Paremus Blogs

Microservices, Platforms & OSGi?

This entry was posted in Uncategorized *on March 16, 2015 by Richard Nicholson*

The concept of a 'Service' is hardly new. In the late 1990's Service Oriented Architecture enabled large monolithic business systems to be decomposed into a number of smaller loosely coupled business components.

Modularity all the way Down

The purpose of any 'Service' strategy should be to break large monolithic entities into groups of smaller interacting components; i.e. modular systems. The interaction between the components in a modular system is defined by some form of 'Contract', 'Service Agreement' or 'Promise': the nature of which dictates the interaction model between the components; i.e. the 'architecture'.

Relative to their monolithic counterparts, well designed modular systems are by their nature significantly simpler to change and maintain. Benefits include:

- Increased Agility – A subset of the components used in the composite system may be rapidly changed in-order to meet new, previously unforeseen business requirements or opportunities.
- Reduced Maintenance Costs – As long as the contracts between components remain unchanged, the internal implementation of each component can be independently refactored and maintained. The ability to cost effectively maintain the composite system avoids the accrual of technical debt.

Microservices simply continue this modularity trend: i.e. the process of decomposition by breaking business components into a number of finer grained functional components.

The justification is again the same:

1. To build more scalable, robust and maintainable systems.
2. To simplify development by assembling composite systems from a number of small single function software components: these simpler to develop in-house, or where appropriate, sourced from third parties.

However the logic that argues that business services should be composed of business components, and that business components should be composed of simpler single functional microservices, also applies to the internal implementation of EACH microservice.

If a microservice is to be maintainable, the internal implementation must be modular.

Microservices: It's not an Architecture

Modularity concepts are fundamental and will underpin any successful IT strategy. Yet modularity is frequently misunderstood.

One common mistake is to confuse general modularity principles with architectural approaches (currently fashionable or otherwise), issues encountered with vendor implementations, or ill-conceived industry standards. As explained by Kirk Knoernschild, structural modularity and architectural patterns are actually **orthogonal** concerns.

In the late 1990's Service Oriented Architecture enabled large monolithic business systems to be decomposed into a number of smaller, but still coarse grained, loosely coupled business components. However, outside the area of Business to Business (B2B), the original implementations – i.e. WS-* protocols and UDDI Directories – are now widely seen as a mistake: rather REST and messaging protocols (either directly or indirectly via a message broker) are the current popular approaches.

Yet looking behind these architectural differences, one can see that modularity principles have been successfully adopted by each approach. Indeed, more so than the advent and influence of the virtual machine, the application of modularity through generic SOA principles is directly responsible for the increasing dominance of today's commercial Web and Cloud based Services.

No Free Lunch

Being built from a number of simple functional units, a microservices based business application is, in principle, simpler to create, maintain and change.

Yet, as noted by Senior Gartner Analyst Gary Oliffee (http://blogs.gartner.com/gary-olliffe/2015/01/30/microservices-guts-on-the-outside/.), microservices are not a zero cost option. Gary describes microservices from two perspectives:

1. Internal Structure: Usually a single function service (hence the term microservice) which – in principle – is simple to develop. Also, as the communication mechanism is usually embedded, a microservice is easy to unit test as heavy weight applications servers are not required.
2. External Structure: This refers to the new platform capabilities that are **now needed** to help manage the interdependencies, life-cycle and configurations between the myriad of microservices. Whereas the unit test was simple, the integration testing of the complete solution requires the deployment and configuration of all these inter-related components.

To conclude, the 'observable' composite system is now significant more complex than the monolithic application it replaced.

The ideal microservices platform?

The purpose of a microservices platform is to shield this runtime complexity from Operations: to automate the discovery, configuration, life-cycle management and governance of a possibly changing set of interdependent runtime entities.

What are the fundamental attributes of an ideal microservices platform? Unfortunately there is no one simple answer, as it depends on *context*.

All businesses will value a platform's ability to abstract and shield hosted microservices from the underlying compute resource used. However, whereas a business providing vanilla hosted websites will have very simple application requirements, a business, comprised of many business units, each potential involved in different markets, may have extremely diverse requirements from a common platform.

For the latter group the platform solution must allow for *Architectural Agility* – meaning the platform solution must not constrain either:

1. The internal structure of the functional components.

Finally, given that the composite application cannot now function without the microservices platform; the platform itself must be engineered to new levels of robustness and agility and must be evolvable: the platform itself must be extremely modular.

Current Industry Fashions

It is my opinion that the current generation of popular '*microservice platform*' offerings fall well short of these objectives. The reason why is easy to understand.

Mainstream vendors pursue 'low hanging fruit' by focusing on enabling developers to quickly and easily assemble simple 'microservice' based applications. For example, the deployment of simple three Tier Web based applications built upon popular RESTful architectural patterns.

Typically:

- Opaque software artifacts are deployed via a light weight container.
- The container provides isolation in multi-tenancy environments.
- The services are usually simple REST based services.
- The platform, which itself is not dissimilar from the previous generation of Grid Compute solutions, provides some level of deployment, discovery and configuration of these simple services.

Yet while providing instant gratification, these same platforms fail to provide sufficient flexibility for more complex applications or diverse business needs:

1. The platform solution may or may not be transparent to the deployed applications: some platforms enforcing rigid restrictions on inter-container communication.
2. The platform may fail to adequately address the scoping and versioning of (i.e. interaction between) these hosted microservices.
3. The platform may only support a subset of interaction patterns or middleware options.

If one finds oneself 'force fitting' a broad set of business applications to a limited set of architectural patterns provided by the microservices platform – then the platform is most probably an inappropriate choice for your organization. More importantly, the platform will continue to remain an inappropriate choice – a point of constriction, reducing business agility without delivering the long term cost saving benefits provided via a *modularity first strategy.*

Microservices and OSGi

OSGi began in the late 1990's as the open industry standard for enforcing structural modularity for Java code running within a JVM. OSGi bundles enforce strong isolation:

- The internal implementation is private to each bundle,
- The behaviour exposed by the bundle is described by its stated '*Capabilities*',
- The dependencies a bundle has on its local environment are described by its stated '*Requirements*'.
- Finally semantic versioning is used. A bundle's *Capabilities* are versioned (*major.minor.micro*). Meanwhile a bundle's *Requirements* specify the acceptable version ranges within which the *Capabilities* of third parties must fall.

Due to strong isolation OSGi bundles may be dynamically loaded or unloaded from a running JVM. As the bundles are self-describing, a process know as 'resolution' may be used to ensure that all inter-related bundles are automatically loaded into the runtime and wired together.

These aspects of OSGi all relate to structural modularity and the concepts are quite generic. Self-describing semantically versioned artifacts are important concepts at all layers of the structural hierarchy.

In an orthogonal decision OSGi also decouples the interaction between bundles via a local Service Registry. In so doing the OSGi Alliance created an extremely powerful microservices architecture for Java. Due to OSGi's modularity first mindset, OSGi's service architecture is extremely powerful and evolvable with advertised Service Contracts representing:

- Synchronous or asynchronous remote procedure calls – with a choice of language specific or agnostic serialization mechanisms.
- Event based interactions.
- Message based interaction.
- Actor style interactions.
- Or, RESTful based interactions.

Where appropriate, pluggable discovery and serialisation mechanisms are supported.

But OSGi is difficult?

Not anymore.

Well designed modular systems do require some thought. However the OSGi Alliance is actively making OSGi simpler to adopt via ongoing investment in tooling (see http://bndtools.org) and investment in tutorials for typical application patterns. For example, the OSGi enRoute project demonstrates the creation of a simple OSGi based modular application. For such requirements the enRoute tutorial demonstrates that OSGi can be as easily to use as Spring Boot or Dropwizard. Additional OSGi enRoute tutorials are planned which will address other common architectural patterns used in IoT and the Enterprise. The implementation of sophisticated business systems in a modular manner does require an enhanced level of engineering and architecture skills. However the OSGi Alliance again provide support to achieve this in-terms of OSGi Alliance Member training (e.g. Paremus OSGi training) and the new OSGi Developer Certification programme.

To conclude, OSGi provides the basis for a compelling microservices strategy. However unlike the alternatives, this is only part of a larger coherent strategy. OSGi provides the necessary open industry standards upon which the next generation of modular, and so highly maintainable, software systems will be built.

Share This:

Leave a comment

Your email address will not be published. Required fields are marked *

Comment

Name *

Email *

Website

Post Comment

Post navigation

← It has been a while… Announcing the bnd Maven Plugin →

Follow Us

Recent Posts

- March of the IDEs — IDE Agnostic OSGi Development
- Bndtools and Maven: A Brave New World
- `OSGi Enabled` – This is not a statement, this is an ongoing commitment to you…
- Java 9, OSGi and the Future of Modularity
- Using Let's Encrypt Certificates with OSGi HTTP Service

Tweets

Mike Francis @service_fabric

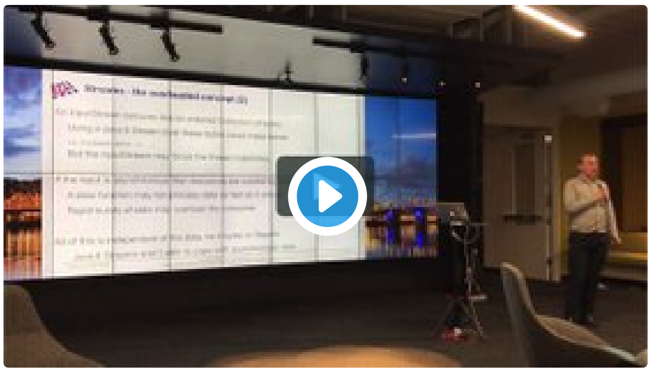
Great to see @TimothyWard presenting this. The live demo using British Rail real time train info (just over 20 mins in) is excellent.

Feb 1, 2017

PDX Java User Group @pjug

OSGi PushStreams is a streaming API that provides high thruput and flow control. Hear more from @TimothyWard #java vimeo.com/201982439

Vimeo @Vimeo



Embed View on Twitter