The OSGi Architecture

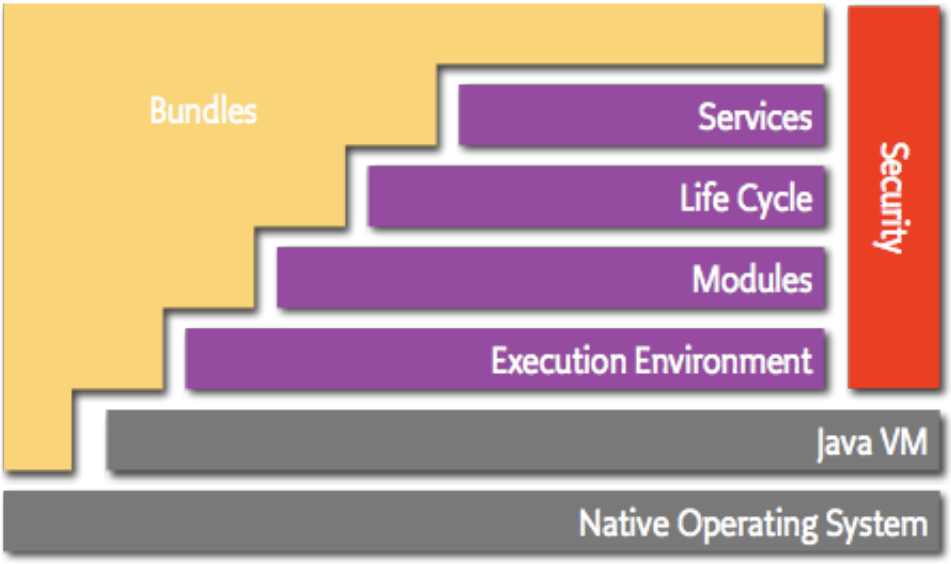
The OSGi technology is a set of *specifications* that define a dynamic component system for Java. These specifications enable a development model where applications are (dynamically) composed of many different (reusable) components. The OSGi specifications enable components to hide their implementations from other components while communicating through *services*, which are objects that are specifically shared between components. This surprisingly simple model has far reaching effects for almost any aspect of the software development process.

Though components have been on the horizon for a long time, so far they failed to make good on their promises. OSGi is the first technology that actually succeeded with a component system that is solving many real problems in software development. Adopters of OSGi technology see significantly reduced complexity in almost all aspects of development. Code is easier to write and test, reuse is increased, build systems become significantly simpler, deployment is more manageable, bugs are detected early, and the runtime provides an enormous insight into what is running. Most important, it works as is testified by the wide adoption and use in popular applications like Eclipse and Spring.

We developed the OSGi technology to create a *collaborative* software environment. We were not looking for the possibility to run multiple applications in a single VM. Application servers do that already (though they were not yet around when we started in 1998). No, our problem was harder. We wanted an application to *emerge* from putting together different reusable components that had no *a-priori* knowledge of each other. Even harder, we wanted that application to emerge from *dynamically* assembling a set of components. For example, you have a home server that is capable of managing your lights and appliances. A component could allow you to turn on and off the light over a web page. Another component could allow you to control the appliances via a mobile text message. The goal was to allow these other functions to be added without requiring that the developers had intricate knowledge of each other and let these components be added independently.

Layering

The OSGi has a layered model that is depicted in the following figure.



The following list contains a short definition of the terms:

- Bundles - Bundles are the OSGi components made by the developers.
- Services - The services layer connects bundles in a dynamic way by offering a publish-find-bind model for plain old Java objects.
- Life-Cycle - The API to install, start, stop, update, and uninstall bundles.

- Modules - The layer that defines how a bundle can import and export code.
- Security - The layer that handles the security aspects.
- Execution Environment - Defines what methods and classes are available in a specific platform.

These concepts are more extensively explained in the following sections.

Modules

The fundamental concept that enables such a system is *modularity*. Modularity, simplistically said, is about assuming less. Modularity is about keeping things local and not sharing. It is hard to be wrong about things you have no knowledge of and make no assumptions about them. Therefore, modularity is at the core of the OSGi specifications and embodied in the *bundle* concept. In Java terms, a bundle is a plain old JAR file. However, where in standard Java everything in a JAR is completely visible to all other JARs, OSGi hides everything in that JAR unless explicitly exported. A bundle that wants to use another JAR must explicitly import the parts it needs. By default, there is no sharing.

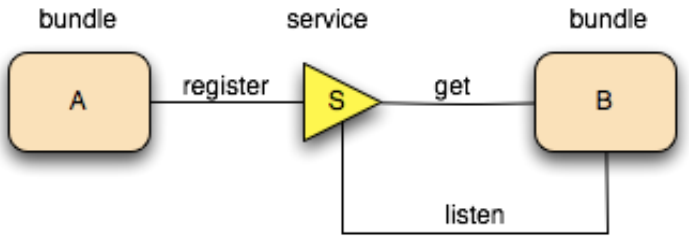
Though the code hiding and explicit sharing provides many benefits (for example, allowing multiple versions of the same library being used in a single VM), the code sharing was only there to support OSGi *services* model. The services model is about bundles that collaborate.

Services

The reason we needed the service model is because Java shows how hard it is to write collaborative model with only class sharing. The standard solution in Java is to use *factories* that use dynamic class loading and statics. For example, if you want a DocumentBuilderFactory, you call the static factory method DocumentBuilderFactory.newInstance(). Behind that façade, the newInstance methods tries every class loader trick in the book (and some that aren't) to create an instance of an implementation subclass of the DocumentBuilderFactory class. Trying to influence what implementation is used is non-trivial (services model, properties, conventions in class name), and usually global for the VM. Also it is a *passive model*. The implementation code can not do anything to advertise its availability, nor can the user list the possible implementations and pick the most suitable implementation. It is also not dynamic. Once an implementation hands out an instance, it can not withdraw that object. Worst of all, the factory mechanism is a *convention* used in hundreds of places in the VM where each factory has its own unique API and configuration mechanisms. There is no centralized overview of the implementations to which your code is bound.

The solution to all these issues is the OSGi *service registry*. A bundle can create an object and register it with the OSGi service registry under one or more interfaces. Other bundles can go to the registry and list all objects that are registered under a specific interfaces or class. For example, a bundle provides an implementation of the DocumentBuilder. When it gets started, it creates an instance of its DocumentBuilderFactoryImpl class and registers it with the registry under the DocumentBuilderFactory class. A bundle that needs a DocumentBuilderFactory can go to the registry and ask for all available services that extend the DocumentBuilderFactory class. Even better, a bundle can wait for a specific service to appear and then get a call back.

A bundle can therefore *register* a service, it can *get* a service, and it can *listen* for a service to appear or disappear. Any number of bundles can register the same service type, and any number of bundles can get the same service. This is depicted in the following figure.



What happens when multiple bundles register objects under the same interface or class? How can these be distinguished? The answer is properties. Each service registration has a set of standard and custom properties. A expressive filter language is available to select only the services in which you are interested. Properties can be used to find the proper service or can play other roles at the application level.

Services are dynamic. This means that a bundle can decide to withdraw its service from the registry while other bundles are still using this service. Bundles using such a service must then ensure that they no longer use the service object and drop any references. We know, this sounds like a significant complexity but it turns out that helper classes like the Service Tracker and frameworks like iPOJO, Spring, and Declarative Services can make the pain minimal while the advantages are quite large. The service dynamics were added so we could install and uninstall bundles on the fly while the other bundles could adapt. That is, a bundle could still provide functionality even if the http service went away. However, we found out over time that the real world is dynamic and many problems are a lot easier to model with dynamic services than static factories. For example, a Device service could represent a device on the local network. If the device goes away, the service representing it is unregistered. This way, the availability of the service models the availability of a real world entity. This works out very well in, for example, the distributed OSGi model where a service can be withdrawn if the connection to the remote machine is gone. It also turns out that the dynamics solve the initialization problem. OSGi applications do not require a specific start ordering in their bundles.

The effect of the service registry has been that many specialized APIs can be much modeled with the service registry. Not only does this simplify the overall application, it also means that standard tools can be used to debug and see how the system is wired up.

Though the service registry accepts any object as a service, the best way to achieve reuse is to register these objects under (standard) interfaces to decouple the implementer from the client code. This is the reason the OSGi Alliance publishes the Compendium specifications. These specification define a large number of standard services, from a Log Service to a Measurement and State specification. All these standardized services are described in great detail.

Deployment

Bundles are deployed on an OSGi *framework*, the bundle runtime environment. This is not a container like Java Application Servers. It is a *collaborative environment*. Bundles run in the same VM and can actually share code. The framework uses the explicit imports and exports to wire up the bundles so they do not have to concern themselves with class loading. Another contrast with the application servers is that the management of the framework is standardized. A simple API allows bundles to install, start, stop, and update other bundles, as well as enumerating the bundles and their service usage. This API has been used by many *management agents* to control OSGi frameworks. Management agents are as diverse as the Knopflerfish desktop and an IBM Tivoli management server.

Implementations

The OSGi specification process requires a reference implementation for each specification. However, since the first specifications there have always been commercial companies that have implemented the specifications as well as open source implementations. Currently, there are 4 open source implementations of the framework and too many to count implementations of the OSGi services. The open software industry has discovered OSGi technology and more and more projects deliver their artifacts as bundles.

Conclusion

The OSGi specifications provide a mature and comprehensive component model with a very effective (and small) API. Converting monolithic or home grown plugin based systems to OSGi almost always provides great improvements in the whole process of developing software.

Further reading

- Benefits of using OSGi
- How to get Started with OSGi?
- Technology Overview