**Microservices Notes**

April 18, 2016

References:

* <http://kasunpanorama.blogspot.com/2015/11/microservices-in-practice.html>
* <http://kasunpanorama.blogspot.com/2016/02/microservices-enterprise-integration.html>
* <http://microservices.io/patterns/microservices.html>
* <http://martinfowler.com/articles/microservices.html>
* <http://techblog.netflix.com/2011/12/making-netflix-api-more-resilient.html>
* <http://www.infoq.com/articles/seven-uservices-antipatterns>
* <https://pragprog.com/book/mnee/release-it>
* <http://nordicapis.com/how-to-control-user-identity-within-microservices/>
* <http://blogs.gartner.com/gary-olliffe/2015/01/30/microservices-guts-on-the-outside/>
* <http://blog.arkency.com/2014/07/microservices-72-resources/>
* <https://developers.redhat.com/blog/2017/05/04/the-truth-about-microservices/>
* <http://www.apiacademy.co/designing-a-system-of-microservices/> with example

**Definition of Microservices**

“..an approach to developing a single application as a suite of small services..”

The *micro-services* are:

“…running its own process…”

“…built around business capabilities…”

“…independently deployable…”

“…communicate with lightweight mechanisms...”

“…may be written in different programming languages and use different storage technologies…”

- *James Lewis, Martin Fowler*, [www.martinfowler.com](http://www.martinfowler.com/)

*Notes from “How To Capture The Benefits Of Microservice Design”, Forrester report by Jeffrey S Hammond, John R Rymer, May 26, 2016*

**Benefits of Microservices**

* Implementation flexibility
* Graceful scaling
* Faster delivery
* Higher resiliency
* Speeding app delivery
* Enabling quick, incremental app changes
* Ensuring resiliency at scale
* Optimizing production costs
* Supporting graceful scaling
* Supporting polyglot architecture

**Challenges for Enterprises**

* Microservices create a dependency-management challenge
* CI and CD are difficult to scale: These practices cannot mature overnight
* Microservice composition requires new API and messaging technologies
* Internet-scale application on public cloud platforms require new platforms

**Micro-services Migration path**

* Downsize deployment units
* Suvey your developers to understand how current applications might adapt (survey apps for microservices fitment)
* Find new architectural balance: Best practices, guidelines and tools
* Fit new workloads to best programming patterns
* Decide explicitly on build versus buy for platform services

Self-notes from the paper

* New programming models for Micro-services
  + Functional pipeline (Pipes & Filter pattern)
    - Trigger code on demand by services, HTTP endpoints or in-app activity
    - E.g. AWS Lambda
    - Fit for for real-time stream processing; ETL; file processing; experimental apps with unpredictable, variable, or low infrastructure demands; and as an infrastructure services layer for mobile apps.
  + Actor model
    - In response to a message that it receives, an actor can: make local decisions, create more actors, send more messages, and determine how to respond to the next message received.
    - Actors do not share state, implement concurrency via asynchronous message passing
    - Lends itself as a straightforward way to map an entity model
    - Fit for high-scaling transactional systems or as the server-side representation of connected IoT devices.
* New platforms for Microservices

The new platforms employ containers as their native service-deployment unit and remove control and management functions from application servers so they may apply to all microservices — whichever language or runtime they use. These platforms:

* + Support self-contained, independent modules
  + Provide Control services, which
    - Provide the dynamic deployment and provisioning services
    - Provide the communications protocols for microservice interactions within the same node or cluster as well as interactions with external services using APIs that enable composition,
    - Provide means to managing access and usage limits
    - Provide state management and other app services
    - Collect signals and data used by the platform’s management plane.
  + Provide Management services
    - Monitoring, logging, configuration
    - Identity resolution and establishment of trust boundaries

**Microservices Architecture Goals**

Here are twelve goals that we are trying to achieve with microservices:

1. Independent deployment of components
2. Independent scaling of components
3. Independent implementation stacks for each component
4. Easy self-serve deployments of components
5. Repeatable deployments of components (external configuration management)
6. Deployments without service interruptions
7. Protection of system availability from individual Instance failure
8. Automatic replacement of component instances when they fail (self-healing)
9. Easy scaling of components by adjusting a simple parameter value
10. Canary testing
11. "Red/black" or "blue/green" deployments
12. Instant reversal of new revision deployments

**Microservices Architecture**

* Inner Architecture
  + Service boundaries
  + Data management
  + Transaction support
  + Deployment
    - While the implementation for microservices can leverage any technology stack, it is important that the units of deployment for micro-services are consistent
* Outer Architecture
  + Service discovery
  + Service communication
  + Orchestration
  + Scalability
  + Security
  + SDLC
  + Deployment

Microservices Concerns

* Service design
  + Service boundaries
    - Basic rule: business capability encapsulation
    - However, this is too broad a rule. There are other things to consider to find good candidates for microservices
      * Outside of user response loop: Very immediate real-time activities vs asynchronous jobs/processing
      * 3rd party service API calls: great place to start.
      * Long running processes: e.g. encoding of video
      * Transaction processing
      * Scale-out/burst processing
      * Scheduled jobs
  + Design for failure
  + Forward compatible
  + Useful patterns
    - Tolerant reader <http://martinfowler.com/bliki/TolerantReader.html>
    - Consumer Driven contract <http://martinfowler.com/articles/consumerDrivenContracts.html>
* Service communication
  + Communication protocol
    - Synchronous calls:
      * REST
      * Thrift (Apache Thrift)
        + Thrift is a software library and set of code-generation tools developed at Facebook to expedite development and implementation of efficient and scalable backend services. Its primary goal is to enable efficient and reliable communication across programming languages by abstracting the portions of each language that tend to require the most customization into a common library that is implemented in each language. Specifically, Thrift allows developers to define datatypes and service interfaces in a single language-neutral file and generate all the necessary code to build RPC clients and servers.
    - Asynchronous calls
      * AMQP, STOMP, MQTT
  + Communication format
    - Text based: JSON, XML
    - Binary: Thrift, ProtoBuf, Avro
  + Inter-service communication
    - Should it be allowed?
      * Yes
      * No – use orchestration/fascade pattern instead
  + Orchestration
    - Orchestration pattern also deployed as microservice
    - To support scalability, gateway only for routing/NFR fulfilment
* Data management
  + Implementing checks & balances
  + Sharding for scale <http://soa.sys-con.com/node/3347703>
  + Eventually consistent
* Transaction support/ orchestration
  + Should avoid distributed transaction as far as possible
  + Options if unavoidable
    - Combine micro-services
    - Compensating transactions
  + “Eventual consistency”
* Service discovery
* Scalability
  + Load balancing
  + Auto-scaling
  + CQRS
  + Sharding data
  + Asynch calls
* Security
  + Challenges
    - Should each services have their own security firewall?
    - How should identity be distributed between microservices and through the entire system?
  + Solution 1
    - 2 part solution
    - External security domain
      * Issue a by-reference token (like access token) which is completely transparent/meaningless to users outside the organization
      * The token can be issued by an Auth Server (OAuthV2)
    - Internal security domain (higher trust)
      * When the client reaches out with the token, at the firewall/entry point, by reference token is translated to a by-value token (JWT containing user information and other data)
      * The JWT token is now freely passed to microservices within the organization
      * JWT processing is lightweight and federated can be included in each micro-service
* Design for failure
  + Ability to detect failures quickly and, if possible, automatically restore service
  + Monitoring of both architectural elements (request throughput) and business relevant metrics (biz transactions)
* DevOps
  + CI – Automated deployments
  + Testing
    - Testing of microservices in a huge challenge
    - Challenges
      * During integration testing, which version of which microservices do you test?
      * How do you test forward/backward compatibility
  + Implementation/Rollout
* Monitoring
  + Logging
    - Has to happen at a central location - scalable/robust
    - Has to have transactional identity to trace calls across microservices
  + Monitoring
    - Need a single pane of glass for monitoring ALL microservices
    - Transaction id becomes important
    - Very important to be able to view performance of all microservices. One low performing microservice can bring everything to halt
* Governance
  + Decentralized - Support for multiple technologies
  + Focused on building reusable tools rather than writing standards
  + Encourage internal open source model – share useful, battle-tested code

**Getting Micro-services Ready**

* In Enterprise Architecture, it will be a combination of Microservices and conventional monolithic architecture of existing systems
* Infrastructure (Outer Architecture)
  + Security
  + Packaging, deployment & orchestration
  + CI
  + Logging
  + Monitoring

**Micro-services Challenges:**

* Which host machine(s) should each service run on?
* How do I avoid port conflicts for multiple instances of the same microservice running on a single machine?
* How does the application code find the (required) Guest Book service?
* How do we keep all of these service running?
* What happens if a host machine has trouble?
* Are the services healthy?
* How do we scale when load changes?
* Run this in another environment? QA, dev, another cloud, your servers?

Solution:

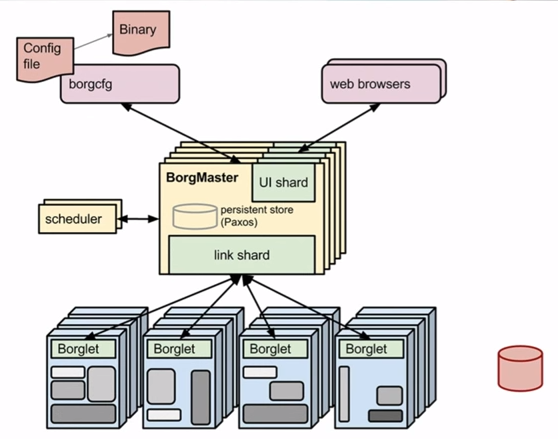
* Run microservices in containers
* Run containers in clusters with set resource caps

Here is how Google manages it using their Borg tool

* Defining the container parameters



* Borg architecture



* Steps
  + Put the binary to centralized repository
  + Take the config file defined above and push it to BorgMaster
  + BorgMaster will check against the scheduler as to which of these machines can I deploy to
  + Once the machine is identified, BorgMaster will ask the Borglet to download the binary and bring up the container/app

Google put this Borg architecture in open source Kubernetes

**Iron.io** – Meeting with Martin

* Job processing as a service
* Iron Worker – container within a vm that allows you use the infra more efficiently
* Closest competitor is AWS Lambda – do the same on any platform
* Components
  + Iron Worker
    - Spinoff docker and process and die off
  + Iron MQ
    - Queues up these tasks – provides the guarantees of FIFO
* Use-cases
  + Aviva – Speed bumps that go up and down based on the rate of traffic
    - They would take pictures of the traffic
    - Fast
  + They are being queued and batch processed on Iron
* Iron.io
  + IaaS
* Iron.io Licensing
  + Worker: Priced on concurrency and the size of workers (
  + MQs: Priced on # of nodes and the throughput

Notes from: <https://developers.redhat.com/blog/2017/05/04/the-truth-about-microservices/>

* Microservices are a method of breaking down an application into a suite of small, lightweight services, and are processes that typically communicate over HTTP. Building a single microservice is easy, building a microservice architecture is extremely hard.

**8 Challenges with Microservices**

Building Microservices

* How do you identify dependencies between services? A change to one can result in multiple service rebuilds.
* You must carefully track individual service versions and then determine what versions of what services constitute a release.

Testing

* Integration testing requires figuring out what actually needs to be tested, what are its neighbors?
* End-to-End testing requires standing up the entire microservice architecture with the right versions, but by its nature, these microservices are all versioned independently. It can be difficult to event determine what versions of other microservices you need to test.

Versioning

* Each service will have a unique, independent version.
* The collection of microservices will have a version.
* Maintaining a version matrix becomes very complex, especially maintaining this over time.
* How do you release and maintain backward compatibility? Do you maintain the old code and add the new features? This will lead to code bloat, which is contrary to the nature of microservices. Or do you release breaking changes, where you have to release multiple dependent versions all at the same time?

Deploying

* Deploying microservices requires strong automation, as the system is too complex to deploy by hand. This is a prerequisite to moving to microservices.
* Blue/Green deployments are a desirable deployment methodology for microservices. The system is too complex to try to back-out changes.

Note: If your app uses a relational database, blue-green deployment can lead to discrepancies between your Green and Blue databases during an update. To maximize data integrity, configure a single database for backward and forward compatibility.

Logging

* Microservices require centralized logging,
* You will also require some type of global request tracing, assigning a request ID to each request such that you can figure out how things flow.

Monitoring

* Microservices should only be deployed along with a centralized dashboard. Monitoring multiple microservices is a very difficult task, you need something to aggregate the service data.
* Your monitoring solution also requires distributed request tracing. Microservice architectures really require a mechanism to visualize a request ID across the services and measure performance. One slow microservice can slow down the application to a halt.

Debugging

* It is virtually impossible to tell where the failure is without the monitoring solution.
* Remote debugging is not feasible across many microservices at once. You need to be able to iterate the microservice versions very quickly, sometimes guessing to solve your problem.

Connectivity

* Microservice architectures should include a service discovery registry. You don’t want to hardcode URLs into your microservices, especially with moving between CI/CD environments. How do they contact each other? How do they contact the right version of each other?
* Your architecture needs to accommodate failures gracefully. Build in short TTLs, fail quickly to bubble up the failures rather than a slow crawl to death. Failing quickly lets you retry quickly.

**When should you consider moving to microservices?**

Traditional monolithic architectures have their own unique and well-understood challenges. If you are starting to see these block development efforts, microservices might be your answer.

Size

* The monolith application code base size has grown too large for local development. Can you even load all the code in your IDE at once?

Stack

* Your monolith will be a single technology stack rather than the right stack for the right purpose.
* The initial large jump from one stack to another is very difficult, subsequent jumps are much easier to make.

Failure

* If anything fails in a monolith, everything fails… it is all one system.
* Much larger surface area to attack and to be exposed to external conditions

Scaling

* Only really can scale a monolith vertically, you have to increase everything at the same time. This leads to excessive cost and resource consumption.
* A few large VMs are usually much more expensive than many small VMs or containers

Developer Productivity

* Developers cannot work independently, rather in fewer, and larger teams.
* In a single code-base, CI takes much longer occurs much less frequently. You typically have to wait for nightly or weekly builds to see if anything breaks.

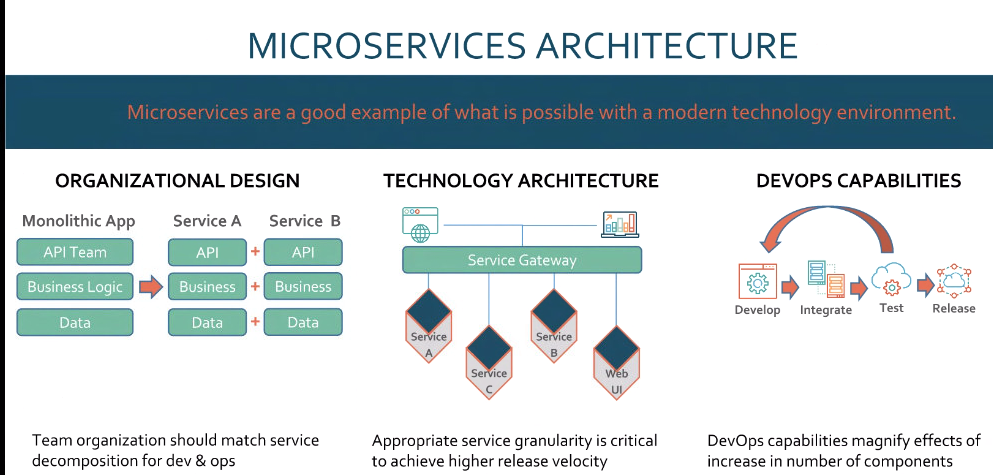
If you do decide to move to microservices, you might realize these benefits:

* Agility and flexibility.
* Smaller code-base, easier to wrap head around.
* More, smaller teams rather than one larger team.
* Easier to scale microservices, only scale the ones that are hot.
* Right stack for the right job, you are no longer bound to one stack.

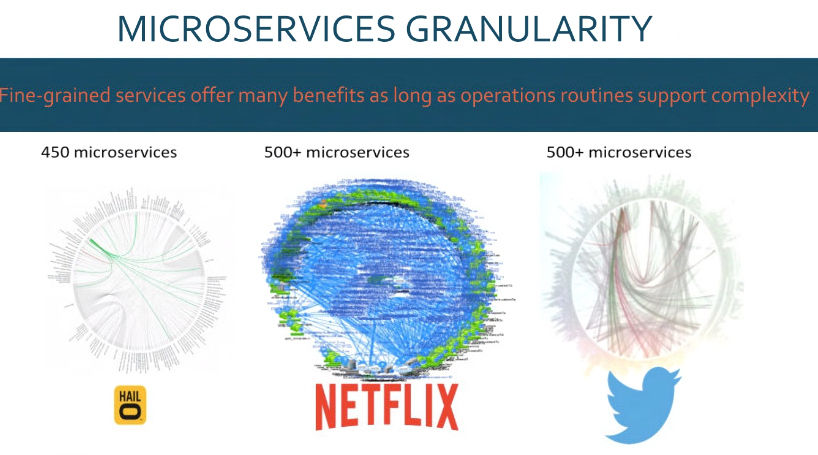
Start out from day one with automation. Trying to add it later will ultimately make your effort fail, you need to invest in the automation before starting to build.

Finally, remember Conway’s law – moving to microservices will likely require an “Agile Transformation”. Your organization should mirror the microservice architecture.

RedHat presentation to NLG



* Recommendations
  + Rather than building microservices from scratch, use existing applications to decompose into microservices
  + Have API and DevOps story tightened up



* Big architectural questions
  + How fine grained should their microservices be? How many microservices?

