Savaş Ali TOKMEN
http://ali.tokmen.com/

JOnAS – OSGi and Apache CAMEL integration

Summary: In this article I'll be describing some technologies such as OSGi, Java EE or EIPs, detail some products such as OW2 JOnAS or Apache CAMEL and finally how the integration of these products gives you a powerful yet lightweight and free (both as in free speech and free beer) framework.

Some technologies...

OSGi

The OSGi Alliance (formerly known as the Open Services Gateway initiative, now an obsolete name) is an open standards organization founded in March 1999 that originally specified and continues to maintain the OSGi standard. Among its members are more than 35 companies from quite different business areas, for example BMW, Ericsson, IBM, Motorola, Nokia, Oracle, Red Hat, VMware (SpringSource), etc.

OSGi is commonly referred to as “The Dynamic Module System for Java”. An OSGi module with a service interface (typically, Apache iPOJO):

- Provides a service layer, with weak and dynamic linking between the producer and consumer based on the SOA contract / bind / lookup pattern. Services can be started, stopped, bound, unbound and rebound dynamically.
- May require (i.e., “import”) Java packages with or without version numbers
- May provide (i.e., “export”) Java packages and their version numbers
- May require services to be present, which are seen by the application as pure, standard Java POJOs
- May provide services
- May have start and stop methods

Thanks to OSGi, you therefore can:

- Isolate each application and ease the independent development of each module
- Ease the integration of different software modules
- While deploying a module, guarantee that all prerequisites of an application or library are present
- Guarantee that different versions of the same libraries will not cause unexpected errors in your applications

- Dynamically deploy and start applications or libraries
- Dynamically stop and undeploy applications or libraries
- Dynamically update versions of applications or libraries

One major advantage of OSGi is that it is technically implemented by adding some properties to the `META-INF/MANIFEST.MF` file of a standard Java JAR file. As a result, for many "classic" libraries, the actual library module can be run as an OSGi module or executed as a standard Java program -by simply adding all required libraries to the Java classpath.

Today, OSGi is present in many Java frameworks of different sizes : small gateways, cars and even application servers. There are many open source or commercial implementation of OSGi frameworks: Apache Felix, Eclipse Equinox, Knopflerfish, etc.

Java EE

Java EE (Java Platform, Enterprise Edition) is a widely used platform for server programming in the Java programming language. The Java platform (Enterprise Edition) differs from the Java Standard Edition Platform (Java SE) in that it adds libraries which provide functionality to deploy fault-tolerant, distributed, multi-tier Java software, based largely on modular components running on an application server.

For the applications, Java EE can for example provide many standards:

- Multi-tier architecture in order to clearly separate rendering, business, storage and other code
- Web applications container, so you won't have to write and launch your own Web server
- Database access, so you won't battle with JDBC
- Persistence management, so you won't actually even have to write SQL queries
- Automated transaction management, guaranteeing data integrity
- Security management, with an easy-to-use RBAC context
- Asynchronous, reliable messaging
- Easy access to many technologies: XML input and output, Web Services, JSF, …
- Connectors for integration with existing enterprise applications

These standards, backed by many tools, drastically reduce the time required to develop, test applications and deploy enterprise-class applications. Moreover, since Java EE applications are made respecting the same standards, Java EE also eases the integration of compatible applications.

For the architects and administrators, Java EE standardizes the way applications are packaged; as an extra nearly all Java EE application servers provide standards for deploying, load balancing, clustering and monitoring:

- Multi-tier packaging: Web Applications, Java Beans, Transaction and entity manager, Resource Adapter, …
- Clustering, session replication, ...
- Centralized configuration: port numbers, component names, size and behavior of the various pools (threads, database, ...), garbage collection policies, ...
- MBeans (management beans) for accessing the state of many components and changing settings, locally or remotely

Java EE therefore rationalizes development and administration competences.

What's also extremely interesting with Java EE is that it is a standard, and the Java EE certification (tens of thousands of tests) can be passed by any person, organization or company; and many servers do: Apache Geronimo, IBM WebSphere, JBoss AS, Oracle GlassFish, Oracle WebLogic, OW2 JOnAS, etc. What this guarantees is that if your application does follow the Java EE standards (which all are well-established and recommended patterns), then it will work correctly on any of these servers. You therefore are both “high quality” and “free” (as in “freedom”), and can easily switch from an application server to another. Many Java EE users even develop on one server and deploy on another.

EIP

All organizations have their way of functioning for exchanging messages: some of their applications use a common database, some (generally, COBOL) read and write CSV files, some people like to send and receives files via e-mail, modern applications exchange XML files, Java EE applications might use message-driven JMS beans written in Java, some others use Web Services. All that represents a large number of formats and connectors.

When one needs to interconnect this following a certain “business logic” without any using any tools, not only he or she doesn't know which standards to follow for defining the business logic, but also that high number of formats and connectors makes the task extremely hard.

Enterprise Integration Patterns as described by Gregor Hohpe and Bobby Woolf in their book with the same name, are design patterns for the use of enterprise application integration and message-oriented middleware. A list of these patterns can be accessed via http://camel.apache.org/enterprise-integration-patterns.html and these patterns have been designed for being used by functional designer or architects; hence an advanced technical knownledge is not required.

These patterns therefore help you describe how to integrate all these applications and processes: where and how do we get the information, how do we aggregate information, how do we split the information, how do we send it to multiple applications or persons, etc. All this description is done without entering in the technical details.

In addition to the implementation of these patterns, a good EIP implementation ships with many common connectors and transformers, to help you play with the various protocols and data formats out of the box.

Tools that implement these technologies

Apache CAMEL: A multi-protocol EIP, compatible with OSGi

CAMEL is an Apache project, that uses a classic Apache license. It is therefore open source, available free of charge and has a non-viral license (i.e., can be used even in non-open products and projects). Of course, many companies (including the companies where most of the CAMEL architects and developers work) offer you nice support plans for CAMEL.

CAMEL comes with:

- A large number of EIP patterns, such as message channel, message transport, point-to-point, publish, dead letter, aggregator, splitter, multicast, …
- Tens of components (i.e., protocol connectors): Java Bean, File (local and remote), FTP, SFTP, POP3, IMAP, SMTP, HTTP, SOAP, Web Services via CXF, JMS, TCP, UDP, … a full list can be accessed via http://camel.apache.org/components.html
- Tens of data formats: CSV, XML, JSON, Zip, … see the list on the CAMEL web site, http://camel.apache.org/data-format.html for the whole list
- Programmability via multiple languages: Java, Spring XML, DSL, SQL, XPath, many scripting languages such as Groovy or Python, …

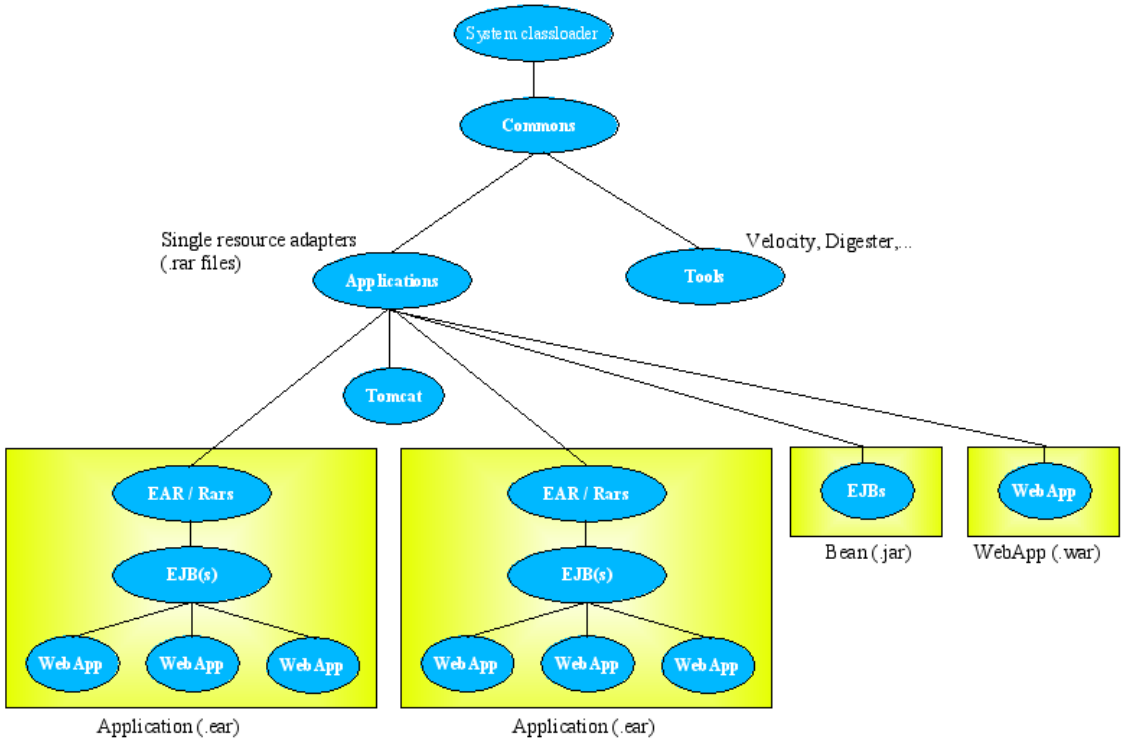
- Extensibility: you can add your own components, data formats and data converters.
- OSGi integration: all these components can be used in a “standalone” mode as well as in an OSGi framework.

When used in an OSGi framework, CAMEL:

- Reuses existing services such as HTTP service, thread pool, libraries (CXF, Spring, …), registries (OSGi, JNDI, …), dynamic class and component loading, etc.
- Enriches the platform with its routing, transformation and other processing capabilities.

Java EE on the top of OSGi

“Classical” Java EE servers are designed using a “tree-like” classpath model: the Java EE server has a “commons” classloader containing all Java EE APIs and implementations (even these no applications use), on which we put the applications' shared classpath, and finally one classpath for each application. For example, the JOnAS 4 classloader hierarchy looks like the following:



1

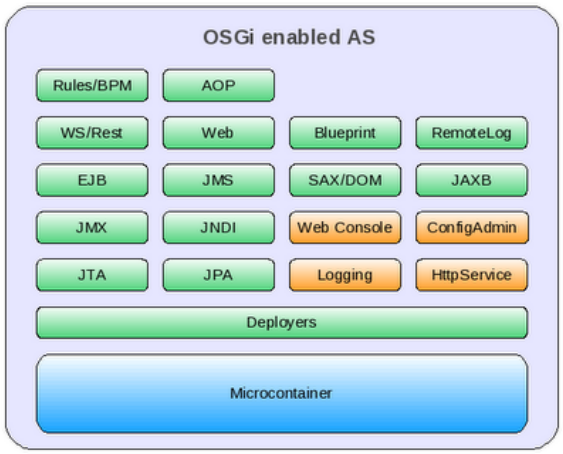
That organization has been a burden for all Java EE servers:

- If the server provides a certain version of a library but the application needs (and ships with) another, “weird” and extremely hard to solve errors occur.
- The “tree-like” classpath model requires many libraries to be indexed (and even for some JVMs loaded) even if they won't be used for the whole lifetime of the server. This made many people have the impression that Java EE servers are “heavy”.

When it was first designed, OSGi was intended for small-sized applications; the typical example being the “embedded gateway”. As a result, OSGi is actually optimized for low memory and high performance environments and therefore minimizes memory footprint while at the same time providing dynamic management capabilities.

1 Image taken from the JOnAS's documentation “JOnAS 4 class loader” chapter on http://jonas.ow2.org/JONAS_4_10/doc/doc-en/integrated/PG_J2eeApps.html

OSGi has a “flat” classloader (as opposed to the “tree-like” classloader): each classloader does not have a “parent classloader” but rather imports Java packages from other classloaders, and sometimes provides Java packages that can be used by other classloaders. Each service that runs on these classloaders also provide and consume services. It can be schematized as follows:



2

As a result, the “HTTP Service” for example is not anymore a service that lies in the “commons” classloader but is rather a standalone service that can reuse other services of the platform (typically, JNDI or logging).

It becomes more interesting when we look at applications: your Web application is not anymore an application that reuses a large package called “commons”, but rather a certain number of “smaller” services of the platform: HTTP Service, JNDI, logging, JAXB, REST and persistence, for example.

Being extremely modular, OSGi-based Java EE application servers can easily be shipped as different “profiles”:

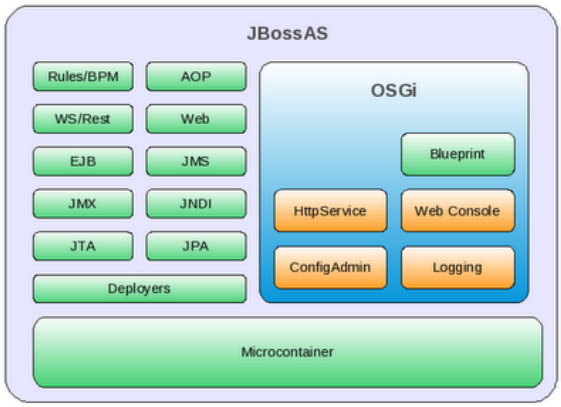
- A “micro” profile that can for example only contain the minimum for starting the server and remote administration; so any required service will be downloaded and started dynamically.
- A “web” profile that would include only the Web-related components.
- A “full” profile with all features.
- etc.

Today, only two servers have been able of implementing Java EE on the top of OSGi:

- OW2 JOnAS (http://jonas.ow2.org/) is the first Java EE certified server that's fully based on OSGi (version 5.1.0 M5, March 2009). It is certified as being fully compatible with the Java EE 5 specifications.
- Oracle/Sun GlassFish (http://glassfish.dev.java.net/) is the first Java EE version 6 certified server that's fully based on OSGi (version 3, December 2009). It is also the first Java EE version 6 certified server.

2 Image taken from the JBoss OSGi diary: http://jbossosgi.blogspot.com/2009_06_01_archive.html

As a side note, it is important to precise that some servers, most notably JBoss, come along with some "OSGi support" but are not based on OSGi: they simply deploy an OSGi runtime side-to-side to their existing container implementation. It therefore looks like the following:



3

That kind of implementation does solve any issues, and does not even bring in any advantages:

- The classloader is still "tree-like" and does not take advantage of OSGi.
- The microcontainer is application-server-specific and not based on an independent OSGi-compliant framework.
- Applications still suffer of the component and library version issues.
- Worse, the OSGi framework is not even aware that it is in a platform that has useful services. It will therefore not be able of reusing the HTTP connector, the persistence provider, …

The JBoss-OSGi projects aims at solving these issues and "replicate" what has been done in OW2 JOnAS and Sun/Oracle GlassFish but does not look ready yet. The current JBoss roadmap points us that this should be done for JBoss AS version 6 (currently available as a milestone).

OSGi + Java EE + EIP: The platform that "does it all"

Integration example: OW2 JOnAS (based on Apache Felix) + Apache Camel

In this part, we will be describing one integration of Java EE and EIP, all this taking advantage of the OSGi framework. The OW2 JOnAS server packaged with the Apache CAMEL EIP solution is available on the following URLs:

- The whole integration source code, with examples, documentation and integration tests, is on http://repo2.maven.org/maven2/org/ow2/jonas/camel/package-sources/
- The pre-packaged binary can also be directly downloaded, using the following URL: http://repo2.maven.org/maven2/org/ow2/jonas/camel/package-with-jonas/

That solution is based on an LGPL license, it is "free" as in "free beer" and "free speech". Finally, the "L" of LGPL implies that you can use it as you wish in any project.

3 Image taken from the JBoss OSGi diary: http://jbossosgi.blogspot.com/2009_06_01_archive.html

Technically, that integration takes advantage of all features described before:

- The Java EE server is used as the basis for the integration of many types of existing applications and most importantly the integration of many types of enterprise components.
- The Java EE server centralizes the configuration of many aspects of the server (port numbers, component names, clustering, session replication, data sources -including drivers for many different vendors, etc.).
- Apache CAMEL follows the Java Management standards (JMX) and makes its components, routes and processors available as MBeans, for both monitoring and administration purposes.
- The OSGi framework makes trivial the integration of Apache Camel and OW2 JOnAS.
- Finally, Camel routes can be creates, tested and deployed using the exact same set of tools as the ones used for OW2 JOnAS: IDEs, Maven2 and associated plugins, OW2 JASMINe, the JMX console, etc.

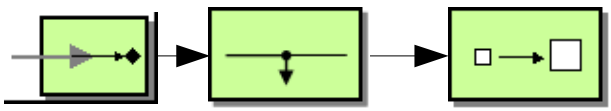
We will be showing each of these points in the following steps.

First, the integration: as soon as you place the Apache CAMEL bundles onto the JONAS_BASE/deploy directory of JOnAS, you'll see a message:

```
Activator.start : Camel activator starting
Activator.start : Camel activator started
CamelService.__initialize : Camel service started
```

As you can see, since OW2 JOnAS is already running on an OSGi framework and Apache CAMEL had been packaged as OSGi bundles, we did not have to repackage anything, write any deployment descriptors of any sort for CAMEL to automatically bind itself with the existing OSGi services provided by the Java EE server (such as JNDI, JDBC, persistence, HTTP or Web Service).

In order to create a route that actually does something, we simply define our CAMEL route builder (which will later be deployed as an OSGi service). First, the process described using EIP:



Our (extremely simple) process:

- Receives a message
- Logs the message
- Enriches the message
- Responds with the enriched message

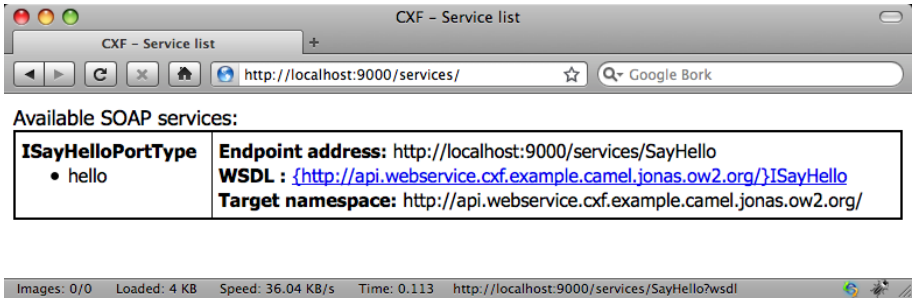
Described using the Java DSL, this can be written as follows:

```
@Override
public void configure() throws Exception {
    super.configure();

    // This is the global logger, which simply takes all messages
    // and writes a copy in the /var/log/camel directory
    this.getContext().getDefaultTracer().setDestinationUri("file:///var/log/camel");

    this.from(
        // Receive a message, using Web Service
        "cxf://http://localhost/services/SayHello?"
      + "serviceClass=org.ow2.jonas.camel.example.cxf.webservice.api.ISayHello&"
      + "bus=#cxfBus")
        .process(new Processor() {
            public void process(final Exchange exchange) throws Exception {
                MessageContentsList msgList =
                    exchange.getIn().getBody(MessageContentsList.class);
                String name = (String) msgList.get(0);

                // Enrich the message with a "hello, "
                MessageContentsList response = new MessageContentsList();
                response.add("hello, " + name);

                // Respond
                exchange.getOut().setBody(response);
            }
        });
}
```

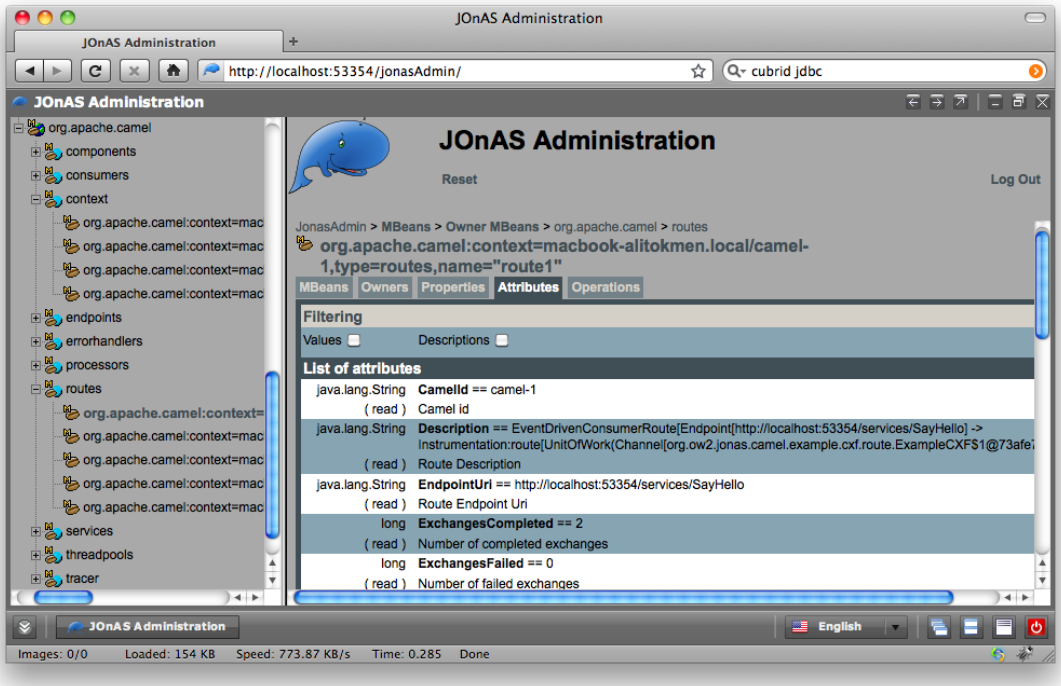
Once we deploy that route on the OSGi-based Java EE server with CAMEL, it will automatically bind itself with the existing CAMEL context, the OSGi HTTP Service, the Web Service components deployed by the Java EE server and immediately start working:

```
CamelService.__startNewContext : Starting a new camel context
    ...
ExampleCXF$1$1.process : Received CXF message: guillaume
ExampleCXF.test : Got CXF response: hello, guillaume
```

CAMEL integrates itself nicely with the Java EE server, for example the Web Services it deploys are visible on the CXF registry:



What's even more interesting is that CAMEL will enrich the existing MBeans using its own, routing-specific MBeans:



It is therefore trivial to create, deploy, monitor and administer our CAMEL-based modules deployed on an OSGi-based Java EE server: existing monitoring and administration tools will indeed simply have to take into account the extra MBeans deployed by CAMEL.

Java EE + EIP: A large scope of possibilities for applications

Java EE and EIP coupled together indeed represents and extremely interesting opportunity for application and process integration:

- Existing Java EE application keep on running “exactly as they used to”: all Java EE services, such as Web, servlets, transaction management, JNDI, JMS, EJB, … are available, exactly as they used to be.
- The EIP platform manages connectivity, message format conversion, aggregation and message routing between all applications and persons -no matter how they used to communicate and what data format they used to use.
- Existing Java EE applications can be accessed as standard Java beans, injected using OSGi annotations. As a result, for Java EE applications, all external applications and processes are accessed exactly as if they were local Java services.
- Applications and people that want to exchange data simply expose and retrieve messages exactly they used to be. The EIP platform then handles all the message retrieval, data format conversion and message routing tasks.

Moreover, deployment, monitoring and management of the EIP platform can be done using the existing Java EE server tools.

Conclusion

The OSGi-based JOnAS – Camel integration is a concrete implementation of the ESB/Java EE integration. The typical use cases can be:

- Application integration, including Java EE, existing Java applications (which can be developed independently) and many open source and commercial projects (many of which already packaged as OSGi bundles).
- Exchange server: it can synchronously or asynchronously link existing applications or business processes.
- Service proxy, in particular for exposing existing applications using modern protocols without changing the applications.

Want to give a try?

The OW2 JOnAS server packaged with the Apache CAMEL EIP solution is available on the following URLs:

- The official JOnAS – Camel page:

  http://wiki.jonas.ow2.org/xwiki/bin/view/Main/JOnASCamel

- The JOnAS downloads page:

  http://wiki.jonas.ow2.org/xwiki/bin/view/Main/Downloads

That solution is based on an LGPL license, it is “free” as in “free beer” and “free speech”. Finally, the “L” of LGPL implies that you can use it as you wish in any project.

Enjoy!