OSGi as an IoT platform
We are a Strategic Member and a member of the OSGi Alliance board

We have been involved since 2006

Our staff are active and influential participants:

– Tim Ward (Paremus CTO) - Chair of the OSGi Alliance IoT Expert Group.
– Mike Francis (Paremus Sales & Marketing Director) - OSGi Alliance VP of Marketing & member of the Liaison Committee.
– Richard Nicholson (Paremus CEO/Founder) - OSGi Alliance President (from 2011-2013), currently Treasurer and responsible for Strategy.

Current OSGi Alliance board members include:
IoT - The Challenge
Behaviours run local to the physical environments they control. Local environmental events may affect these local behaviours.

Processes must be adaptively controlled within the environment within which they work, and not some ‘Cloud Data Centre’ hundreds or thousands of miles away! (A heat sensor should not need the internet to trigger the fire alarm).

Software updates must be small, rapid, reversible and optionally hot-swappable. As IoT is not homogenous - the local environment will affect the selection of some of the components installed.
Privacy & Data

Processing data locally preserves context, privacy and avoids uploading large amounts of data.

Example:
A body scan generates many GB of data. Uploading this for remote processing is time consuming and costly. Processing via local Machine Learning algorithms is quick and cost effective. Analysis of data may also require additional context - Why was the patient scanned? What is the Patient's medical history? This full context is available to the Doctor / Hospital which generates the data - so this is where the analysis should occur.

Anonymised / derivative / results may be propagated to up stream Services.
The providence of each software component must be assured.

Interdependence between the atomic software components in the assembly must be explicit.

This is true whether the software component is deployed on-Premise, at the Access or Aggregation layers, or in the Cloud Core.

This is difficult to achieve if the deployment artefact is an opaque image!

Iran’s Uranium enrichment centrifuges destroyed by *stuxent* software virus.

http://www.wired.com/2014/11/countdown-to-zero-day-stuxnet/
IoT Scale & Complexity: It is only just beginning…

Traditional engineering:
Building a re-usable Spacecraft.
100s of Engineers

Unicorn engineering:
An application for hailing a Cab.
1,000s of Engineers!

The approach for IoT
A Federated, Hierarchical & Edge centric environment
? of Engineers!
Adaption, Evolution & Longevity

“Modern-day software systems, even those that presumably function correctly, have a useful and effective shelf life orders of magnitude less than other engineering artifacts. While an application's lifetime typically cannot be predicted with any degree of accuracy, it is likely to be strongly inversely correlated with the rate and magnitude of change of the ecosystem in which it executes.”

April 2015

Evolvability means that the system doesn’t have to be restarted or redeployed in order to adapt to change.

Roy T. Fielding is a Senior Principal Scientist at Adobe

Adaptive / Evolvable Software → Modularity
Why Modularity?

**Modularity** makes complexity **manageable**.

Modularity masks necessary internal implementation from the external environment. Modularity allows accidental complexity to be removed.

While conceptually simple - Modularity requires coherent approaches to:

- Defining boundaries - i.e. identifying / defining the module.
- Characterising exposed properties - i.e. describing the module.
- *The management of the resultant exposed dependencies.*
- Accommodate different perspectives.

Modular principles are hierarchical and must be applied to each orthogonal set of **Aspects / Attributes** that describe the System under consideration.

<table>
<thead>
<tr>
<th>Aspect</th>
<th>Structural Layers</th>
</tr>
</thead>
<tbody>
<tr>
<td>Physical Resource</td>
<td>Regional, Data Centre, Physical Compute, Container.</td>
</tr>
<tr>
<td>Software Artefact</td>
<td>System, Container Image, JAR's / bundles, Classes.</td>
</tr>
<tr>
<td>Service / API</td>
<td>Business Services, Microservices, µServices.</td>
</tr>
<tr>
<td>Processes</td>
<td>Software and Organisational. (Question: <em>Is DevOps modular?</em>)</td>
</tr>
</tbody>
</table>

---


What is OSGi?

- A **modular runtime** for the Java Virtual Machine:
  - Modules can be added to or removed from a running framework
  - Module dependencies are enforced at runtime
  - Remote management and monitoring capabilities

- First developed as a home automation platform
  - Lightweight implementations exist
  - Proven on a large variety of hardware configurations

- OSGi is an Open Standards body
  - Specifications are royalty free to implement
  - All API is published under the Apache 2 Licence
Why is OSGi helpful?

• Runtime heterogeneity
  • Not all systems offer the same capabilities (e.g. available memory)
  • Some systems are just totally different! (e.g. MIPS vs x86)
  • OSGi can transparently model these differences using Requirements and Capabilities

• Application complexity
  • As applications grow in size they become more complex and interlinked
  • Managing application dependencies becomes a full-time role
  • OSGi’s runtime module enforcement allows complexity to be managed and understood — important from a governance & security perspectives!
Why is OSGi helpful?

• Remote Systems
  • Remote management is built in to the OSGi standards
  • Management can be REST based, or using standard Framework APIs
  • Configuration management is also available, as well as code deployment

• Dynamic systems
  • The Dynamic nature of OSGi means that it can easily adapt to changes in its surroundings
  • Naturally address the hierarchical / locality requirements for IoT - Edge, Access, Aggregation & Core - ability to migrate functionality and flow processing between each layer.
“IoT systems are fundamentally different from the software systems that we build today. The systems are so large and have so many distributed parts that they will never all be available simultaneously. The only way for an IoT system to succeed is to design it so that it can continue to function despite a constantly changing set of catastrophic failures.”

IoT systems are constantly changing

- New devices are added all the time
- Existing devices break, are removed, or run out of power
- Network interruptions may partition large groups
- Evolving protocols and standards.

Embracing planned & unplanned changes is the only way to succeed
The OSGi Service Registry

Most OSGi Users focus on creating modules

- The real power of OSGi comes from dynamic services
- A lightweight, loosely-coupled way to share implementations
OSGi at the Edge?
OSGi for Edge Processing

• OSGi’s origins are as a home gateway platform
  • OSGi is still well suited to modest consumer hardware
  • Built to be “Always On”, so suitable for low interaction devices

• Modular Architecture makes it easy to upgrade/expand software
  • Paid modules/services can be added without needing to reimage the device
  • Different implementations can be used based on the underlying hardware.
OSGi for Edge Processing

- Security is a core part of the OSGi standards
  - Signing can be used to ensure a module’s safety and integrity
  - Fine-grained permissions can be used to limit the functions available to a module
  - Dynamic permission management is possible

- OSGi Edge devices provide an excellent “autonomic decision point”
  - Aggregation and filtering of data before forwarding on to a higher-level service
  - Fast “reflex” decisions can initiate local actions without higher level decision making
Flexible Mobile devices use “Apps” to provide modular functionality
   • Each app is self-contained
   • No sharing of dependencies

However, this model leads to significant bloat, and to security risks
   • Each app packages its own MQTT client
   • Pulling in a critical fix requires every app to be updated separately

OSGi allows dependencies to be shared based on version
   • Multiple incompatible versions can safely exist concurrently
   • Runtime linking ensures that only “safe” dependencies are used
Dynamic OSGi for IoT

- Dynamic services are the perfect way to represent connected devices
  - When discovered or activated then register a service
  - When disconnected or “lost” then unregister the service

- Standard injection frameworks allow components to be notified of the services
  - Components automatically react to new devices
  - Components automatically react to device “failures”
  - Components automatically recover when a device reconnects

- OSGi services are therefore the perfect way to describe IoT “things”
  - Components are abstracted from the discovery mechanism
  - No need for separate UPnP/mDNS/CoAP multicast handlers
OSGi in the Cloud/Core
OSGi in the Cloud

• Even though it was intended for the home, OSGi is great in the cloud
  • Home devices are constrained by hardware
  • Server devices are constrained by software complexity!

• The benefits of modularity are multiplied by the complexity of the application
  • Bigger applications are more likely to become monoliths

• OSGi services provide great abstractions for RPC
  • The call may be local, or “burst out” remotely for more capacity
  • Dynamic endpoints allow cloud servers to fail or move transparently
  • A new specification for Stream based Data Flows later this year.

• Standardised support for asynchronous calls
  • High-Performance event-based systems are easy to assemble
Addressing Scale & Data Gravity - the OSGi Way

Flow of 'Behaviours' & Site Configurations

Cloud Core                      Aggregation Layer              Access Layer             IoT Edge

Flow of derivative / anonymised data
OCF talked yesterday about a cloud of interconnected microservices. This is the right direction - but which implementations?

- 1990’s - Java / Jini - no.
- 2016 - REST Microservices / XMPP (OCF) - currently fashionable, but…

What if higher density / performance discovery is required?

What if MQTT, or Async RPC, or content streaming is required?

Is the XMPP “Things Repository” idea repeating the mistakes of UDDI? Who knows?
Evolvability - Example

Modular IoT Platforms shouldn’t care about plumbing!

Real Life Example - The Paremus Service Fabric:

• 2005 - Java RMI / Jini Discovery
• 2010 - OSGi Synchronous RPC / Discovery using DDS
• 2016 - OSGi Async / Synchronous RPC, REST, Streaming / Discovery Gossip

Migrating from Java / Jini to Distributed OSGi was hard. The remoting implementation / protocols were highly coupled into platform.

The latest change to the remoting and discovery stacks was trivially easy - no changes were made to any services!

So simple to re-configure to implement an OCF view of IoT Cloud or a oneM2M view of an IoT Cloud, or …, or …
Current IoT / EEG EG Activities

- Device Abstraction Layer update
  - Simplified Driver service registration
  - Simpler refinement of devices
- MQTT event handling abstraction
- CoAP resource discovery and access
- Push Streams & Distributed Eventing in progress with EEG.
- Security Best Practices
  - Security test suite

Future possibilities
- Software Defined Network management
  - Dynamically controlling Service QoS / Properties.
- Other ideas?
  - Offline store for batch upload?
OSGi & IoT Example Use Case

OSGi IoT Demo

Extendable, reusable demo produced by OSGi Alliance members
Being used at CeBIT March 2016
New extended version for OSGi Community Event Oct 2016

2016 OSGi IoT Contributors are currently:
Thank You

www.paremus.com

@Paremus

info@paremus.com