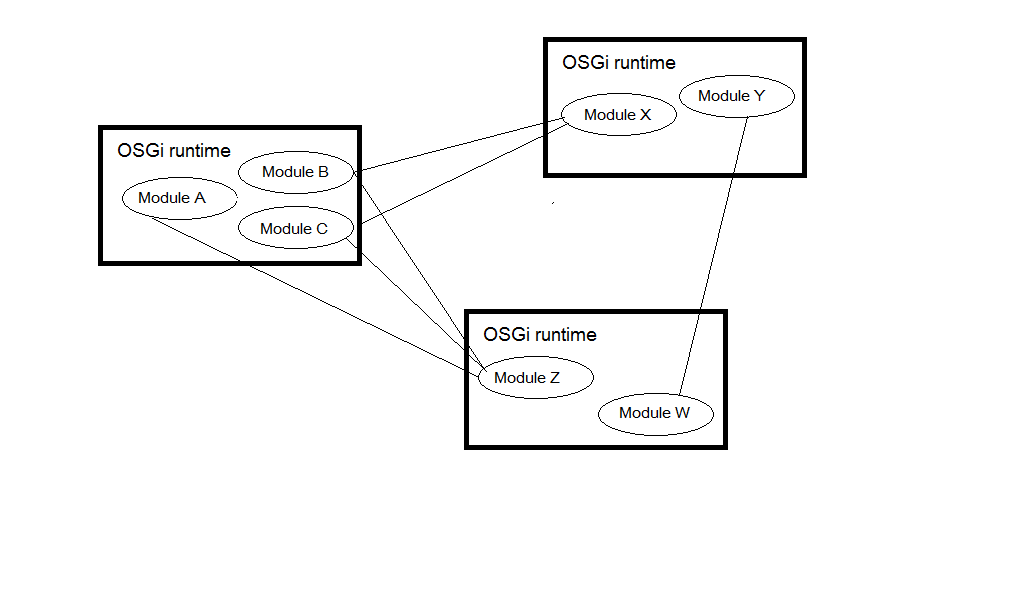
Building Microservices in OSGi with the Apache Karaf Framework

The microservice paradigm has been a topic of heavy discussion and interest in recent years. To make the matter even more involved developers and architects have different understanding on the core principles of microservices and the need to adopt a microservice-based architecture. Taking the decision to build a new system or modernize an existing on as a set of microservices brings further the question of choosing the right framework for the purpose. These could be the Netflix OSS, Spring Cloud, Eclipse Microprofile, Oracle’s Helidon framework, Wildfly Swarm or even Vert.x framework to name a few. Or if that needs to be a polyglot system one can also go with Kubernetes. Since the Spring and JavaEE frameworks provide mechanisms for adopting a microservice-based architecture why not have one build on top of OSGi as well? OSGi being the de-facto standard Java module system that brings Java applications the benefits of a highly dynamic module system might turn out to be a great candidate for building a microservice system. Even though many might argue that OSGi is an alternative to microservices considering that it splits a typical Java application into a set of modules the idea of using OSGi in combination with microservices is in fact not new and has been already successfully applied in software applications as a basic Google search on the **microservices** and **OSGi** keywords altogether is revealing.

We are bundled with the task of architecting a complex system. So complex that it is even required that the separate microservices be split into separate self-container modules internally that can evolve over time without having to bring down and the entire microservice and delivering a full new version of that microservice (bit just a version of the particular module being affected). Such a system would be represented at a very high level schematically by the following diagram:



Examples of such complex systems that might be implemented in that manner are:

* a complex factory process (i.e. for an automotive or airplain factory);
* an IoT system (where each device integration is an OSGi microservice split into modules that implement the different functions of the device);
* complex supply chain (i.e. flight cargo supply chain system).

In this article we will demonstrate how we can get the best of both worlds and build a microservice-based system on top of the Apache Karaf framework (using default Apache Felix OSGi runtime through Karaf) that addresses this scenario.

# Microservices from the OSGi perspective

One can find different definition of microservices but all of them boil down to the notion of an architectural style for development of loosely-coupled, independently deployable units of work that provide distinct business capabilities, can communicate with lightweight protocols and can be developed, tested and scaled independently. This implies that microservices relate directly to modularization of the system. In that sense we can think of OSGi as microservice-oriented framework running in a JVM environment. To be more precise lets reveal how do the different microservice concepts map to the OSGi world:

* configuration management – that is the OSGi config admin defined in the OSGi compendium specification;
* service discovery – that is the OSGi service registry defined in the OSGi core specification;
* dynamic routing – dynamic routing among service (in potentially different modules) can be established by means of OSGi filters which allow for the retrieval of a service reference using an LDAP-like syntax based on the properties of the service;
* API interface – the OSGi remote services specification defines a mechanism for the exposal of OSGi services to the external world (effectively providing a mechanism for the definition of a public API of the module);
* security – certain capabilities (like permission checking) are provided by the Permission Admin and Cnditional Permission Admin utilities defined by the OSGi specification. Additional security capabilities are providing by other specifications such as the User Admin Service specification;
* centralized logging – that can be achieved through the OSGi log service specification;
* packaging – OSGi bundles are packaged as JAR files ready for deployment in the container;
* deployment – either from the OSGi console, from the file system (hot deploy) or through a dedicated tool/API (specific to the OSGi runtime).

However if we span outside the premises of the OSGi environment then we get into additional challenges related to all of the above including additional considerations such as load balancing, auto-scaling, self-healing, distributed tracing, resilience, fault tolerance and backpressure to name a few.

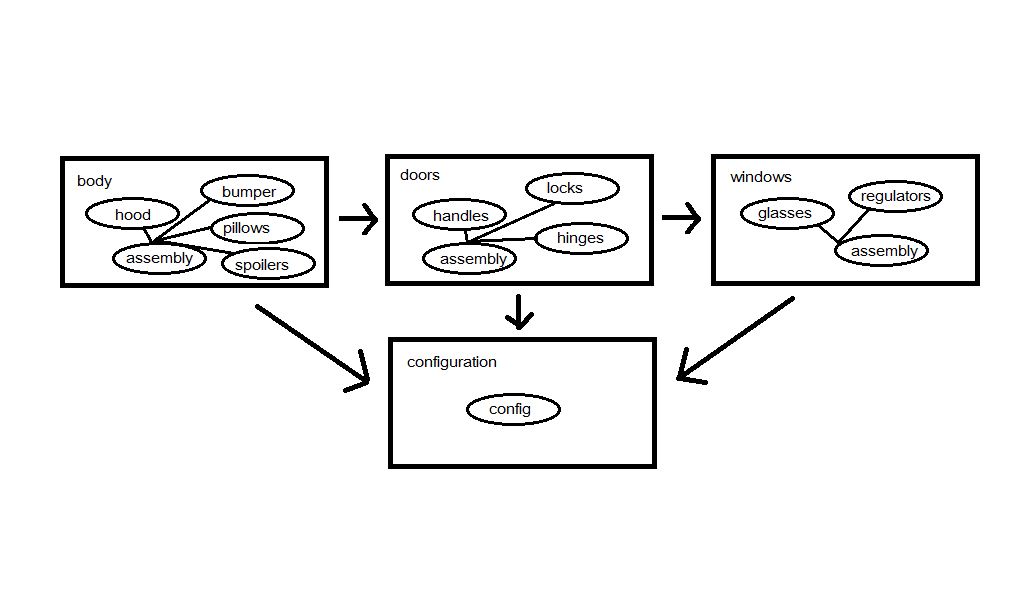
All of these can be enabled by means of the Remote Service and Remote Service Admin specification defined by the OSGi compendium specification and to be more price the Remote Service Admin spec defines a pluggable management agent called Topology Manager that can fulfill the different concepts mentioned that characterize the interaction between remote OSGi services (effectively the communication channel between the different OSGi runtimes that form the set of microservices).

# A microservice-based OSGi system

Now let’s demonstrate how the discussed architecture applies to a simple system for car manufacturing. For the purpose of simplicity we will limit the system to the production of the main parts. The system consists of three microservices which are distinct OSGi runtimes with different modules related to the particular microservice:

* body – includes separate modules for the production of the different body parts such as hood, bumper, pillars and spoilers;
* doors – includes separate modules for the production of door components such as handles, locks and hinges;
* windows - includes separate modules for the production of window components such as glasses and glass regulators.

The system works in a supply-chain manner whereby we first build the body of the car, then the doors and finally the windows before they are finally assembled. Moreover since each particular phase of the supply chain is modular we can upgrade or replace entirely modules that build certain parts (such as the hood) without really interacting with the full supply-chain and bringing the system out of operations. The different microservices gather configuration from a central configuration microservice deployed in a separate OSGi container having just a single configuration module. The system is illustrated schematically on the following diagram:



Each distinct microservice has an assembly module that is responsible to assemble the parts for the component represented by the microservice. Let’s jump straight to the implementation with some extra details.

Remote services provide a way to expose standard OSGi services to external applications using transport mechanisms such as SOAP or REST webservices, Apache Aries Remote Service Admin, r-osgi (Remote Services for OSGi) and others. Two of the most notorious frameworks that provide implementation of the remote service and remote service admin specifications are Apache CXF and the one provided by ECF (Eclipse Communication Framework).

We will be using the Apache CXF distributed OSGi subproject that implements a REST provider for the Aries Remote Service Admin. We will be deploying our application in a Karaf environment using Felix OSGi runtime (Karaf also provides support for running an Equinox OSGi runtime).

*Note that Karaf provides an alternative mechanism for creating a distributed OSGi runtime by means of the Karaf Cellar runtime.*

First download latest Karaf version from <https://karaf.apache.org/download.html>. Instead of downloading the CXF DOSGi distribution Karaf comes with a feature that installs it directly in Karaf. Unzip the Karaf distribution at a proper location and start the Karaf runtime with an interactive console as follows:

bin/karaf.bat

After Karaf is started install the CXF DOSGi feature as follows:

feature:repo-add cxf-dosgi  
feature:install cxf-dosgi-provider-rs

install file:///D:/projects/exoscale/repo\_karaf\_article/sources/config.api/target/config.api-1.0.0-SNAPSHOT.jar

install file:///D:/projects/exoscale/repo\_karaf\_article/sources/config/target/config-1.0.0-SNAPSHOT.jar

// feature:repo-add cxf-dosgi 2.0.0  
// feature:install cxf-dosgi-provider-rs

/

curl -X POST http://localhost:8181/cxf/config3/cxf/config2/add -d '{"key":"10","value":"abc"}' --header "Content-Type:application/json"

# Summary

We have managed to implement a complex style of architecture that is suitable for large scale systems using the OSGi framework in conjuction with the microservice paradigm. While our example was simple it really models an architecture that can span hundreds of microservices with multiple modules on each. OSGi remote services provided out of the box distributed service discovery and implementing concepts such as load balancing, fault tolerance and any of the others already mentioned can be done similarly to how we’ve introduced a centralized configuration service in the architecture.