port |

You can test the Microservice in the browser using: <http://localhost:9000/java/microservice>

Once this works, you can stop the microservice using: **docker stop hello-microservice-local**

**Publish the Microservice Docker Image**

Now that you have the Microservice Docker Image working locally, you can publish this image to DockerHub to share with your team. This can be accomplished as follows:

|  |  |
| --- | --- |
| Create a DockerHub Account | <https://docs.docker.com/mac/step_five/> |
| Create your Remote Image Repository | yourdockerhubusername/hello-microservice-remote |

Steps to post your local image to the remote repository

|  |  |
| --- | --- |
| Get the image id of the local hello-microservice-local image | docker images |
| Tag the local image for push to remote repository | docker tag **localimageid** yourdockerhubusername/hello-microservice-remote:latest |
| Push to the remote repository | docker push yourdockerhubusername/hello-microservice-remote |

Before testing the remote image, you need to delete the local images. Get the image id for both 'hello-microservice-local' and 'hello-microservice-remote' using: **docker images**

and remove the two images using the command: **docker rmi -f imageid**

Once, the images are removed, you can test the remote image using the following command:

|  |
| --- |
| **docker run -p 9000:9000 --name hello-microservice-remote -t yourdockerhubusername/hello-microservice-remote** |

#### Microservices Architecture, Containers and Docker.

[Erhan-Ekici@IBM](javascript:;) | Dec 8 2014 | Comments (2) | Visits (13445)

[[0 people like this](javascript:;)](javascript:;" \o "0 people like this)

[inShare](javascript:void(0);)

In last few years, some of the large scale online companies including -but not limited to- Netflix, eBay and Amazon changed their application architectures to microservices architecture. It was not a revolution but evolution. They have evolved from a monolithic architecture to microservices architecture.

Everybody is talking about containers, in particular, Docker. Why container technology and Docker have been catching huge attention since last year? Is microservices architecture one of the reasons behind that huge interest? To me, yes. Microservices architecture needs light-weight mechanisms, small independently deployable services, scalability and portability. Those requirements can be met by using containers.

Before talking about containers and Docker, let see first what microservices architecture is?

“Microservices architectural style is an approach to developing a single application as a suite of small services, each running in its own process and communicating with lightweight mechanisms, often an HTTP resource API. These services are built around business capabilities and independently deployable by fully automated deployment machinery. There is a bare mininum of centralized management of these services, which may be written in different programming languages and use different data storage technologies.” (Martin Fowler and James Levis, “[Microservices](http://martinfowler.com/articles/microservices.html)”, March 2014.)

We can look at some characteristics and requirements of microservices and then we can see if containers (Docker in particular) can meet those requirements:

* each microservice is relatively small
* each service can be deployed independently of other services
* easier to scale development.
* improved fault isolation.
* each service can be developed and deployed independently
* eliminates any long-term commitment to a technology stack

**Containers** are usually described as light-weight runtime environments with many of the core components of a virtual machine and isolated services of an operating system designed to make packaging easy and execute these micro-services smoothly.  Containers are not new technology. They have been around Linux world for long time. (Windows Server Containers will be available soon. More on this: [here](http://azure.microsoft.com/blog/2014/10/15/new-windows-server-containers-and-azure-support-for-docker/))

**Docker** is an open-source project which aims to automate the deployment of applications inside portable containers that are independent of hardware, host operating system, and language. In contrast with Virtual Machines, Docker containers do not include a guest operating system but share the operating system with other containers. Docker uses resource isolation features of the Linux kernel such as cgroups and kernel namespaces (\*) to allow independent "containers" to run within a single Linux instance, avoiding the overhead of starting virtual machines.

What docker provides more than other linux containers do is to package an application and all of its dependencies in a virtual container that can run on any Linux server which docker runs.

**Why you should consider Docker as a solution? 5 reasons :  
  
Best suits for microservices architecture:**  Containers support micro services architecture. These services are built around business capabilities and independently deployable by fully automated deployment machinery. Each micro service can be deployed without interrupting the other micro services and containers provide an ideal environment for service deployment in meaning of speed, isolation management, and lifecycle. It is easy to deploy new versions of services inside containers.

**Application Portability:** Docker puts application and all of its dependencies into a container which is portable among different platforms, Linux distributions and clouds. You can build, move and run distributed applications with containers. By automating deployment inside containers, developers and system administrators can run the same application on laptops, virtual machines, bare-metal servers, and the cloud.

**Resource Utilization:** Containers comprise just the application and its dependencies, neither more nor less. Each container runs as an isolated process in userspace on the host operating system, sharing the kernel with other containers. Thus, it enjoys the resource isolation and allocation benefits of virtual machines but is much more portable and efficient. This does not mean that containers can run not only on VMs, but also on physical servers. Due to the lightweight nature of containers, you can run more containers on  a physical server than virtual machines. The result is higher resource utilization.

**Beyond the virtualization is “containers”:** Whenever you need tight resource control and isolation for your application environment, you use virtual machines. But, what if your application environment does not require the hardware resources of full virtualization? Containers can provide user environments whose resource requirements can be strictly controlled with or without using virtualization.

**Enterprise Partnerships:** Container technology is the new emerging technology in the IT industry after server virtualization revolution. Docker is leading this trend with new partnerships. Industry leading companies, including [IBM](http://www-03.ibm.com/press/us/en/pressrelease/45597.wss), Google, Vmware, Redhat, and Microsoft signed partnership agreements with Docker. Those agreements show that Docker has huge potential in the era of cloud and virtualization.

 -------------------------------

\* **Containers** are derived from the idea of OS resource isolation in all levels. That could be possible with a Linux kernel which supports all of the resource-isolation use cases, without the overhead and complexity of running multiple kernel instances at the same time. Linux kernel provides resource isolations with implementation of different types of namespaces. Some of those namespaces are network namespaces, process namespaces, user namespaces and mount namespaces. The purpose of each namespace is to wrap a particular host operating system resource in an abstraction that makes it appear to the processes within the namespace that they have their own isolated instance of the host operating system resource. (More info on this can be found [here](http://lwn.net/Articles/531114/) - Michael Kerrisk’s great article)

**REST and microservices – breaking down the monolith step by asynchronous step**

Posted by [Mark Little](http://developers.redhat.com/blog/author/nmcl2001/) on April 27, 2016

A few days ago I [had a rant](http://markclittle.blogspot.co.uk/2016/04/you-keep-using-that-word-rest-and-i.html) about the misuse and misunderstanding of REST (typically HTTP) for microservices.

To summarize, a few people/groups have been suggesting that you cannot do asynchronous interactions with HTTP, and that as a result of using HTTP you cannot break down a monolithic application into more agile microservices. The fact that most people refer to REST when they really mean HTTP is also a source of personal frustration, because by this stage experienced people in our industry really should know the difference. If you’re unsure of the difference then check out the restcookbook or even [Roy’s PhD thesis](http://www.ics.uci.edu/%7Efielding/pubs/dissertation/rest_arch_style.htm) (it’s quite a good read!)

However, I digress, so back to the rant: My goal is to point people in the right direction and make some recommendations, hence this followup post.

**REST and HTTP**

To start with, it’s definitely wrong to assume that if you are building microservices you must stick with HTTP (although as has been shown in the last decade, a RESTful approach can be beneficial when developing with a service-oriented architecture). [Take a look](http://www.infoq.com/articles/rest-introduction) at [some of these](http://www.infoq.com/articles/roa-rest-of-rest) [older InfoQ articles](http://www.infoq.com/articles/RESTSOAFuture) [for inspiration](http://www.infoq.com/news/2008/10/mark-little-soa-rest), or at what we’ve been doing with WildFly/EAP and other projects/products over the past 7 years or so.

HTTP is not the only option – it has its drawbacks – not least of which is its text nature, and as we’ve found over on the [Narayana](http://narayana.io) project it’s still not really comparable performance with more mature approaches such as IIOP(!) This is despite binary HTTP/2.

It’s not just the performance nature of HTTP that may persuade you to look elsewhere. Traditional messaging products such as [A-MQ](https://www.redhat.com/en/technologies/jboss-middleware/amq), which support a range of patterns including brokered and broker-less messaging, also support messaging standards like [AMQP](https://www.amqp.org) or [MQTT](http://mqtt.org) – making interoperability with heterogeneous systems possible.

I’m not suggesting microservices shouldn’t be developed with HTTP; however, when you’re developing with distributed systems you need to consider all aspects including but not limited to: reliability, performance, and coupling (as [I mentioned in another article](https://developer.jboss.org/blogs/mark.little/2016/04/04/microservices-and-distributed-systems)).

Don’t feel that you have to use HTTP, but likewise don’t feel you must stick with JMS (or even REST) if that’s what you’ve been using in the past. I know that using HTTP it’s relatively easy to test your service (spin up a browser), but check out [Arquillian](http://arquillian.org), for example, if you want to see ways of testing other approaches, including HTTP, without REST.

**Asynchronous invocations**

OK, so what about asynchronous HTTP? Is it impossible as some have stated? Of course it’s possible, and here’s where I give you some references to check out.

First let’s start with some of our well known projects that can be used to develop with the asynchronous message exchange pattern using HTTP: [Vert.x](http://vertx.io/docs/apidocs/io/vertx/core/http/HttpClient.html) and [Undertow](http://undertow.io). Both are exceptionally popular projects with a [range of customers](http://vertx.io) having developed large scale applications with them.

Second? Well you don’t need to take my word for this; again [go and check out](http://www.infoq.com/news/2009/07/AsynchronousRest) a [huge number of InfoQ](http://www.infoq.com/news/2012/02/websockets-rest) articles on that [exact topic](http://www.infoq.com/articles/designing-restful-http-apps-roth), some of which date back [almost a decade](http://www.infoq.com/articles/REST-INTEROP). Now if I had to recommend some books to read on the topic I’d definitely go with one from my old friend and ex-colleague [Jim Webber](https://www.amazon.co.uk/REST-Practice-Hypermedia-Systems-Architecture/dp/0596805829?ie=UTF8&*Version*=1&*entries*=0) or one from my old friend and current colleague [Bill Burke](https://www.amazon.co.uk/RESTful-Java-JAX-RS-Bill-Burke/dp/144936134X/ref=sr_1_3?s=books&ie=UTF8&qid=1461422621&sr=1-3&keywords=Bill+Burke) (more on JAX-RS in a moment).

Whenever you’re working with HTTP it’s always important to understand the [response codes](http://www.w3.org/Protocols/rfc2616/rfc2616-sec10.html) that are available. We’re all pretty familiar with 200, 403, and 404 response codes (and maybe some of the 300’s – e.g., 301 when the service moves), but HTTP has a thing or two to help with asynchronous interactions, too. Specifically 202, where the standard says:

The request has been accepted for processing, but **the processing has not been completed**. **The request might or might not eventually be acted upon**, as it might be disallowed when processing actually takes place. There is no facility for re-sending a status code from an asynchronous operation such as this. The 202 response is intentionally non-committal. **Its purpose is to allow a server to accept a request for some other process (perhaps a batch-oriented process that is only run once per day) without requiring that the user agent’s connection to the server persist until the process is completed**. The entity returned with this response SHOULD include an indication of the request’s current status and either a pointer to a status monitor or some estimate of when the user can expect the request to be fulfilled.

I added highlighting to draw the eye to the bits which clearly mean “asynchronous processing”. Sure, it’s not one of those response codes you see much and if it is used then there’s a good chance you’re not seeing it as it is probably masked by the browser. The point is, however, that HTTP supports asynchronous invocations, so as a developer you can most certainly make use of them.

Of course if you’re looking to do asynchronous HTTP with a standards-based framework then you’re probably thinking of using JAX-RS. There are a plethora of resources on the Web and Bill Burke’s book that I mentioned earlier is another good one. He’s even written about [JAX-RS 2.0 elsewhere](https://dzone.com/articles/whats-new-jax-rs-20), which is worth a look. The standards group, of which Bill was a member, explicitly added an asynchronous client API with callbacks in the most recent version of the specification, and let’s not forget his presentation on [earlier versions in 2009](http://javasymposium.techtarget.com/html/images/BBurke_Scaling_JAX-RS.pdf).

**Is it really asynchronous?**

OK, so if you’ve read this far I hope it’s clear that it is entirely possible to do asynchronous processing with HTTP; however, there’s something else I wanted to try to point out as a flaw in some of the postings from other groups on the topic – things I did hint at in the original rant.

When people have been talking about asynchronous interactions they tend to fall into one of two categories: either they mean that the service request is delivered synchronously to the service which returns an acknowledgement or “ack” to the caller to indicate the work will be done and later the result is sent back to the caller, which has made making forward progress concurrently (think Promises and Callbacks), or the request is sent in a “fire and forget” manner, such that there is no indication of successful delivery to the caller.

Fortunately (or unfortunately depending on your perspective) most people who talk about the latter approach are really thinking about the former, they just forget/ignore the delivery “ack”. The distinction is important to understand and here’s why: in a truly asynchronous system (the second category) it’s impossible to rely on the concept of time to determine whether an endpoint has failed, or if it is just slow. This has a significant impact on deterministic consensus.

**Managing the costs**

In [their 1985 paper](https://groups.csail.mit.edu/tds/papers/Lynch/jacm85.pdf) – which later won the Dijkstra award given to the most influential papers – Fischer, Lynch and Patterson proved the theory (often referred to as the “FLP Theorem”) that it is impossible to rely on the concept of time to determine whether an endpoint has failed. Consensus (agreeing on a value between participants) is possible in synchronous systems but it’s impossible to do this in an asynchronous system with even just a single faulty processor.

You might ask why this is important to you? There’s the obvious aspect that if you move to a truly asynchronous invocation mechanism, then you need to understand what is and is not possible as a direct result. This isn’t theoretical either, as the FLP paper proved. So be aware and develop accordingly. There’s a good reason all [ACID transaction protocols](http://jbossts.blogspot.co.uk/2013/05/2pc-or-3pc.html), such as those in Narayana, are synchronous.

The other thing to note is that some people assume that [Brewer’s CAP theorem](http://www.infoq.com/articles/cap-twelve-years-later-how-the-rules-have-changed) – which discusses trade-offs that need to be made between Consistency, Availability and Partition tolerance when developing a distributed system – is the same as FLP; Some even completely confuse the two theorems.

Although CAP and FLP are related in so much as they are both about behaviors in asynchronous distributed systems, there are some important differences:

For example, [CAP says](https://pdfs.semanticscholar.org/0b0a/af71707a8247b35822f91a95319f1c97476c.pdf) that it is not possible to build an implementation of read-write storage in an asynchronous network that satisfies all of the following three properties:

* Availability – each request eventually receives a response.
* Consistency – each server returns the right response to each request (they are atomic or linearizably consistent).
* Partition tolerance – the network is allowed to drop messages.

As I mentioned, in some ways [CAP](http://markclittle.blogspot.co.uk/2011/06/heisenberg-and-cap-theorem.html) sounds like FLP, but it’s not, and if you’re really interested, [Ken Birman’s paper](http://sigops.org/sosp/sosp15/history/05-birman.pdf) (or some others) can explain more details.

To summarize, FLP permits one failed node to be totally partitioned from the network and does not have to respond to requests, it does not allow messages to be lost (the network is asynchronous, not lossy) and consensus is a different problem to atomic storage.

Now maybe the above is a little more information than you need as a developer, but I think it’s always better to know about all of the possible pitfalls that are waiting for you in distributed systems. Plus, it really is important to understand when people throw around CAP or FLP that sometimes they’re not truly understanding the basis behind them. Unfortunately, [I have to agree with Ken Birman](http://www.devconf.cz/files/slides2015/friday/TransactionsReturningToBigData.pdf), that CAP is often overused and misunderstood.

**Join** [**Red Hat Developers**](http://developers.redhat.com/products/rhel/overview/)**, a developer program for you to learn, share, and code faster – and get access to Red Hat software for your development.  The developer program and software are both free!**

# Tear Down Data Silos with Mobile Microservices

Posted by [cianclarke](http://developers.redhat.com/blog/author/cianclarke/) on November 3, 2015

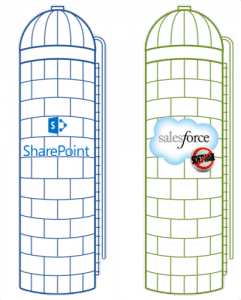
[](https://redhatmobile.files.wordpress.com/2015/11/silos.png)A huge problem facing modern enterprises is managing the large software systems and applications they deal with on a daily basis.

Be it the CRM system purchased by a predecessor, a bundled HRM product thrown in to sweeten a deal, or the CMS that marketing could not live without, silos of information exist in the modern enterprise, and it can often be difficult to utilize the data that these systems contain. When an enterprise decides to buy a proprietary system, often little consideration is given to future interoperability of that product. Many of these decisions were made before the mobile era of computing began, and these products share many common characteristics which are ill-suited to the changing face of computing.

### Designed for the Desktop Era

These applications are typically delivered through a desktop browser. The limited screen real estate of modern smartphones and tablets makes interacting cumbersome. Mobile experiences are often retro-fitted to these products as an afterthought. Sometimes vendors release a mobile application to the app store, but these can be limited in functionality.

### Designed to Silo Data

The SaaS services we’ll be pulling from today

I’m not going to present as fact the concept that many software vendors design products which are a closed black box of data to keep you coming back to buy more – but I can certainly suggest this.  
It’s often not in the interest of the vendor to make a product interoperable with other systems, because that same interoperability can make it very easy to jump ship. This means any API for connecting to the system is at best an afterthought, and often doesn’t exist at all.  
This leaves us with a silo of information which has grown over time, and a system of record which severely limits our future expansion.

### Designed for the Desk, not the Pocket

Interaction with software was vastly different pre-mobile. Users were behind a desk, or perhaps sat with a laptop computer. The tasks they sought to accomplish were longer-running,  e.g. sending an email or filing an expense report. People spent time interacting with software, over a more traditional request-response API interaction.

In the mobile era, users perform tasks standing in line at the coffee shop, or walking through the airport terminal. These tasks are geared towards shorter interactions e.g. adding a comment to a case or approving a workflow.

### Designed for Some-time, not Real-time

We’re interacting with APIs in short, bursty sessions – so we need to design our API for this new world of mobile. We’ve established in a previous blog post[1] the importance of making cohesive, mobile-specific APIs which trim payloads, and combine multiple API responses, but sometimes this isn’t enough.  
We may need to consider building an API which can respond in real-time to changes in data over an always-open connection like websockets, or be able to use push APIs to update a mobile user with some new information, even when the device is asleep in their pocket.

This is a far cry from the functionality which these monolithic silo applications provide.

### Designed for Dumb Sensors, not Smart Ones

Modern smart-devices have a plethora of sensors which, when used collectively, make much smarter systems of engagement. Traditional software systems were not built with such extensive inputs in mind, often only storing simple text and numbers, with no awareness of concepts such as location.

### Designed to be Expensive

Chances are this same vendor of a piece of software which costs a large sum of money in the late 2000’s wants to charge you more to add mobile functionality. There’s an alternative! Instead of paying for the vendor-supplied solution for mobile, which will likely be outdated in two years, why not take ownership and create your own?

## Setting Information Free

We’ve established that most enterprise systems haven’t been built with mobile in mind, and using vendor-provided solutions can often introduce greater cost. However, using an open source software stack, combined with a microservices architecture, it’s very possible to free data from these proprietary systems, moving from a closed, desktop-centric system of record to a highly expansible, mobile-enabled system of engagement.

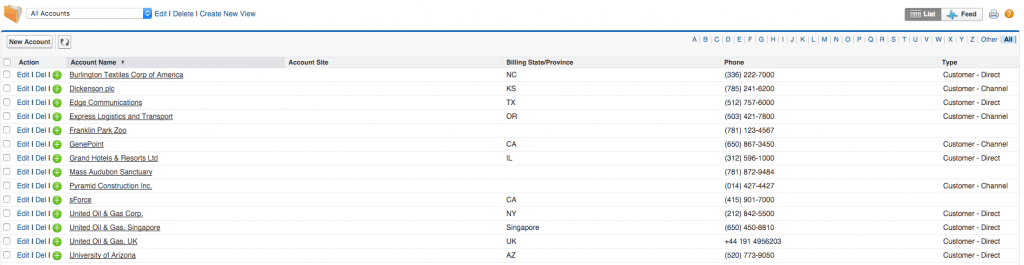
### The Traveling Salesperson

Remember our travelling umbrella salesperson of [Microservices for Mobile](http://www.feedhenry.com/microservices-for-mobile/) fame? In this previous post, we found using a microservices architecture made for much faster response times on lossy networks.  
They’ve since discovered umbrellas aren’t really a hot market right now, and have decided to move into selling birdhouses. They are as of yet unaware of the migratory avian patterns which make October a bad time of year for business, and, yes, I’m well aware these examples keep getting more and more contrived – but that’s part of the fun!

Today, we’re going to build a Mobile-Specific Microservice, which will accept information from the field about birdhouse orders, and interface with Salesforce and Sharepoint connectors in a flexible manner.  
We’re also going to make use of the sensory capabilities of the device which we wouldn’t get with paper to build a mobile app which collects the location where the birdhouses are to be stored.  
Lastly, we’re going to use some open source Node.js modules to pull customer data from Salesforce and product information from a Sharepoint List, and use these to populate the mobile app with data.

### Standing Up Connectors

I’m going to be using a trial account of both Salesforce and Sharepoint 365 for today’s integration. Salesforce will contain the Accounts with which we’re going to associate our orders.  
SharePoint will have a list, each item thereof representing a product’s information.  
Before standing up these microservices, I need to gather some information about my instances – usernames, passwords and the service URLs associated. I’ve used the Red Hat Mobile [Sharepoint](https://github.com/feedhenry-templates/fh-connector-sharepoint-cloud) and [Salesforce](https://github.com/feedhenry-templates/fh-connector-salesforce-cloud) connector. These expose discoverable interfaces to these services within our platform, but can be used standalone too.

[](http://www.feedhenry.com/wp-content/uploads/2015/11/Salesforce.png)  
Our product data in SalesForce

You could also write your own connectors – we’ve used the [Sharepointer](https://github.com/cianclarke/sharepointer) and [Node Salesforce](https://github.com/stomita/node-salesforce) modules in our connectors. Whatever approach you take, know that these connector microservices should be reusable by any future projects which may need to pull data from Sharepoint or Salesforce.

### Taking Ownership of our Data

Instead of storing our installation information in one of the aforementioned systems, we’re going to take ownership of our IT infrastructure the microservices way.

We’re going to build a small mobile microservice which stores information in an orders database. If we ever need to move this data elsewhere, we’ve already got ownership over the schema, and control over the location of the system of record. We can swap this microservice for a more resilient storage solution in future, should the need arise.

We’re just going to expose a Create (i.e. POST /orders) and List (i.e. GET /orders) interface to this service. We’ve already [set up a db connection](https://github.com/birdhousesinc/orders/blob/master/lib/orders.js#L8-L17).

**Code Snippet:**

const COLLECTION = 'orders';

orders.get('/', function(req, res) {

var collection = db.collection(COLLECTION);

collection.find().toArray(function(err, results){

if (err) {

return res.status(500).json(err);

}

return res.json(results);

});

});

orders.post('/', function(req, res) {

var collection = db.collection(COLLECTION);

collection.insert([req.body], function(err) {

if (err) {

return res.status(500).json(err);

}

return res.json(req.body);

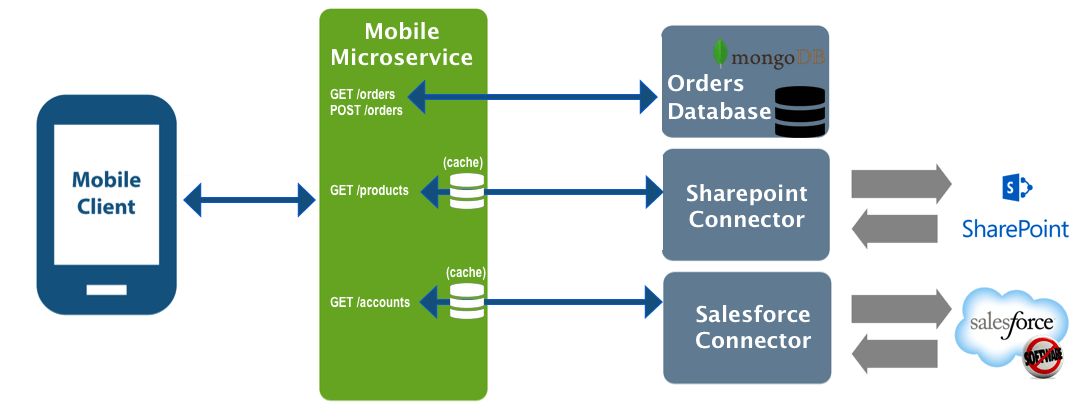
});

});

You can also see the full Orders Database Microservice [on GitHub](https://github.com/birdhousesinc/orders).

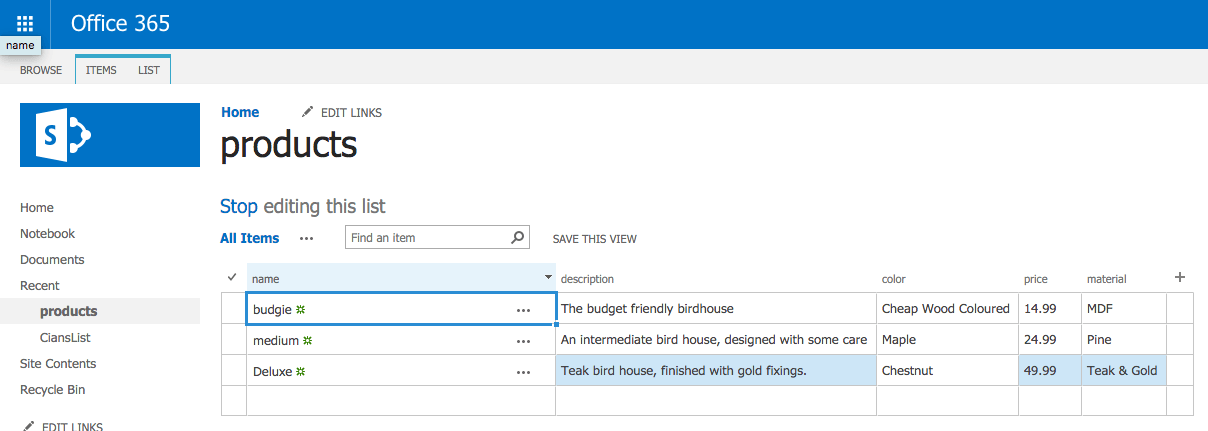
## Enter the Mobile Microservice

Now that we’ve created a series of microservices to help us with this project, there’s one step left. We’re going to create a microservice responsible for interfacing with the mobile app, a gateway of sorts.  
Within this microservice, we’ll massage data into a format more suitable for mobile, and even mash up data from multiple sources to reduce the number of requests from mobile – since we learned in a [previous blog post](https://developers.redhat.com/blog/2015/04/15/evolving-a-mobile-centric-architecture-the-microservices-way/) this can have a drastic impact on performance.

[](http://www.feedhenry.com/wp-content/uploads/2015/11/mashup1.png)  
The Microservices We’ll Be Using

### Overcoming Sharepoint API Oddities

Sharepoint has a number of unique intricacies which make it a challenging platform to integrate with – [I’ve written about these at length.](http://cianclarke.com/blog/on-sharepoint-horrible-integrations/)It came as some surprise that there could be another surprise the SharePoint API could throw at me.

[](http://www.feedhenry.com/wp-content/uploads/2015/11/sharepoint.png)  
Our data in SharePoint

Turns out, when I create a custom list, like I did to store my products, the field names I set up in my headings are output in the API response [keyed with 4 random letters and numbers](https://gist.github.com/cianclarke/206ff07c5e1f52c0a662#file-products-json-L17).  
I then have to look these four character IDs up, iterating over the entire fields array, to find out [what it relates to.](https://gist.github.com/cianclarke/206ff07c5e1f52c0a662#file-products-json-L8)This would be an incredibly fragile binding to have baked into our mobile app, so we’ll do this in our mobile microservice.

### Adding Caching

When integrating with Salesforce, we encounter a different problem. The time it took to login then retrieve accounts varied wildly, from instant responses to timing out after 60 seconds.  
It’s entirely possible I was doing something incorrect with the API code, but since we’re reaching out to third parties where the data doesn’t change too often, I figured a cache wouldn’t hurt.  
Since we’re writing ExpressJS middleware in Node.js, we can add in some middleware after we mount our orders routes – that way, they don’t get cached.  
As with fh.db before, fh.cache is a convenience wrapper on the Redis cache – of course, we could just use Redis directly.

**Code Snippet:**

api.use(function(req, res, next){

var cacheKey = \_.last(req.path.split('/'));

return fh.cache({ act : 'load', key : cacheKey }, function(err, cacheRes){

if (!err && cacheRes){

cacheRes = JSON.parse(cacheRes);

return res.json(cacheRes)

}

return next();

});

});

You can also see this middleware in action [on GitHub](https://github.com/birdhousesinc/mobile-microservice/blob/1c2e59924820e784ddc0829c96b841adbd9abe12/lib/api.js#L80-L89) – note that we included it after the ‘orders’ routes.

Now, we’re caching both Salesforce and Sharepoint responses – much better!

### The Mobile Specific Mashup

Now that we have two connectors, can return orders and products, and can create orders in our database microservice, there’s one piece of cleverness left to do. When we retrieve our list of orders from the database, we store the Id of the product in SharePoint, and the Id of the Account in Salesforce. These are foreign keys of sorts, into external systems, and we’d like to know more about these fields in our mobile apps. The solution is simple – when we return the field to our mobile app, replace its contents with the full Account and Product objects. We have this information in our cache, so there’s no need to do a full API call – so with some async Node.js magic, we can retrieve the information we need.

var async = require(‘async’);

// Load both our Accounts list and our Products list from our cache

async.parallel({

accounts : async.apply(fh.cache, { act : 'load', key : 'accounts' }),

products : async.apply(fh.cache, { act : 'load', key : 'products' })

}, function(err, cacheResults){

// We store stringified values in the cache.

// JSON.parse them & we have a full list of accounts and products

var accounts = JSON.parse(cacheResults.accounts),

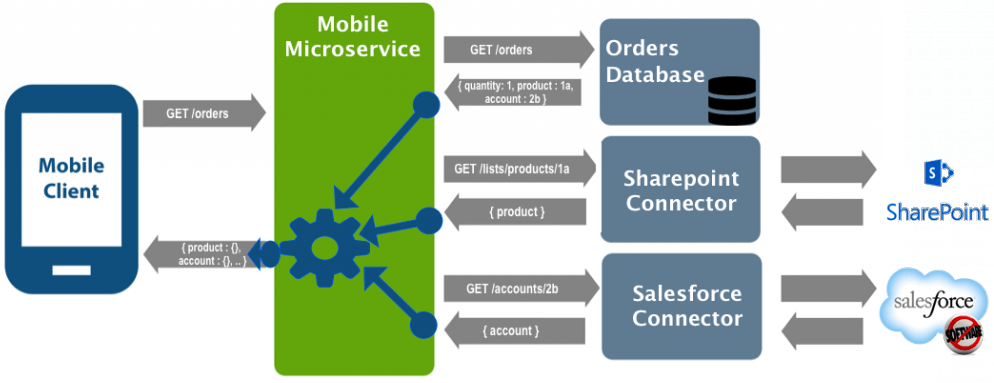
products = JSON.parse(cacheResults.products);

//TODO: Now we just need to find the relevant account and product,

// and include this in our response

});

You can see the full code [on GitHub](https://github.com/birdhousesinc/mobile-microservice/blob/1c2e59924820e784ddc0829c96b841adbd9abe12/lib/api.js#L30-L68).

[](http://www.feedhenry.com/wp-content/uploads/2015/11/mashup2.png)  
The Mashup Process

### Writing the Mobile Client

We have our server-side code finished, now I’m going to create a hybrid mobile app to utilize these new services. We’re going to [inject the contents of our products listing from Sharepoint](https://github.com/birdhousesinc/ordering-app/blob/master/www/hello.js#L65-L67) into a select element, and our [accounts listing into another select element](https://github.com/birdhousesinc/ordering-app/blob/master/www/hello.js#L49-L51). We’re going to [list all completed orders](https://github.com/birdhousesinc/ordering-app/blob/master/www/hello.js#L27-L40), and [draw a table](https://github.com/birdhousesinc/ordering-app/blob/master/www/hello.js#L33-L35) – making use of our mashup data by retrieving information which wouldn’t otherwise be available to us such as the account name and phone number.

## The Result

Through the use of a microservices architecture, we’ve enabled field staff to place orders, and retrieve historical order information. We’re no longer at ransom to big vendors for our data – instead of storing this information in a proprietary data silo, we’ve taken steps to ensure we take control and ownership. You can see all the components of this example [on GitHub](https://github.com/birdhousesinc).

Through our use of connecting microservices to integrate third party systems, and business logic in our mobile specific microservice, we’ve ensured our relationship with third party software is loosely coupled.  
We can swap out integrations for other systems in future should we ever choose to migrate, or take even more of this data in house.

Adopting a microservices approach for mobile helps us reduce our costs, iterate faster, and helps with future expansibility of our mobility efforts. It should be considered an essential part of any enterprise app developer’s toolkit.

[1] Evolving a Mobile-centric Architecture: The Microservices Way  
<https://developers.redhat.com/blog/2015/04/15/evolving-a-mobile-centric-architecture-the-microservices-way/>

# Evolving a Mobile-centric Architecture: The Microservices Way

Posted by [cianclarke](http://developers.redhat.com/blog/author/cianclarke/) on April 15, 2015

[](https://rhdevelopers.files.wordpress.com/2015/04/feedhenry_byrh_rgb.png)There seems to have been an explosion in the use of the term “microservices” recently. I’ve been peripherally aware of the concept for some time now, but it seems it first came to light with a fantastic collection of [thoughts by Martin Fowler](http://martinfowler.com/articles/microservices.html)[1] – some great reading on the topic.

This three-part post will not help you make a business case for rewriting your existing monolith as a series of microservices – for that, I’d recommend [Rich Sharples’ writings on the topic](http://blog.softwhere.org/2015/02/21/microservices-the-new-architecture-for-digital-engagement/) [2] – but I am going to show how microservices make sense for mobile in a series of practical examples.

1. A Microservices Primer: Introducing the concepts, along with mobile-specific microservice considerations
2. Hands-on examples how to build some simple microservices
3. A benchmarking exercise revealing the impact microservices-based architectures can have on mobile.

The end result of this exercise can be performance improvements between ~**2.5x** and **~250x**, depending on network conditions[3]. To see a 10 minute video presentation [click here](http://www2.feedhenry.com/mobile-tech-talk-microservices)

# 1: A Microservices Primer

## Microservices – The Term

I’ve been somewhat amused by the term microservices, since we’ve been composing small, loosely coupled applications which combine to do work since I first started using Node.js with FeedHenry. I’d love to claim some visionary stroke of trend-predicting genius, but really it’s the path the Node.js community led us down. Of course, it’s widely accepted the concept has been around since Unix days – it’s just a shiny new name.

In the Node community, what started as cries of “Make everything you possibly can as small re-usable modules” (micromodules anybody?) quickly became “Make everything small re-usable applications,” and now we’re calling them microservices.

## Microservices – Considerations for Mobile

Introducing a microservices-based architecture has some specific considerations when it comes to delivering content to mobile applications. I’m going to deal with two main concerns – **coupling and performance**.

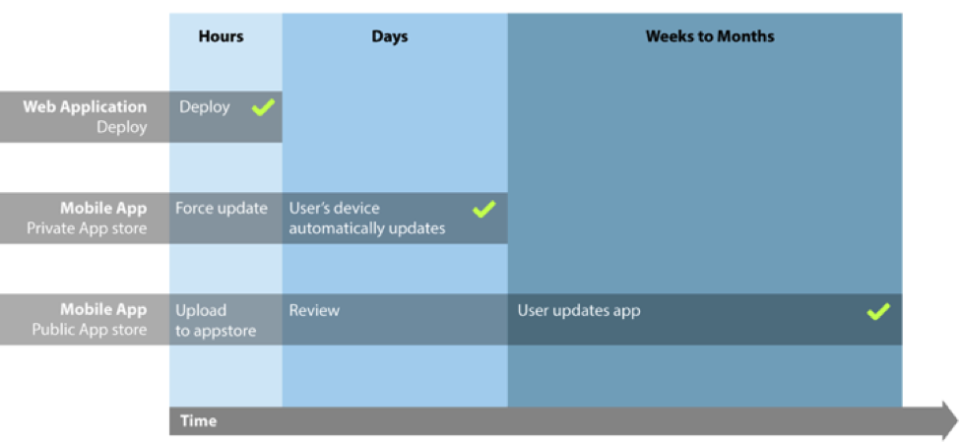
### Loose-Coupled, Tight-Coupled, Practically Welded Shut

For web applications, if we change the API we know that once we deploy an update to the web application, all our connected clients are using the new API. It’s easy to swap out URLs, and even expected payloads in the code of the web application, because we can deploy in tandem. Then we can deprecate the old API. This makes for a relatively loose coupling between client and server.

Mobile applications are different, however. A mobile app is released into an app store. In an enterprise environment, this app store is often private. We can usually force an update out to our app and watch it propagate to users within a matter of days. This makes the relationship between client and API more tightly coupled than in the case of the web application.

Releasing to the public app store, there may be a review period, but once released, users download this update over the course of weeks, months, maybe never. The previous API still needs to be maintained, and this makes for an integration which is so tightly coupled, it’s practically welded shut. (See, these days everybody is coining new terms!)

This makes for some very special considerations when architecting for mobile. The lifespan of any APIs the mobile application touches need to be much longer-lived, and APIs should be built with this in mind. Consider versioning, and some way of forcing users to update.

[](https://rhdevelopers.files.wordpress.com/2015/04/microservices_diag1.png)

Mobile release cycles are much longer, leading to tighter coupling between client and API

### Performance Considerations

The other major consideration specific to mobile is performance. Web applications typically run on devices connected over WiFi or Ethernet, with low latency. Mobile devices often connect over lossy connections – 3G, Edge, or even GPRS.

Returning the minimum payload required to render the screen is more important now than ever. Intelligent pagination on lists can drastically reduce payload size, especially when users are only operating on the most recent items. This also means reducing the number of calls made across the network. As the number of calls grow, every HTTP transaction can contribute to an exponential growth in overall response time. Creating a microservice to act as a unified mobile API, which returns data from many sources in one single call is often a good option.

This can introduce some interesting trade-offs that need to be balanced:

1. At what point does a unified API returning multiple types of data compromise the RESTful nature of an API?
2. At what point does the response body size of a combined payload negate any potential performance gains?

# 2. A Mobile Example of a Microservice

In part 1, we introduced the term, and presented some unique considerations for mobile. Now, let’s take a more practical look by creating some microservices.

### The Tale of the Traveling Umbrella Salesperson

To illustrate the use of microservices, let’s take a mock use case, that of a traveling umbrella salesperson (of no relation to the Traveling Salesperson of Distributed Computing fame). This salesperson needs to be able to:

1. Create orders in a back-end database
2. Automatically scale their order quantities according to the weather (I know – work with me here, people…)
3. Notify the account manager of the new order on their account

### Our first Microservice – Orders

First, we’re going to create a microservice for orders for our traveling umbrella sales team. We’re going to write our microservices in Node.js, and they’re going to communicate JSON payloads over HTTP, but these are by no means prerequisites. Microservices can of course be implemented using any programming language, over any communication protocol.

Here’s a service which can both create and list umbrella orders. It creates a REST API, /orders.

var app = require('express')().use(require('body-parser')());

var orders = [];

// Create a new order

app.post('/orders', function(req, res){

orders.push(req.body);

return res.json(req.body);

});

// list orders

app.get('/orders', function(req, res){

return res.json(orders);

});

var server = app.listen(3000);

Ten lines – pretty micro, huh? Ok, so I cheated a bit – that first line is concise to the point of unreadable, and we’re just putting orders in memory – but we now have a really micro service.

For future code snippets, I’m going to drop some of the boilerplate setup code on the first and last lines

## Adding more Microservices – Weather and SMS

Our order service is working just fine – our folk out in the field can create orders, and list the orders they’ve previously created. But before the team creates an order, they want the system to scale their order size based on the upcoming weather forecast at that location. Let’s call this the rain service. Here, we’re going to reach out to a third party API, sum the rainfall totals for the upcoming forecast, and append it to the original weather information.

var weatherUrl = 'http://api.openweathermap.org/data/2.5/forecast';

app.get('/rain', function(req, res){

var city = req.query.city,

country = req.query.country;

request.get({url : weatherUrl + '?q=' + city + ',' + country, json : true}, function(err, response, weatherbody){

// sum all the inches rainfall in the forecast

weatherbody.rainfall = \_.reduce(weatherbody.list, function(a, b){ return a + b.rain['3h'] }, 0);

return res.json(weatherbody);

});

});

Lastly, we’re also going to add a service to allow us to push an SMS alert to the account manager when a new order is created.

app.post('/sms', function(req, res){

var to = req.body.to,

message = req.body.message;

client.sms.messages.create({ to: to, from : process.env.TWILIO\_NUM, body : message}, function(error, message){

return res.json(message);

});

});

app.listen(3002);

We’ve now created our series of three microservices. To get started with the examples provided, follow these steps in a terminal:

# clone the repository

git clone https://github.com/cianclarke/microservices-primer.gitcd microservices-primer

# Install dependencies

npm install -d

# Set Twilio environment variables

export TWILIO\_AUTH=foo; export TWILIO\_SID=bar; export TWILIO\_NUM="+1234567";

# start the 4 microservices & the test runner

npm start

# To view the test runner, visit http://localhost:3004/ in a browser.

We’re now running our series of microservices, and can interact with them using CURL

# Service 1: Create a new order in the database

curl 'http://127.0.0.1:3000/orders/umbrellas' -H 'Content-Type: application/json' --data-binary '{"city":"Dublin","country":"Ireland","quantity":984.4999999999999,"accountManager":"+1 123 456 789"}'

# Service 1: List orders in the database

curl 'http://127.0.0.1:3000/orders/umbrellas'

# Service 2: GET request to rain svc to retrieve info for Dublin

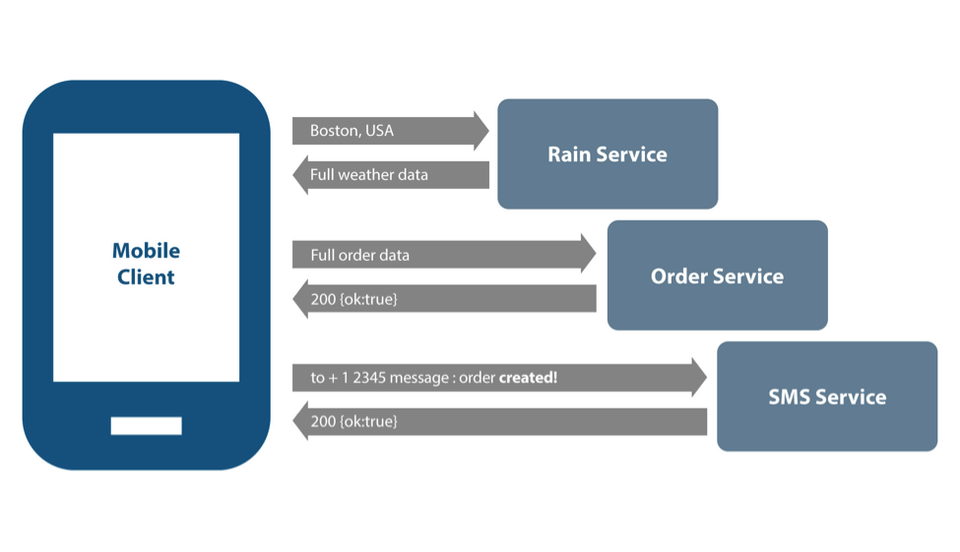
curl 'http://127.0.0.1:3001/rain?city=Dublin&country=Ireland'

# Service 3: POST to the SMS service

curl 'http://127.0.0.1:3002/sms' -H 'Content-Type: application/json' --data-binary '{"to":"+1 123 456 789","message":"My SMS Message!"}'

We’ve now built a series of microservices, and verified we can interact with them.3: Taking our Microservices MobileNow, let’s look at two different approaches to consuming these. The first will show a more traditional approach to building out the application. The second will introduce another microservice specifically for mobile, taking into account the mobile-specific concerns raised in part 1. Then, we’ll look at some results of our benchmark.

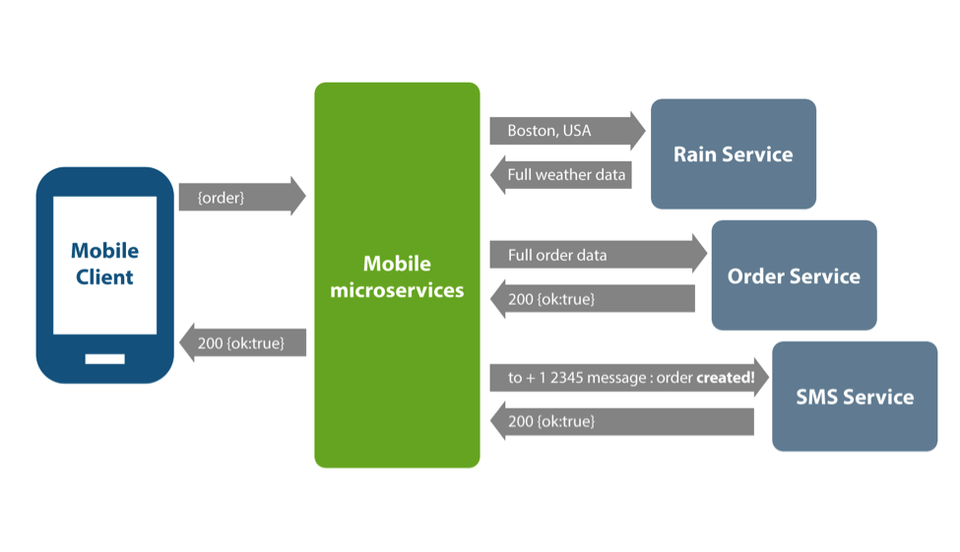
## Take 1: Client-Side Business Logic

First, we’ll build this application the way many existing mobile apps are built – we’ll implement a lot of business logic on the client (scaling the order according to the rainfall microservice, submitting to the orders microservice, then notifying the account manager via the SMS microservice), and make three separate REST calls from the mobile device.Sure, we’ve still got microservices on the server-side – but we could equally picture this as a monolith, for what little use we’re making of the microservices philosophy.[](https://rhdevelopers.files.wordpress.com/2015/04/microservices_2.png)

This illustrates the “wrong way.”

* We’re making several calls from the app, so we’re not minimizing our number of requests.
* None of the calls have been optimized for mobile. We’re sending back unnecessary data, so we’re not trimming the payload for mobile.

## Take 2: Microservices for Mobile

As before, we’re going to achieve this integration using a series of microservices – but we’re going to meld the data together in a fourth and final mobile-specific microservice.[](https://rhdevelopers.files.wordpress.com/2015/04/microservices-3.png)

Now, we send our order information to a new mobile specific microservice which wraps up the three steps we took in “take 1.” We’re now making one call from the mobile device. We’ve got one simple API to maintain, a perfectly reasonable POST of a JSON payload, and this integration becomes a much looser coupling. We’re not sending unnecessary data back to the mobile app.

## Benchmarking – Client-side Logic vs. Microservices approach across mobile networks

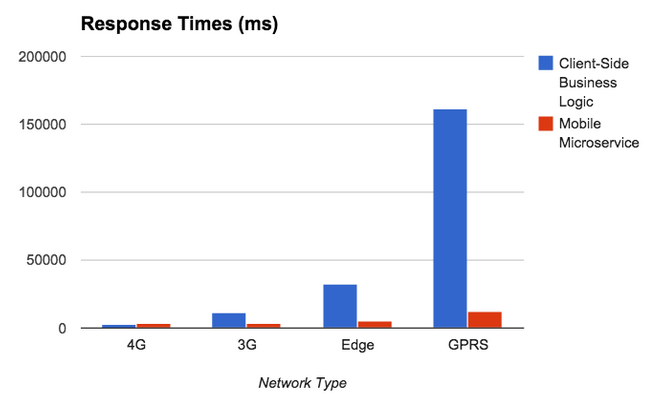
Now, let’s see how these two approaches compare across a range of mobile network types. If interested, some notes on the benchmarks are available in the footnotes[3].

A simple mobile client implementing the two approaches to integrating discussed above was created, which also measures request time across each approach.

### Payload Size

First, a brief look at the total payload size over 100 requests. Using the mobile microservice, **68kb** of data was exchanged. Calling each microservice individually resulted in **1.5mb** of data, a substantial increase likely due to returning weather information in it’s entirety on every request.

### Response Times

[](https://rhdevelopers.files.wordpress.com/2015/04/responsetimes.png)

Examining the results, we see nominal difference between approaches using a 4G network. When we move to a 3G network, the gap widens – there’s eight **seconds** in the difference across requests. Edge networks show a wider gap still, with an increase in average request time of almost **27 seconds**. At this point, the end user would really feel the pain of an inferior architecture. Lastly, over GPRS the impact of implementing multiple calls on the client is eye-opening. With average request times of **more than 2 minutes**, the application would be effectively unusable. Compare this to a microservice based architecture, with response times averaging 12 seconds, the application is slow but still usable. The key take away from this benchmark is that as the network slows, **response times grow exponentially** when not considering mobile in your microservice architecture.

## Conclusion

Having seen the results, what can we conclude from these benchmarks? I’m not going to try make such bold, linkbait-esque claims as “Microservices make mobile 10x faster” – that’s simply not true. What is true, however, is that the roll-out of a Microservices based architecture needs to consider mobile as a first class citizen. Not doing so can ruin the user experience for end users, and render applications virtually useless on slower networks. If the above considerations are taken into approach, the rollout of this new breed of architecture should prove a much smoother transition.

# Further Reading

[1] Martin Fowler – Microservices. The post which for many started the momentum behind the term.  
<http://martinfowler.com/articles/microservices.html>

[2] Rich Sharples – Micro services – the new architecture for digital engagement? Some observations from a higher level on the trends and applications of microservices.  
<http://blog.softwhere.org/2015/02/21/microservices-the-new-architecture-for-digital-engagement/>

[3] 100 requests in each run. SMS API calls to Twilio simulated with 250ms timeout to avoid excessive calls to their API. Chrome Developer Tools network throttling used to simulate network speeds. Node.js processes restarted after every batch of 100 requests. Requests had timestamp appended to prevent browser caching.

Watch a powerful [10-minute video](http://www2.feedhenry.com/mobile-tech-talk-microservices) on this content that includes a live demo of the benchmarking exercise.

[*@cianclarke*](http://www.twitter.com/cianclarke)is a Software Engineer with FeedHenry. Primarily focused on the mobile-backend-as-a-service space, Cian is responsible for many of FeedHenry’s mBaaS developer features.For other mobile topics visit the [*Mobile Tech Talks*](http://www2.feedhenry.com/mobile-tech-talks-rh).You may also be interested in building a business case for a mobile application platform with the help of this [*eBook*](http://www2.feedhenry.com/business-case-mobile-app-platform-rh).

**Microservice principles and Immutability – demonstrated with Apache Spark and Cassandra**

Posted by [jayunit100](http://developers.redhat.com/blog/author/jayunit100/) on January 20, 2015

**[](https://rhdevelopers.files.wordpress.com/2014/11/shipping_containers_at_clyde.jpg)Containerizing things** is particularly popular these days.   Today we’ll talk about the idioms we can use for containerization, and specifically play with apache **spark** and **cassandra** in our use case for creating easily deployed, immutable microservices.

*Note: This post is done using centos7 as a base for the containers, but these same recipes will apply with RHEL and Fedora base images.*

There are a few different ways to build a container.  For example, for beginners, you can build a container  as a .  but in some ways, this can be a bit of an [anti-pattern.](http://en.wikipedia.org/wiki/Anti-pattern) Rather than serving as lightweight VMs, we are seeing that the common practice for containerization of an application **is evolving in lock step with the** [microservices](http://microservices.io/patterns/microservices.html) movement, which is quite different from the pre-container world view of scalable systems (which involved, typically, **mutable services on VMs and heavy-weight provisioning**).

One of the most exciting things about containers, is that they **(if designed properly)** can embrace the **immutability** principle, which is a fundamental concept behind many increasingly popular [functional programming languages](http://www.ibm.com/developerworks/library/j-ft4/) (which gave rise to frameworks such as Spark and MapReduce), and highly reliable software systems which are easy to parallelize operations on top of.  Immutable services can, by definition, be deployed without any heavy weight installers or configuration management… this paves the way for a new paradigm in load balancing, high availability, and dynamic resource sharing.

Enough high level stuff…  To really grok these connections, you need to build a distributed system with containers from scratch.   **In this post,** I’ll walk through **two** different ways to containerize an application.  There are many ways to containerize an application, but currently ***the microservices movement, which is becoming a more popular idiom for using docker, provides a lot of benefits which cannot be realized on just “any” Dockerfile.***Additionally, for folks interested in spark, we will demonstrate how to orchestrate and test a spark microservices cluster using Docker on Centos.  And, even if you don’t fully buy into the microservices  ecosystem in its current state, building clean and simple containers which do one thing perfectly is probably never a mistake.

For those wondering how you can use **vagrant** in the post VM universe which we are now living in, this post should also be helpful.  To leverage the snippets here, you will need Docker and Vagrant  installed.  If you are on a non linux system, you can install VMWare/VirtualBox and have vagrant launch the containers for you in a VM of your choice which is docker friendly.  In any case, I have tried to keep the code snippets to the point so that this post can be read in isolation.

I hope that, by the end of this post, a light bulb will go off in your head, and you to will see the connection between **immutable infrastructure,** idiomatically designed **microservices,** and **portable testing** of distributed applications (in this case, we will use vagrant, but you can use any orchestration framework you want to test a microservice, thanks to its self-sufficient and easily composable nature – even a shell script).

This intimate connection will be realized in this post by building some dockerized spark containers (first, the wrong way), and then rebuilding them as proper standalone microservices.

So lets get started !

**In the first attempt,** we will create some simple containers, and use **SSH** to get into them, so that we can start some services.  This is a common design pattern for typical cloud based infrastructures and testing, and it is extremely flexible and easy to hack around with – but a its also an anti-pattern… the mutable nature of the VMs leads to many things which can go wrong, and also to systems which might be poorly documented, with multipurpose functionality that isn’t composable, or easily load balanced.

*However, by implementing this anti-pattern, we can better appreciate the microservices design pattern, in the way that it really properly implemented.*   The reason this is important is that the definition of a microservice is quite vague, and in my opinion, it allows for quite a [few anti-patterns](http://jpetazzo.github.io/2014/06/23/docker-ssh-considered-evil/), like ssh provisioning.  According to Wikipedia:

“In [computing](http://en.wikipedia.org/wiki/Computing), **microservices** is a [software architecture design pattern](http://en.wikipedia.org/wiki/Architectural_pattern), in which complex [applications](http://en.wikipedia.org/wiki/Software_application) are composed of small, independent [processes](http://en.wikipedia.org/wiki/Process_%28computing%29) communicating with each other using language-agnostic [APIs](http://en.wikipedia.org/wiki/Application_programming_interface). These [services](http://en.wikipedia.org/wiki/Service_%28systems_architecture%29) are small, highly [decoupled](http://en.wikipedia.org/wiki/Coupling_%28computer_programming%29) and focus on doing a small task.”

This leaves alot to the imagination.  So… lets play around with some different ways to create a docker container that runs Apache Spark.  Apache Spark is a distributed Big Data processing framework with a master/slave architecture (you need one master, and at least one slave).    However the principals can apply to any distributed system.

**SSHD: A “naughty” microservice  ?**

As a long time vagrant user, I’m used to building vagrant infrastructure using this workflow.

1. Define a “box” (i.e. an OS image).
2. Make sure the “box” has SSH credentials, and SSH running.
3. Write a Vagrantfile with my application semantics.

Come to think of it – *vagrant boxes are actually in some ways designed along a microservice pattern* – where the box itself does  one thing, and one thing very well : spin up and run SSHD.  After that, puppet, chef, or pure shell commands are run in the box to provision software.

So, here is an SSHD based microservice, in a Dockerfile.

FROM centos:centos7

RUN yum clean all

RUN yum install -y yum-utils

RUN yum-config-manager --save --setopt=fedora.skip\_if\_unavailable=true

RUN yum update -y

RUN yum install -y wget

RUN wget -O /etc/yum.repos.d/bigtop.repo http://bigtop01.cloudera.org:8080/view/Releases/job/Bigtop-0.8.0/label=centos6/6/artifact/output/bigtop.repo

#### Now install SSH so we can layer in the spark components.

RUN yum -y install openssh-server openssh-clients sudo

RUN sed -i.bak s/UsePAM\ yes/UsePAM\ no/ /etc/ssh/sshd\_config

RUN ssh-keygen -q -N "" -t dsa -f /etc/ssh/ssh\_host\_dsa\_key

RUN ssh-keygen -q -N "" -t rsa -f /etc/ssh/ssh\_host\_rsa\_key

# requiretty off

RUN sed -i.bak 's/requiretty/!requiretty/' /etc/sudoers

# setup vagrant account

RUN mkdir /root/.ssh/

RUN chmod 0755 /root/.ssh

RUN wget http://github.com/mitchellh/vagrant/raw/master/keys/vagrant.pub --no-check-certificate -O /root/.ssh/authorized\_keys

RUN chmod 0644 /root/.ssh/authorized\_keys

CMD /usr/sbin/sshd -D

This Dockerfile does a few simple things.  It adds some yum repos to a image, and installs openssh\*.  This is preparation for starting and configuring our **spark** as a service which we will run inside of a container.  Its useful for legacy interop (that is, for creating a Dockerfile which can service as a drop in replacement for a VM). We can now start some services in the container in a our provisioning framework.

(1..$spark\_num\_instances).each do |i|

config.vm.define "scale#{i}" do |scale|

scale.vm.provider "docker" do |d|

d.build\_dir = "spark/"

d.create\_args = ["--privileged=true", "-m", CONF["docker"]['memory\_size'] + "m"]

d.has\_ssh = true

if "#{i}" == "1"

scale.vm.provision "shell", inline:"yum install -y java-1.7.0-openjdk-devel.x86\_64"

scale.vm.provision "shell", inline:"yum install -y spark-worker"

scale.vm.provision "shell", inline:"echo \"export JAVA\_HOME=/usr/lib/jvm/java-1.7.0-openjdk/\" >> /etc/spark/conf/spark-env.sh"

scale.vm.provision "shell", inline:"echo \"export STANDALONE\_SPARK\_MASTER\_HOST=scale1.docker\" >> /etc/spark/conf/spark-env.sh"

So, with these two snippets, we now have a Vagrantfile which is going to layer some services into our docker container, using the SSHD microservice as the mechanism to get into the container. So, while its easy to modify the Vagrantfile to add new services into the image, there are some heavy costs we’re paying:

* The container is not immutable – its state is modified after it is started.  This means that it cannot be orchestrated as an atomic unit.
* The container will take a while to start : because every time we start it we have to *do some stuff* to it after the fact.

**Microservice Provisioning, the right way.**

Consider, the evolution of the container ecosystem.  We now have tools such as **kubernetes** which is rapidly becoming a standard for highly available containerized applications.  Kubernetes is based on the idea that a microservice, in and of itself, runs as a application service.  In this sense, microservices can be **composed** into higher level applications which run in a **fault tolerant, distributed** context.

The goal of a microservice should, at least partially, be leveraged as part of software that is maintained using a [**higher level framework, such as the components of Red Hat’s Atomic**](https://developers.redhat.com/blog/2014/07/10/repost-going-atomic-with-the-red-hat-enterprise-linux-7-high-touch-beta/) or even, a tool like [vagrant](http://vagrantup.com), with minimal changes.

Since we cannot really assume much about a higher level framework, this pretty much means that our **SSHD** service container is not a particularly good design choice (it requires the framework to setup and install stuff on our containers, which assumes the framework can SSH into those containers, knows what to install on which container, etc…).   This low level coupling is obviously suboptimal for a typical application, which  would like to be loosely coupled to composed services (i.e.  PostgreSQL (a RDBMS),  Vowpallwabbit (a real time classifier) , SOLR (a popular search engine), Apache (the popular web server), which comprise some typical resources an end to end application might want to rely on).

So, now lets look at a “real” microservice implementation, which satisfies the atomicity and immutability principles.

First, we will create a “jvm” container, which installs java and nothing else.  I won’t show that container here, for space purposes.  But, once you have it, you can simply create it locally using “docker build -t jvm ./”.  At that point, you can now build containers that leverage the base JVM container.

Now, for **spark** we will build a container which might look something like this (the if statement for looking at the hostname isn’t particularly important here, its an implementation detail for easy test spin up).

FROM jvm

RUN yum clean all

RUN yum install -y tar yum-utils wget

RUN yum-config-manager --save --setopt=fedora.skip\_if\_unavailable=true

RUN yum update -y

RUN yum install -y java-1.7.0-openjdk-devel.x86\_64

COPY spark-1.2.0-bin-hadoop2.4.tgz /opt/

RUN tar -xzf /opt/spark-1.2.0-bin-hadoop2.4.tgz -C /opt/

RUN echo "SPARK\_HOME=/opt/spark-1.2.0-bin-hadoop2.4" >> /etc/environment

RUN echo "JAVA\_HOME=/usr/lib/jvm/java-1.7.0-openjdk/" >> /opt/spark-1.2.0-bin-hadoop2.4/conf/spark-env.sh

CMD if [[ `hostname` = 'scale1.docker' ]] ; then /opt/spark-1.2.0-bin-hadoop2.4/sbin/start-master.sh ; else ping -c 2 scale1.docker && /opt/spark-1.2.0-bin-hadoop2.4/sbin/start-slave.sh -h spark://scale1.docker:7077 ; fi ; tailf /opt/spark-1.2.0-bin-hadoop2.4/logs/\*

So what are the differences here?

* This time, we ran directly from a **spark tarball.** Since our container is doing one, and only one thing, the value of using a RPM or DEB packaged spark distribution is diminished – and we just use a tar distribution.  This is a common idiom you will see in microservices.
* The container **STARTS with a single service which is essential to our high level application.**Each container either runs a spark master, or a spark slave.
* For those new to docker, note that we use **COPY** here, to copy a LOCAL tgz file into /opt/

**Idiomatic Microservices** : **Immutability leads to testability…**

Now, lets look at how we orchestrate these components.  In Vagrant, we can easily create a lightweight orchestration layer that leverages dockers **–link** option, as a replacement for heavyweight fully functional orchestration frameworks.  This allows us to **test our microservices** in a lightweight and cross-platform manner.  This highlights the benefits of building stand-alone microservices : They are testable in any environment, because they have a minimal dependency on the orchestration layer which launches them.   Here is how our Vagrantfile for building these microservices will look, again, this a snippet, for explaining the high level concepts.

# number of instances : First one is master.

$spark\_num\_instances = 2

$cassandra\_num\_instances = 1

Vagrant.configure("2") do |config|

# nodes definition

(1..$spark\_num\_instances).each do |i|

config.vm.define "scale#{i}" do |scale|

scale.vm.provider "docker" do |d|

d.build\_dir = "spark/"

d.name = "scale#{i}"

d.create\_args = ["--privileged=true"]

d.remains\_running = true

if "#{i}" == "1"

d.ports = [ "4040:4040", "7707:7707" ]

else

d.create\_args = d.create\_args << "--link" << "scale1:scale1.docker"

end

end

scale.vm.synced\_folder "./", "/scale-shared/"

scale.vm.hostname = "scale#{i}.docker"

end

end

#With cassandra we don't have master/slave architecture.

(1..$cassandra\_num\_instances).each do |i|

config.vm.define "cassandra#{i}" do |scale|

scale.vm.provider "docker" do |d|

d.build\_dir = "cassandra/"

d.create\_args = ["--privileged=true", "-m", $conf["docker"]['memory\_size'] + "m"]

d.remains\_running = true

end

scale.vm.synced\_folder "./", "/scale-shared/"

scale.vm.hostname = "cassandra#{i}.docker"

end

end

end

The above vagrantfile creates containers using the **–link** option of docker which allows us to link one container to another, when it spins up.  So, it essentially follows this path outlined below.  Note that this implementation only allows one cassandra node.

1. If we are starting the 1st container, start up a spark master.
2. If we are starting any other spark container, start a slave.  link it to the already created master so that they can talk.
3. Start a single cassandra node.

To tie all of this together, here is how I test this setup.

echo "WARNING REMOVING ALL CONTAINERS in 5 SECONDS !"

sleep 5

# Remove all containers.

docker rm -f `docker ps --no-trunc -aq`

echo "NOW RESTARTING DOCKER !"

service docker restart

echo "NOW CREATING VAGRANT DOCKER CLUSTER "

vagrant destroy --force && vagrant up --no-parallel

## Run calculate pi.

echo "RUNNING smoke tests..."

docker exec -i -t scale1 \

/opt/spark-1.2.0-bin-hadoop2.4/bin/spark-submit \

--class org.apache.spark.examples.SparkPi \

--master spark://scale1.docker:7077 /scale-shared/spark-examples\_2.10-1.1.1.jar 10000

echo "DONE TESTING . RESULT OF PI CALCULATION ABOVE "

Note thate, the vagrant assigned names for the containers for us can be used to execute one off commands in a smoke test of your microservices architecture.  This can all be done using pure docker commands also, quite easily, of course.

This post should help get you started containerizing your bigdata, or other services, in a way which is **truly immutable** and thus **easily testable.**

**Watch it in action !**

Here’s a video which shows the entire application, including a distributed spark test against multiple running microservice containers in action. In this video, we are only running spark, but you can easily add some links to the cassandra container to test that manually as well. The current code for this is here [https://github.com/jayunit100/SparkStreamingCassandraDemo/tree/master/deploy/](https://github.com/jayunit100/SparkStreamingCassandraDemo/tree/master/deploy/spark), but it may be moved at some point.

Feel free to reach out to me personally or leave comments about these ideas.  Special thanks to [Tim St. Clair](http://timothysc.github.io/) at Red Hat for helping me to develop some of these concepts and differentiate “real” microservices  from “naughty” ones !

# Implementing microservices using Docker containers



[AnandNatraj Chandramohan](https://developer.ibm.com/wasdev/blog/author/anandcha/) / August 28, 2015 / [0 comments](https://developer.ibm.com/wasdev/docs/implementing-microservices-using-docker-containers/#respond)

A step-by-step guide to implementing two microservices using Docker containers and WAS Liberty.

### Share this:

* [LinkedIn](https://developer.ibm.com/wasdev/docs/implementing-microservices-using-docker-containers/?share=linkedin&nb=1)
* [Reddit](https://developer.ibm.com/wasdev/docs/implementing-microservices-using-docker-containers/?share=reddit&nb=1)
* [Twitter](https://developer.ibm.com/wasdev/docs/implementing-microservices-using-docker-containers/?share=twitter&nb=1)
* [Google](https://developer.ibm.com/wasdev/docs/implementing-microservices-using-docker-containers/?share=google-plus-1&nb=1)
* [Email](https://developer.ibm.com/wasdev/docs/implementing-microservices-using-docker-containers/?share=email&nb=1)
* [More](https://developer.ibm.com/wasdev/docs/implementing-microservices-using-docker-containers/)

Microservices is an architectural style in which large complex software applications are broken down into a collection of independent, loosely coupled services. Starting points to learn about microservices include [an excellent blog post by Martin Fowler](http://martinfowler.com/articles/microservices.html) and the [IBM Redbooks publication](https://www.redbooks.ibm.com/redbooks.nsf/RedpieceAbstracts/sg248275.html?Open). Also, review the [12-factor app for microservices](https://developer.ibm.com/wasdev/docs/creating-a-12-factor-application-with-was-liberty/) article by [Kate](https://developer.ibm.com/wasdev/blog/author/katheris/).

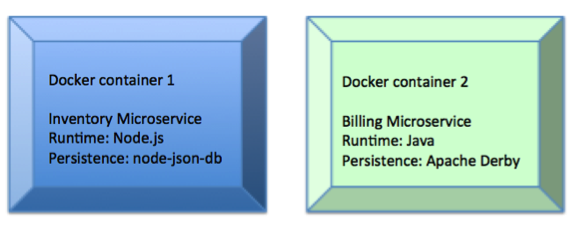
[Docker containers](https://www.docker.com/) wrap up a piece of software in a complete filesystem that contains everything it needs to run: code, runtime, system tools, system libraries – ensuring that it will run consistently across all environments – on-premises/public cloud/private cloud/VMWare etc.

## Building two microservices using Docker containers

In this article, we will put Docker and microservices together using an example microservices application composed of two services, Inventory and Billing. Each service will run in its own Docker container, and will communicate using lightweight REST APIs.

Microservices are small and focused, meaning that each service is responsible for one task. Technology choices can vary from service to service, but should be made based on what it is the service has to do, for example choosing between a Relational or NoSQL datastore based on the type of data the service interacts with. The software stack for the services in our example include the following components:

* **Inventory microservice**
  + runtime: Node.js
  + database: node-json-db and node.js REST client
* **Billing microservice**
  + runtime: WAS Liberty
  + database: Apache Derby

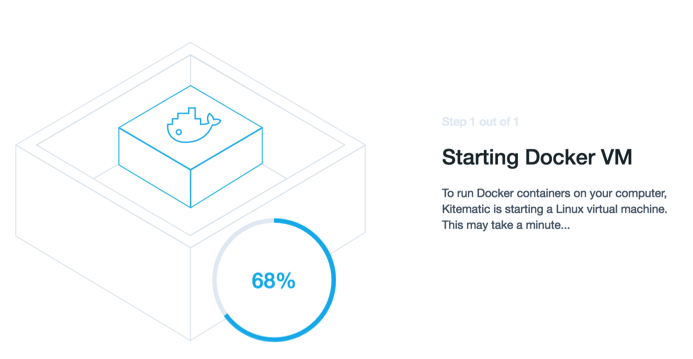
[](https://developer.ibm.com/wasdev/wp-content/uploads/sites/9/2015/08/1.png)

## Kitematic – a GUI for Docker

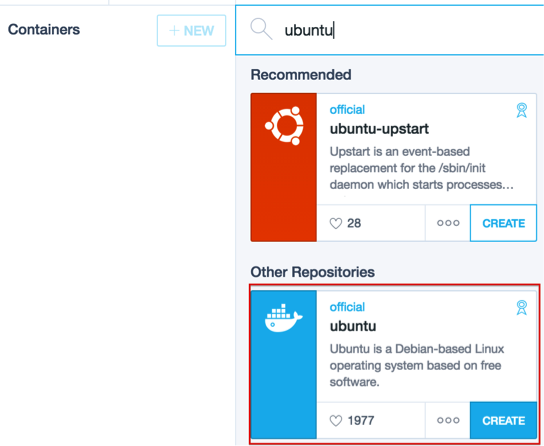
Before we start, you need to [get an account on Docker Hub](https://hub.docker.com/account/signup/).

Also, [download and install Kitematic](https://kitematic.com/), an easy and useful GUI-based tool for running Docker containers.

After installing Kitematic, start the application, which starts a Linux VM that is needed to run Docker containers:

[](https://developer.ibm.com/wasdev/wp-content/uploads/sites/9/2015/08/2.png)

On the home page, search for the Ubuntu image and click **Create** to create a container for the image:

[](https://developer.ibm.com/wasdev/wp-content/uploads/sites/9/2015/08/3.png)

Now, we’ll build two Docker images, one for the Inventory microservice and another for the Billing microservice. You can use Dockerfile to automatically build Docker images (see [Running Liberty profile in a Docker container](https://developer.ibm.com/wasdev/blog/2014/12/04/liberty-docker/) for a detailed example of building a Docker image for running WAS Liberty).

## Creating the Billing microservice

To build the Docker image for the Billing microservice, create a Dockerfile and add the following lines:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | FROM ubuntu    ADD wlp-extended-8.5.5.6.jar /root/  ADD wlp-runtime-8.5.5.6.jar /root/  ADD BillingMicroservice.war /root/  ADD db-derby-10.11.1.1-bin.tar.gz /root/    RUN apt-get update  RUN apt-get install -y default-jre  RUN java -jar /root/wlp-runtime-8.5.5.6.jar --acceptLicense /root/  RUN java -jar /root/wlp-extended-8.5.5.6.jar --acceptLicense /root/  EXPOSE 9080 |

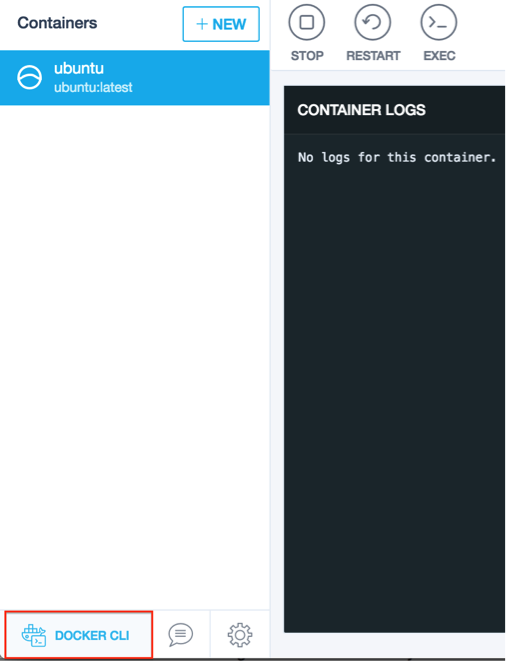
Download the [Liberty runtime](https://developer.ibm.com/assets/wasdev/#asset/runtimes-8.5.5-wlp-runtime) and [Liberty Extended Programming Models](https://developer.ibm.com/assets/wasdev/#asset/addons-8.5.5-wlp-extended) from the Liberty Repository. Download Apache Derby from [this link](http://db.apache.org/derby/releases/release-10.11.1.1.cgi)

Download the attached the WAR for the Billing microservice from here [BillingMicroservice.war](https://developer.ibm.com/wasdev/wp-content/uploads/sites/9/2015/08/BillingMicroservice.war_.zip)

Place the Dockerfile and the other four files in the same directory somewhere in your local file system. You should now see the following 5 files on your filesystem:

* BillingMicroservice.war
* db-derby-10.11.1.1-bin.tar.gz
* Dockerfile
* wlp-extended-8.5.5.6.jar
* wlp-runtime-8.5.5.6.jar

Next, from the Kitematic console, click **DOCKER CLI** to bring up the command line interface for working with Docker images:

[](https://developer.ibm.com/wasdev/wp-content/uploads/sites/9/2015/08/5.png)

In the shell, type the command docker images and you should see the following output:

REPOSITORY TAG IMAGE ID CREATED VIRTUAL SIZE

ubuntu latest d2a0ecffe6fa 2 weeks ago 188.4 MB

Next, navigate to the directory where you placed the Dockerfile and the other software downloads and issue the command:

docker build -t anandnatraj/wlp .

Note that the -t flag allows us to tag our successfully built image with a name. In my case, the anandnatraj also corresponds to the name of my private registry on Docker Hub, making it easy when to push my image to Docker Hub. This step will usually take a few minutes. At the end, you should see a message:

Successfully built ab6b3dd7be8f

Let’s now explicitly publish our container’s exposed ports to specified ports on the host machine by issuing the following command:

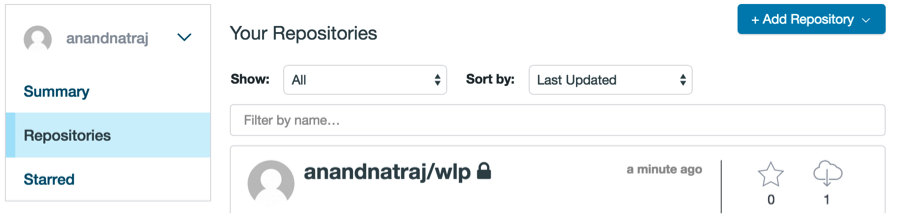
docker run -p 1930:9080 anandnatraj/wlp

What we’re going to do next is a bit of a workaround. Since the current version of Kitematic does not import local Docker images, we have to publish our image to Docker Hub and import it back to the local machine using Kitematic. Also, since the Docker image now contains licensed code, you cannot publish it to the public registry on Docker Hub. So create a private registry on Docker Hub and push your Docker image to it by issuing the following command (in this case, the name of the private registry is anandnatraj):

docker push anandnatraj/wlp

This step takes several minutes, so #coffeetime.

Log into Docker Hub and ensure your image has been published successfully.

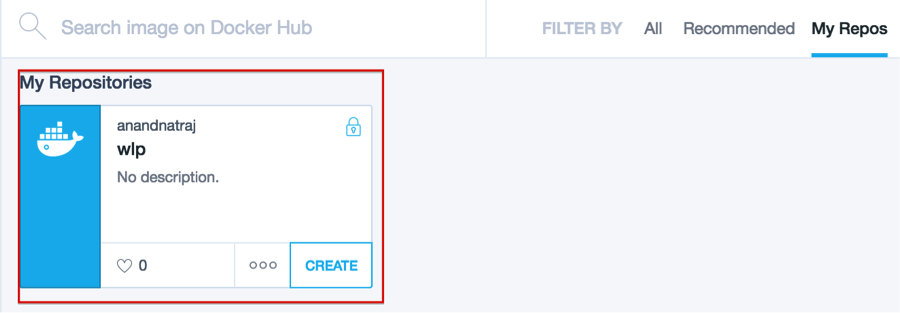
[](https://developer.ibm.com/wasdev/wp-content/uploads/sites/9/2015/08/6.png)

Next, go back to the Docker CLI shell and remove the existing local Docker image because we will download it from Docker Hub in the next step:

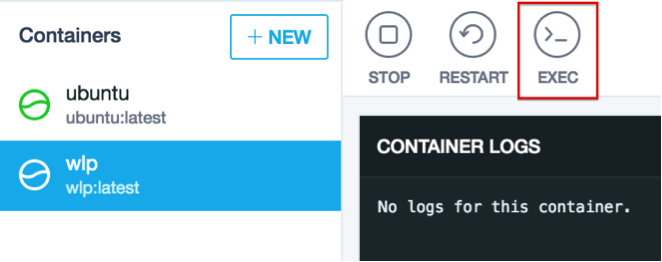
docker rmi anandnatraj/wlp

Now, launch the Kitematic console. Ensure you are signed into it using your Docker Hub account. Click **My Repos** on the top-left-hand corner and you should see the image you just pushed to Docker Hub. If you don’t see it, try signing out and signing back into Kitematic.

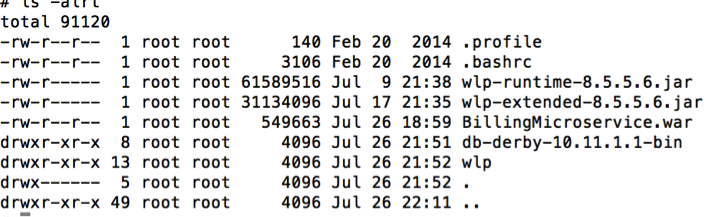
Click **Create** to download the image to your local machine and create a Docker container:

[](https://developer.ibm.com/wasdev/wp-content/uploads/sites/9/2015/08/7.png)

Once the download is complete, click **EXEC** to launch a shell to work with the Docker container:

[](https://developer.ibm.com/wasdev/wp-content/uploads/sites/9/2015/08/8.png)

Navigate to the /root directory and ensure you can see the files you added to the Docker image:

[](https://developer.ibm.com/wasdev/wp-content/uploads/sites/9/2015/08/9.png)

Now, the Billing microservice Docker container is almost ready. Next we have to start the Derby database and install the BillingMicroservice app on the WAS Liberty server.

So, navigate to /root/db-derby-10.11.1.1-bin/bin and type the following command to start the Derby database and to start the database:

./startNetworkServer

You’ll see the following message when it’s started:

Sun Jul 26 22:16:51 UTC 2015 : Apache Derby Network Server - 10.11.1.1 - (1616546) started and ready to accept connections on port 1527

If needed, you can launch another shell for the Docker container by selecting the **wlp** image and selecting **EXEC** on the Kitematic console.

Let’s now create a Liberty server by issuing the following command:

./server create mylibertyserver

Don’t start the server yet.

Install the BillingMicroservice app by copying it into the Liberty server’s dropins directory:

cp /root/BillingMicroservice.war /root/wlp/usr/servers/mylibertyserver/dropins/.

Before you start the server, navigate to /root/wlp/usr/servers/mylibertyserver and open the server.xml file. Replace all its contents and with the following lines:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17 | <?xml version="1.0" encoding="UTF-8"?>  <server description="new server">      <!-- Enable features -->      <featureManager>             <feature>jaxrs-1.1</feature>             <feature>jsp-2.2</feature>             <feature>localConnector-1.0</feature>      <feature>jsp-2.2</feature>      </featureManager>        <!-- To access this server from a remote client add a host attribute to the following element, e.g. host="\*" -->      <httpEndpoint id="defaultHttpEndpoint"                    host="\*"                    httpPort="9080"                    httpsPort="9443" />      <webApplication id="BillingMicroservice" location="BillingMicroservice.war" name="BillingMicroservice"/>  </server> |

Start the Liberty server:

./server start mylibertyserver

Now, ‘Docker-izing’ the BillingMicroservice is complete. The service itself is very simple. It performs the following functions:

1. Inserts Billing Information to the Apache Derby database.
2. Returns a status (always “verified” to keep things simple).
3. Exposed as a JAX-RS service. See [How to create a simple JAX-RS servlet](https://developer.ibm.com/wasdev/docs/jax-rs-basics/).

I pasted the skeleton code below for your reference. Note, you can use sample code from Apache to customize the java code below to access a derby database [sample code in the Apache Derby documentation](http://db.apache.org/derby/integrate/plugin_help/derby_app.html):

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31 | package net.wasdev.jaxrs;    import java.util.HashMap;  import java.util.Map;    import javax.ws.rs.POST;  import javax.ws.rs.Path;  import javax.ws.rs.Produces;  import javax.ws.rs.core.MediaType;    @Path("Billing/microservice")  public class Billingmicroservice {        private static String status = "verified";        @POST      @Produces(MediaType.APPLICATION\_JSON)        public Map<String, String> getSystemProperties(Map<String, String> input) {            System.out.println("Insert records to Apache Derby database");          /\*           \* Apache Derby database record insert           \*/          Map<String, String> result = new HashMap<>();          result.put("result", status);          return result;        }    } |

## Creating the Inventory microservice

In the next step, let’s build another Docker image with the Inventory microservice running on Node.js. On your host machine, create a new Dockerfile for the Inventory microservice image with the following lines:

|  |  |
| --- | --- |
| 1  2  3 | FROM ubuntu  ADD InventoryMicroservice.js /root/  ADD createDB.js |

As before, ensure the Dockerfile and the other dependencies are in the same directory in your host machine. Download createDB.js and InventoryMicroservice.js from here: [Inventory Microservice Downloads](https://developer.ibm.com/wasdev/wp-content/uploads/sites/9/2015/08/Inventory.zip)

The following screenshot shows the Dockerfile and other dependencies in the same directory on the filesystem of your host machine:

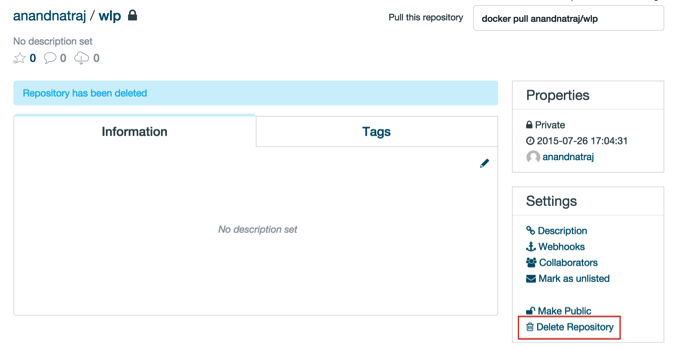
* createDB.js
* Dockerfile
* InventoryMicroservice.js

Launch a command shell from your host machine and navigate to the directory where you placed for the Dockerfile for the node image. Then build and push the image by issuing the following commands (note that a free account on Docker Hub allows you to only keep one image, so you may need to delete the previous image before pushing this new one):

docker build -t anandnatraj/node .

docker push anandnatraj/node

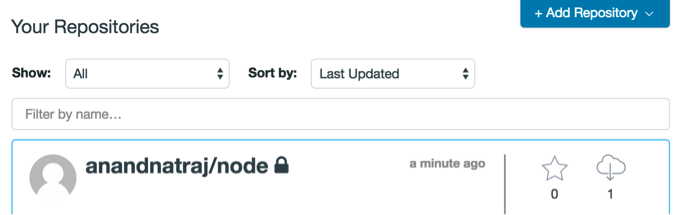
After pushing the image to Docker Hub, you can remove the local image, since we will be getting it back from Docker Hub (remember the work around we used above to get around the restriction of Kitematic not being able to pull images from a local registry).

[](https://developer.ibm.com/wasdev/wp-content/uploads/sites/9/2015/08/11.png)

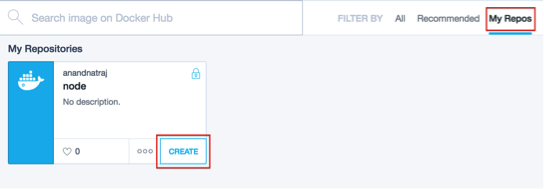
docker rmi anandnatraj/node

These steps, will take several minutes, so #coffeetimeagain.

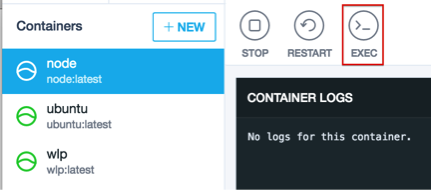
Again, log in to your Docker account to ensure the image made it through successfully:

[](https://developer.ibm.com/wasdev/wp-content/uploads/sites/9/2015/08/12.png)

Now, launch the Kitematic console. Ensure you are signed into it using the Docker Hub account. Click **My Repos** on the top-left-hand corner and you should see the image you just pushed to Docker Hub. If you don’t see it, try signing out and signing back into Kitematic. Click **Create** to download the image to your local machine and create a Docker container:

[](https://developer.ibm.com/wasdev/wp-content/uploads/sites/9/2015/08/13.png)

Once the download is complete, click **EXEC** to launch a shell to work with the Docker container.

[](https://developer.ibm.com/wasdev/wp-content/uploads/sites/9/2015/08/14.png)

Navigate to the /root directory and ensure you can see the file you added to the Docker image. Issue the following commands to install the Node REST Client and Node JSON DB modules:

apt-get update

apt-get install nodejs [ Hit Y to continue if prompted]

apt-get install npm [ Hit Y to continue if prompted]

npm install node-rest-client

npm install node-json-db –save

The Inventory microservice has now been successfully “Docker-ized” as well. One small step is to update the end-point of the Billing microservice.

Open the Inventory microservice:

vi InventoryMicroservice.js

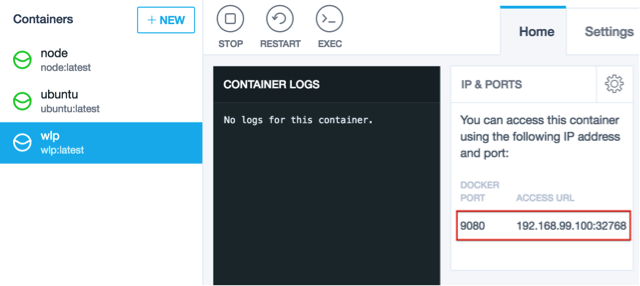
Replace the line:

client.registerMethod("getBillingInfo", "http://localhost:32768/BillingMicroservice/rest/Billing/microservice", "POST");

With the following line:

client.registerMethod("getBillingInfo", "http://192.168.99.100:32768/BillingMicroservice/rest/Billing/microservice", "POST");

You can get the exposed HTTP port of BillingMicroservice from the Kitematic console:

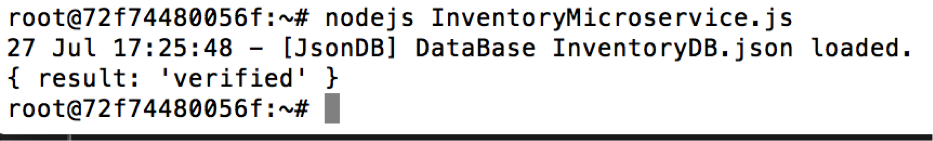
[](https://developer.ibm.com/wasdev/wp-content/uploads/sites/9/2015/08/15.png)

Create the required database for the Inventory service and then run the Inventory microservice:

nodejs createDB.js

nodejs InventoryMicroservice.js

You should see the following output:

[](https://developer.ibm.com/wasdev/wp-content/uploads/sites/9/2015/08/16.png)

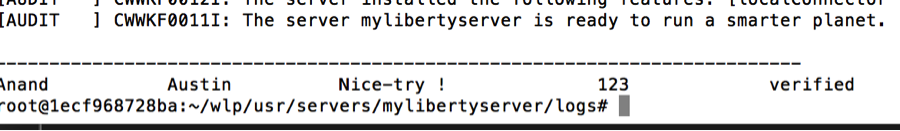
The Inventory microservice is very simple. It performs two simple functions:

1. Updates the Inventory information to a local node JSON DB.
2. Invokes BillingMicroservice running inside another Docker container using the lightweight REST API.

I pasted the code for the Inventory microservice here for your reference. The code for InventoryMicroservice is adapted from the [sample code for node-rest-client](https://www.npmjs.com/package/node-rest-client) and the code for createDB.js is adapted from [sample code for node-json-db](https://www.npmjs.com/package/node-json-db) on www.npmjs.com.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21 | var Client = require('node-rest-client').Client;  var client = new Client();    var myargs = {    data: { Name: "Anand",            Address:"Austin",            Age:"Nice-try !",            SSN:"123",          },    headers:{"Content-Type": "application/json"}  };    var myJSONDB = require('node-json-db');  var mydb = new myJSONDB("InventoryDataBase", true, false);  mydb.push("/Inventory",myargs);    client.registerMethod("getBillingInfo", "<http://localhost:32768/BillingMicroservice/rest/Billing/microservice>", "POST");    client.methods.getBillingInfo(args, function(data,response){    console.log(data);  }); |

Also, login to the shell for the Liberty container and navigate to /root/wlp/usr/servers/mylibertyserver/logs. Cat the console.log file and you should see the following output indicating the BillingMicroservice was successfully invoked:

[](https://developer.ibm.com/wasdev/wp-content/uploads/sites/9/2015/08/17.png)

## Conclusion

Docker and microservices form a very powerful combination. In this article, we examined a very simple example of running a microservices app in Docker containers. This simple microservices app consists of two microservices, Inventory and Billing, communicating via lightweight REST APIs.

**Top Tomcat Performance Problems: Database, Micro-Services and Frameworks**

February 23, 2016   |  by  [Andreas Grabner](http://apmblog.dynatrace.com/author/andreas-grabner/)  |  [2 Comments](http://apmblog.dynatrace.com/2016/02/23/top-tomcat-performance-problems-database-micro-services-and-frameworks/#comments)

Slow or excessive SQL Queries; wrong configured connection pools; excessive service, REST and remoting calls; overhead through excessive logging or inefficient exception handling; as well as bad coding leading to CPU Hotspots, Memory Leaks, impact through Garbage Collection or stuck threads through synchronization issues. These are some of the top performance problems I analyzed through my “[Share Your PurePath](http://bit.ly/sharepurepath)” program last year.

A big “Thank You!” to all our [Dynatrace Personal License](http://bit.ly/dtpersonal) users for sharing their data with me – allowing me to not only help them with a free performance review but also share it with a larger audience.

**Tomcat Performance Issues List**

Because I have many Tomcat users out there I compiled my top 10 problem pattern list I use when analyzing a Tomcat environment. This list also applies to other App Servers, so keep reading if your app runs on Jetty, JBoss, WebSphere, WebLogic, Glassfish, etc:

1. ***Database Access***: Loading too much data inefficiently
2. ***Micro-Service Access***: Inefficient access and badly designed Service APIs
3. ***Bad Frameworks***: Bottlenecks under load or misconfiguration
4. ***Bad Coding*:** CPU, Sync and Wait Hotspots
5. ***Inefficient Logging*:** Even too much for Splunk & ELK
6. ***Invisible Exceptions*:** Frameworks gone wild!
7. ***Exceptions*:** Overhead through Stack Trace generation
8. ***Pools & Queues*:** Bottlenecks through wrong Sizing
9. ***Multi-Threading*:** Locks, Syncs & Wait Issues
10. ***Memory*:** Leaks and Garbage Collection Impact

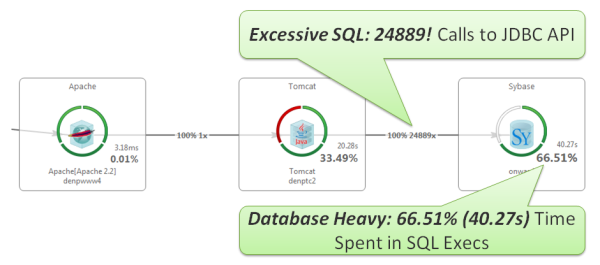
If your app suffers from any of these problems I can guarantee that not even Docker, EC2 or Azure will help you. Throwing more computing power at your problems will not help at all!

In Part I of this blog series I focus on how I identify problem patterns on *Database Access*, *Service Access* and *Bad Frameworks*. Part II & Part III will cover the remaining items. And who knows – if you are up for the challenge I have for you at the end of the blog post — there might be more to come!

I will be using Dynatrace as it is my tool of choice, but my approaches should also work using other APM or Performance Diagnostics Tools! If you want to watch a video rather than reading this blog series check out my [Tomcat Performance Analysis YouTube Tutorial](https://www.youtube.com/watch?v=FfmVM1BC5qQ&list=PLqt2rd0eew1bmDn54E2_M2uvbhm_WxY_6&index=36) instead.

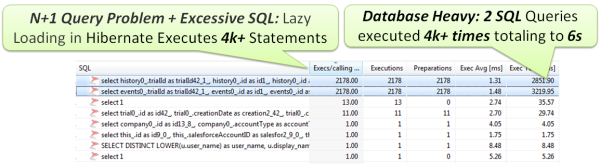
**#1: Database Access: Loading too much data inefficiently**

Believe it or not, loading too much data or loading data in an inefficient way is *THE #1 Problem* out there. And it’s not just me saying this – check out [Theodora’s blog On Top Java Performance Issues](http://www.javacodegeeks.com/2015/02/top-10-common-java-performance-problems.html), [The Netflix Blog on Databases](http://techblog.netflix.com/search/label/database) or Alois Reitbauers blog series on [Hibernate Performance](http://apmblog.dynatrace.com/2009/02/16/understanding-caching-in-hibernate-part-two-the-query-cache/). I can therefore easily live with comments such as “WOW – yet another N+1 query problem – don’t you have anything new to tell?” (recently received this comment via email). It’s up to you out there to whether to fix this issue: As long as I find this problem in > 80% of the performance data that people share with me I will present it as my #1 Problem. The following is my favorite view in Dynatrace to identify bad database access patterns — The Transaction Flow. Similar visualizations are available in other APM Tools. In Dynatrace you get this for every single request which also allows you to analyze bad database access patterns even though it might not yet lead to slow overall transaction performance:



Most often this happens through an incorrectly configured O/R-Mapper such as Hibernate, Spring or the ADO.NET Entity Framework. Simply watch out for the # of SQL Statements being executed per request!

Also, examine the actual SQL statements executed to identify the following patterns: N+1 Query Problem, Slow Running Statements, Unprepared Statements. I covered these and other database patterns in recent article I wrote for InfoQ: [Top Java Performance Hotspots](http://www.infoq.com/articles/Diagnosing-Common-Java-Database-Performance-Hotspots).



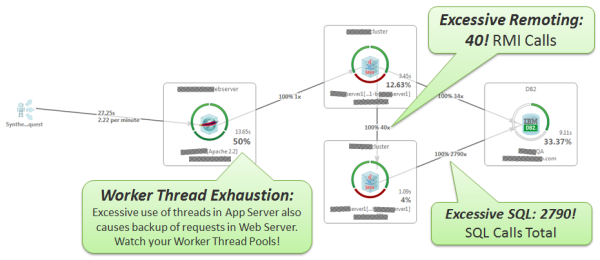
Don’t blindly trust your database access layer. Look at the individual SQL queries, execution count, type, on which connections they are executed and whether they are prepared or not. You will be surprised what you will find!

***Dynatrace Tip***: When drilling to the Database Dashlet I always click on the “Percentage of Transactions Calling” header. This will show me the table as shown above starting with the “Execs/calling transaction” -> If this column shows a value > 1 you know you have a potential N+1!

**#2: Micro-Service Access: Inefficient access and badly designed Service APIs**

With the continued trend towards service orientation I see the problem of apps becoming “very chatty” on these remoting interfaces. For every remote call your app is going to bind two threads (on the caller and callee side), you are going to transport data over the network and your app has to process the data. Whether you still use RMI or traditional SOAP Web Services or you have already moved towards “lightweight” REST Services using JSON: you want to avoid bad service interface design and excessive interaction with your services. The N+1 Query Problem is yet again a hot topic – but now it is between two App Servers vs. between App Server and Database.

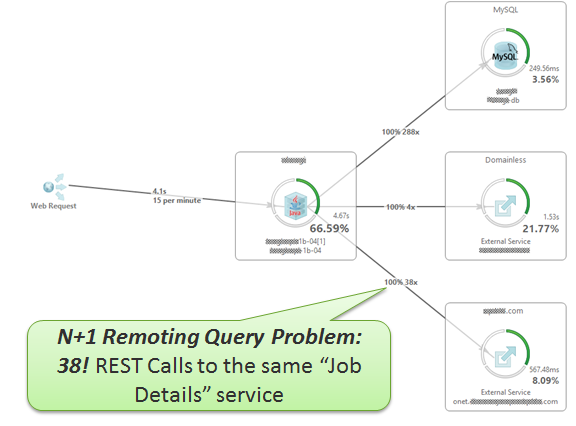
The following Transaction Flow shows a very classical architectural issue. The Frontend Tomcat Server makes 40 Individual RMI calls to a backend service! Each service call itself must access the database, totaling up to 2,790! SQL executions. This is a distributed N+1 Query Problem!



Service Oriented Architecture is not always implemented well. Make sure you know which service is invoking which other service and what the end-to-end impact on the system is. Keep an eye on every thread pool involved!

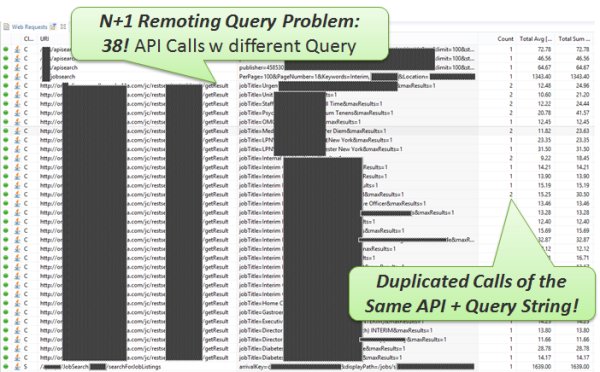
If your app makes service calls – or has a frontend web server like in the example above – make sure to monitor and correctly size all the connection pools involved. Apache has a connection pool in its Tomcat Connector. Tomcat itself has an incoming worker thread pool as well as an outgoing thread pool when making remoting calls to other systems. Pool size and Utilization are typically exposed via JMX. When you do your load and performance testing make sure to find the correct pool sizes for each and every pool in your system.

Most importantly though is to figure out whether all these calls are really necessary or whether there might be a better Service API to get your work done. The following screenshot is an example of a Job Search Portal. When executing a search the frontend Tomcat server is compiling a search result of 38 jobs found for a specific search. For each Job that matches the search criteria it makes a call to a Job Details Backend Service. This is a classical N+1 Query Problem. It would be smarter to provide a backend service API that returns the details of all Jobs that match the query. This would eliminate 37 Remoting calls!:



N+1 Query Problem with potentially even bigger impact than on the database level. Make sure you understand which APIs you call how often. Think about defining “smart APIs” that get the job done more efficiently!

A quick look at the individual Remoting Calls makes it easy to not only spot the N+1 Query Problem, it also highlights that some of these calls are actually done twice! In this case it is time to define a new remote API — getAllResults — and allow a more complex query string to pass instead of only individual Job Titles:



Just as with the SQL Statements you should analyze every single remoting call. It is easy to spot the access pattern. This is also great input for extending the backend service with new API calls to make this more efficient.

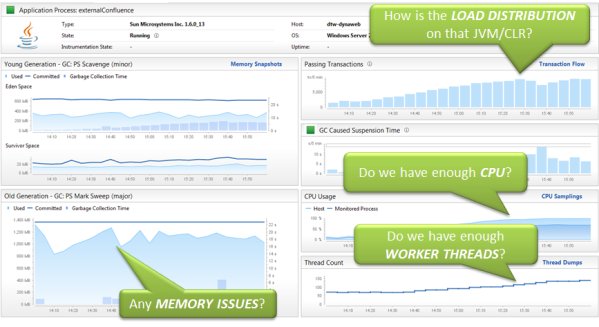
***Dynatrace Tip***: If you use Dynatrace, simply drill to the Web Requests Dashlet. Then make sure to select “Show -> All” and “Group By -> Uri and Query” in the Context Menu. This will now show you all remoting calls WITHIN a single transaction including URI and Query Parameters!

**#3: Bad Frameworks: Bottlenecks under load**

I assume that > 90% of the code that runs in your Tomcat is not your own code but instead Java Runtime, Spring, Hibernate, Netflix Hystrix, or any other cool library you downloaded on GitHub. We all build new software on top of existing frameworks instead of re-inventing the wheel every time we start a new project. The problem with this approach is that many developers or architects start with a sample or prototype implementation which then becomes the real code base without taking the time to figure out whether this framework will also work under “non demo” conditions.

A favorite approach of mine to figure out whether a framework is a potential performance bottleneck is to run an in *Increasing Load* load-test. You can pick any load testing tool available to you. My favorites are JMeter (or BlazeMeter), Neotys, SoapUI, SilkPerformer (I used to be an engineer on that team) or even LoadRunner.

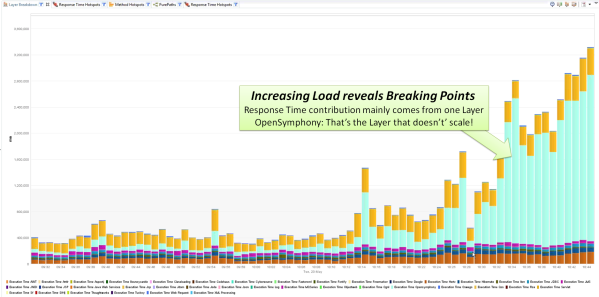
In an Increasing Load load-test I slowly start ramping up the load on the system until I reach a breaking point of the app. A breaking point for me is when Response Time spikes or I run out of CPU, Memory, Disk or Network bandwidth on my servers.



I always do a quick sanity check on host and process metrics before going into the application itself. If I already see that load is not correctly distributed behind the load balancer or if we memory sizes are incorrectly configured I stop here and fix these problems right away

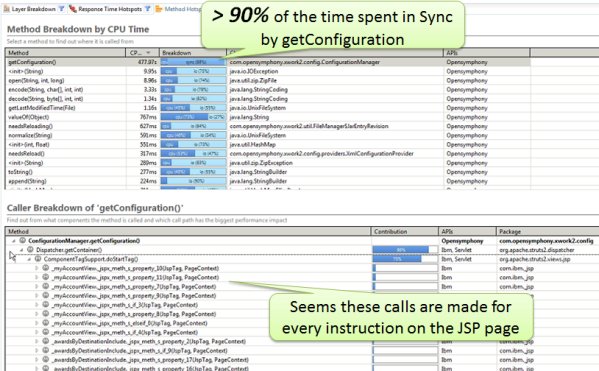
In 80% of the cases I can typically spot the problem by looking at what Dynatrace calls the *Layer Breakdown* chart. Other APM tools have similar views. The idea is that we break down response time into logical layers of your application code. A logical layer could be JDBC, Hibernate, Spring, RMI, Web Services, Caching Framwork, Your Own Code, with the other 20% configuration issues (pool sizes, load balancers, etc).

The following is a *Layer Breakdown* that clearly shows the breaking point in a Caching Framework used by this application. At a certain point into the load test the code in that Caching Framework showed disproportional contribution to the response time of the application.



The Layer Breakdown is the perfect starting point for hotspot analysis. Especially under varying load conditions it is easy to spot which layers of your application don’t scale!

Having identified the layer I start analyzing the actual methods in the code that cause the problem. I can even compare two timeframes (before the problem started and when we see the biggest spikes) to make identifying the root cause even easier. One of my favorite views here is the Methods Hotspot Dashlet that shows me which methods actually contributed to the execution time – even broken down into CPU, Synchronization, Wait, and Garbage Collection Suspension Time:



Make sure you always load and performance test the frameworks you are using. Consult the vendors and ask for best practice configurations.

I have a long history in Load Testing and Test Analysis, and it is one of my favorite tasks. I encourage you to read my two blog posts on Key Load Testing Metrics: [Part I](http://apmblog.dynatrace.com/2015/01/22/key-performance-metrics-load-tests-beyond-response-time-part/) & [Part II](http://apmblog.dynatrace.com/2015/01/28/key-performance-metrics-load-tests-beyond-response-time-part-ii/).

***Dynatrace Tip***: When analyzing Load Tests take a look at the Load Testing Overview Dashboard which you can open through the Start Center. Also check out my [Load Testing with Dynatrace YouTube Tutorial](https://www.youtube.com/watch?v=26nVM_JgStw&list=PLqt2rd0eew1bmDn54E2_M2uvbhm_WxY_6&index=15).

**Challenge: Prove me wrong or show me a new problem & win a speaking gig!**

I have the luxury and privilege of speaking at many user groups and conferences around the world – mainly presenting the top problem patterns I found in prior months and years. I would like to give *you* the opportunity to get on stage with me at a user group or conference that works for you. All you need to do is to either prove to me that your app is not suffering from these problems (demo apps don’t count) or you show me a new problem pattern that I do not yet have on the list!

***Challenge Accepted?*** If so – just sign up for the [Dynatrace Personal License](http://bit.ly/dtpersonal). After the 30-day trial period it remains FREE FOR LIFE for you to analyze your local apps. After you have signed up and receive the license file (check your spam folder for emails from [agrabner@dynatrace.com](mailto:agrabner@dynatrace.com)) you have two options:

1. Full Install of Dynatrace in your environment -> Download and Install [from here](https://community.dynatrace.com/community/display/EVAL/Step+1+-+Download+and+install+Dynatrace)!
2. Just use the pre-configured [Dynatrace Docker Containers on GitHub](http://dynatrace.github.io/Dynatrace-Docker/) -> special thanks to my colleague [Martin Etmajer](https://twitter.com/metmajer)!

I also recommend you check out my YouTube Tutorials on [What Is Dynatrace and How Does it Work](https://www.youtube.com/watch?v=2ycuNlYUl9E&list=PLqt2rd0eew1bmDn54E2_M2uvbhm_WxY_6&index=1) as well as [Tomcat Performance Analysis](https://www.youtube.com/watch?v=FfmVM1BC5qQ&list=PLqt2rd0eew1bmDn54E2_M2uvbhm_WxY_6&index=36) with Dynatrace. Once you have some Dynatrace PurePaths collected share them with me through my [Share Your PurePath](http://bit.ly/sharepurepath) program.

So – who is up for the challenge? First come – first win!

**Monolithic Servers vs Microservices**

[9 Replies](https://technologyconversations.com/2015/01/07/monolithic-servers-vs-microservices/#comments)

**Introduction**

At the beginning applications were simple and small due to simple requirements. With time requirements and needs grew and with them our applications became bigger and more complex. That resulted in monolithic servers developed and deployed as a single unit. Microservices are, in a way, return to basics with simple applications that are fulfilling today’s needs for complexity by working together through utilization of each others APIs.

**What are monolithic servers?**

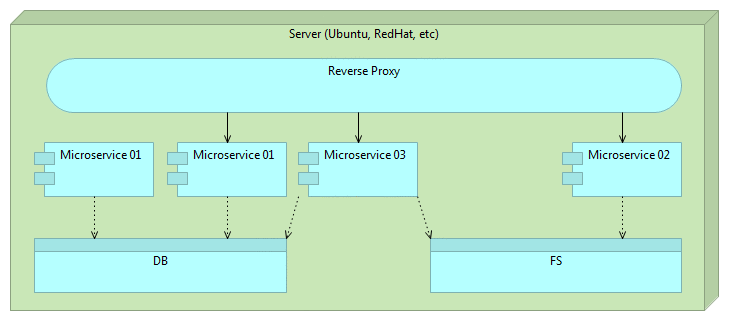
[](https://technologyconversations.files.wordpress.com/2015/01/large_5252421562.jpg)Microservices are best explained when compared with their opposite; **monolithic servers**. They are **developed and deployed as a single unit**. In case of Java, the end result is often a single WAR or JAR file. Same is true for C++, .Net, Scala and many other programming languages.

Most of the short history of software development is marked by continuous increment in sizes of applications we develop. As time passes we’re adding more and more to our applications continuously **increasing their complexity and size** and **decreasing our development, testing and deployment speed**.

With time we started dividing our applications into layers: presentation layer, business layer, data access layer, etc. This separation is more logical than physical. While development got a bit easier we still needed to test and deploy everything every time there was a change or a release. It is not uncommon in enterprise environments to have applications that take hours to build and deploy. Testing, especially regression, tends to be a nightmare that in some cases lasts for months. As time passes, our ability to make changes that affect only one module is decreasing. The main objective of layers is to make them in a way that they can be easily replaced or upgraded. That promise was never really fulfilled. Replacing something in big monolithic applications is almost never easy and without risks.

Scaling such servers means scaling the entire application producing very unbalanced utilization of resources. If we need more resources we are forced to duplicate everything on a new server even if a bottleneck is one module.

**What are microservices?**

[](https://technologyconversations.files.wordpress.com/2014/11/deployment_03_immutable_microservices_05.gif)  
Microservices are an approach to architecture and development of a **single application composed of small services**. The key to understanding microservices is their **independence**. Each is developed, tested and deployed separately from each other. Each service runs as a separate process. The only relation between different microservices is data exchange accomplished through APIs they’re exposing. They inherit, in a way, the idea of **small programs and pipes** used in Unix/Linux. Most Linux programs are small and produce some output. That output can be passed as input to other programs. When chained, those programs can perform very complex operations. It is complexity born from combination of many simple units.

Key aspects of microservices are:

* They do one thing or are responsible for one functionality.
* Each microservice can be built by any set of tools or languages since each is independent from others.
* They are truly loosely coupled since each microservice is physically separated from others.
* Relative independence between different teams developing different microservices (assuming that APIs they expose are defined in advance).
* Easier testing and [continuous delivery or deployment](https://technologyconversations.com/2014/11/27/continuous-deployment-introduction/)

One of the problems with microservices is the decision when to use them. At the beginning, while application is still small, problems that microservices are trying to solve do not exist. However, once the application grows and the case for microservices can be made, the cost of switching to a different architecture style might be too big. Experienced teams might use microservices from the very start knowing that technical debt they might have to pay later will be more expensive than working with microservices from the very beginning. Often, as it was the case with Netflix, eBay and Amazon, monolithic applications start evolving towards microservices gradually. New modules are developed as microservices and integrated with the rest of the system. Once they prove their worth, parts of the existing monolithic application get refactored into microservices.

One of the things that often gets most critique from developers of enterprise applications is decentralization of data storage. While microservices can work (with few adjustments) using centralized data storage, option to decentralize that part as well should, at least, be explored. The option to store data related to some service in a separate (decentralized) storage and pack it all together in the same container is something that in many cases could be a better option than storing that data in a centralized database. We’re not proposing to always use decentralized storage but to have that option in account when designing microservices.

**Disadvantages**

**Increased operational and deployment complexity**

Major argument against microservices is increased operational and deployment complexity. This argument is true but thanks to relatively new tools it can be mitigated. **Configuration Management (CM)** tools can handle environment setups and deployments with relative ease. Utilization of **containers** with [**Docker**](https://www.docker.com/) greatly reduces deployment pains that microservices can cause. CM tools together with Docker allow us to deploy and scale microservices easily. An example can be found in the article [Continuous Deployment: Implementation with Ansible and Docker](https://technologyconversations.com/2014/12/29/continuous-deployment-implementation-with-ansible-and-docker/).

In my opinion, increased deployment complexity argument usually does not take in account advances we saw during last years and is greatly exaggerated. That does not mean that part of the work is not **shifted from development to DevOps**. It definitely is. However, benefits are in many cases bigger than inconvenience that shift produces.

**Remote process calls**

Another counter argument is reduced performance produced by remote process calls. Internal calls through classes and methods are faster and this problem cannot be removed. How much that loss of performance affects a system depends from case to case. Important factor is how we split our system. If we take it towards the extreme with very small microservices (some propose that they should not have more than 10-100 LOC) this impact might be considerable. I like to create microservices organized around functionality like users, shopping cart, products, etc. This reduces the amount of remote process calls. Also, it’s important to note that if calls from one microservice to another are going through fast internal LAN, negative impact is relatively small.

**Advantages**

Following are only few advantages that microservices can bring. That does not mean that the same advantages do not exist in other types of architecture but that with microservices they might be a bit more prominent that with some other options.

**Scaling**

Scaling microservices is much easier than monolithic applications. While in the later case we duplicate the whole application to a new machine, with microservices we **duplicate only those that need scaling**. Not only that we can scale what needs to be scaled but we can distribute things better. We can, for example, put a service that has heavy utilization of CPU together with another one that uses a lot of RAM while moving a second CPU demanding service to a different hardware.

**Innovation**

Monolithic servers, once initial architecture is made, do not leave much space for innovation. Due to their nature, changing things takes time and experimentation is very risky since it potentially affects everything. One cannot, for example, change Apache Tomcat for NodeJS just because it would better suit one particular module.

I’m not suggesting that we should change programming language, server, persistence, etc for each module. However, monolithic servers tend to go to an opposite extreme where changes are risky if not unwelcome. With microservices we can choose what we think is the **best solution for each service** separately. One might use Apache Tomcat while the other would use NodeJS. One can be written in Java and the other in Scala. I’m not advocating that each service is different from the rest but that each can be made in a way that we think is best suited for the goal at hand. On top of that, changes and experiments are much easier to do. After all, whatever we do affects only one out of many microservices and not the system as a whole as long as the API is respected.

**Size**

Since microservices are small they are much easier to understand. There is much **less code to go through** in order to see what one microservice is doing. That in itself greatly simplifies development especially when newcomers join the project. On top of that, everything else tends to be much faster. **IDEs work faster** with a small project when compared to big ones used in monolithic applications. They **start faster** since there are no huge servers or a huge number of libraries to load.

**Deployment, rollback and fault isolation**

**Deployment is much faster and easier**. Deploying something small is always faster (if not easier) than deploying something big. In case we realized that there is a problem, that problem has potentially limited affect and can be **rolled back much easier**. Until we rollback, **fault is isolated** to a small part of the system. [**Continuous Delivery or Deployment**](https://technologyconversations.com/2014/12/03/continuous-deployment-strategies/) can be done with speed and frequencies that would not be possible with big servers.

**No need for long term commitment**

One of the common problems with monolithic applications is commitment. We are often forced to choose from the start architecture and technologies that will last for a long time. After all, we’re building something big that should last for a long time. With microservices that **need for a long-term commitment is not so big**. Change the programming language in one microservice and if it turns out to be a good choice, apply it to others. If the experiment failed or is not optimum, there’s only one small part of the system that needs to be redone. Same applies to frameworks, libraries, servers, etc. We can even use different databases. If some lightweight NoSQL seems like the best fit for a particular microservice, why not use it and pack it inside the container?

**Best practices**

Most of the following best practices can be applied to services oriented architecture in general. However, with microservices they become even more important or beneficial.

**Containers**

Dealing with many microservices can easily become a very complex endeavor. Each can be written in a different programming language, can require a different (hopefully light) server or can use a different set of libraries. If each service is packed as a container most of those problems will go away. All we have to do is run the container with, for example, Docker and trust that everything needed is inside it.

**Proxy microservices or API gateway**

Big enterprise front-ends might need to invoke tens or even hundreds of HTTP requests (as is the case with [Amazon.com](http://www.amazon.com/)). Requests often take more time to be invoked than to receive response data. Proxy microservices might help in that case. Their goal is to invoke different microservices and return an aggregated service. They should not contain any logic but simply group several responses together and respond with aggregated data to the consumer.

**Reverse proxy**

Never expose microservice API directly. If there isn’t some type of orchestration, dependency between the consumer and microservices becomes so big that it might remove freedom that microservices are supposed to give us. Lightweight servers like [nginx](http://nginx.org/) and [Apache Tomcat](http://tomcat.apache.org/) are very good at performing reverse proxy tasks and can easily be employed with very little overhead. Please consult [Continuous Deployment: Implementation](https://technologyconversations.com/2014/12/08/continuous-deployment-implementation/) article for one possible way to use reverse proxy with Docker and few other tools.

**Minimalist approach**

Microservices should contain only packages, libraries and frameworks that they really need. The smaller they are, the better. This is quite in contrast with the approach used with monolithic applications. While previously we might have used JEE servers like JBoss that packed all the tools that we might or might not need, microservices work best with much more minimalistic solutions. Having hundreds of microservices with each of them having a full JBoss server becomes overkill. [Apache Tomcat](http://tomcat.apache.org/), for example, is a much better option. I tend to go for even smaller solutions with, for example, [Spray](http://spray.io/) as a very lightweight RESTful API server. Don’t pack what you don’t need.

Same approach should be applied to OS level as well. If we’re deploying microservices as Docker containers, [CoreOS](https://coreos.com/) might be a better solution than, for example, Red Hat or Ubuntu. It’s free from things we do not need allowing us to better utilize resources.

**Configuration management is a must**

As the number of microservices grows, the need for Configuration Management (CM) increases. Deploying many microservices without tools like [Puppet](http://puppetlabs.com/), [Chef](https://www.chef.io/) or [Ansible](http://www.ansible.com/) (just to name few) quickly becomes a nightmare. Actually, not using CM tools for any but simplest solutions is a waste with or without microservices.

**Cross functional teams**

While there is no rule that dictates what kinds of teams are utilized, microservices are done best when the team working on one is multifunctional. A single team should be responsible for it from the start (design) until the finish (deployment and maintenance). They are too small to be handled from one team to another (architecture/design, development, testing, deployment and maintenance teams). Preference is to have a team that is in charge of the full lifecycle of a microservice. In many cases one team might be in charge of multiple microservices but multiple teams should not be in charge of one.

**API versioning**

Versioning should be applied to any API and this holds true for microservices as well. If some change will brake the API format, that change should be released as a separate version. In case of public APIs or microservices, we cannot be sure who is using them and, therefore, must maintain backward compatibility or, at least, give consumers enough time to adapt. There is a section on API versioning published in the [REST API with JSON](https://technologyconversations.com/2014/08/12/rest-api-with-json/) article.

**Summary**

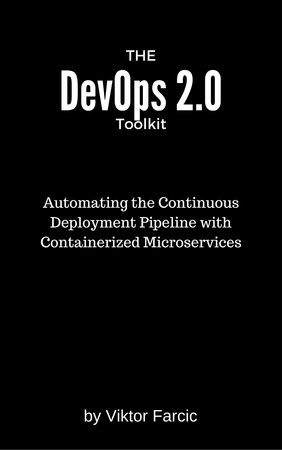
Microservices are not an answer to all our problems. Nothing is. They are not the way all applications should be created. There is no single solution that fits all cases.

Microservices exist for a long time and recent years are seeing increase in their popularity. There are many factors that lead to this trend with scalability being probably the most important one. Emergence of new tools, especially Docker, are allowing us to see microservices in a new light and remove part of the problems their development and deployment was creating. Utilization of microservices by “big guys” like Amazon, NetFlix, eBay, and others, provides enough confidence that this architectural style is ready to be evaluated (if not used) by developers of enterprise applications.

For more information about microservices, please consult [Microservices Development with Scala, Spray, MongoDB, Docker and Ansible](https://technologyconversations.com/2015/01/26/microservices-development-with-scala-spray-mongodb-docker-and-ansible/)

photo credit: [zamito44](https://www.flickr.com/photos/zamito44/5252421562/) via [photopin](http://photopin.com) [cc](http://creativecommons.org/licenses/by-nc/2.0/)

**The DevOps 2.0 Toolkit**

[](https://leanpub.com/the-devops-2-toolkit)If you liked this article, you might be interested in [The DevOps 2.0 Toolkit: Automating the Continuous Deployment Pipeline with Containerized Microservices](https://leanpub.com/the-devops-2-toolkit) book.

This book is about different techniques that help us architect software in a better and more efficient way with *microservices* packed as *immutable containers*, *tested* and *deployed continuously* to servers that are *automatically provisioned* with *configuration management* tools. It’s about fast, reliable and continuous deployments with *zero-downtime* and ability to *roll-back*. It’s about *scaling* to any number of servers, design of *self-healing systems* capable of recuperation from both hardware and software failures and about *centralized logging and monitoring* of the cluster.

In other words, this book envelops the whole *microservices development and deployment lifecycle* using some of the latest and greatest practices and tools. We’ll use *Docker, Kubernetes, Ansible, Ubuntu, Docker Swarm and Docker Compose, Consul, etcd, Registrator, confd, Jenkins*, and so on. We’ll go through many practices and, even more, tools.