Benefits of Using OSGi

DEVELOPERS

OSGi reduces complexity by providing a modular architecture for today's large-scale distributed systems as well as small, embedded applications. Building systems from in-house and off-the-shelf modules significantly reduces complexity and thus development and maintenance expenses. The OSGi programming model realizes the promise of component-based systems.

BUSINESS

The OSGi modular and dynamic model reduces operational costs and integrates multiple devices in a networked environment, tackling costly application development, maintenance and remote service management.

The key reason OSGi technology is so successful is that it provides a very mature component system that actually works in a surprising number of environments. The OSGi component system is actually used to build highly complex applications like IDEs (Eclipse), application servers (GlassFish, IBM Websphere, Oracle/BEA Weblogic, Jonas, JBoss), application frameworks (Spring, Guice), industrial automation, residential gateways, phones, and so much more.

So, what benefits does OSGi's component system provide you? Well, quite a list:

- **Reduced Complexity** - Developing with OSGi technology means developing bundles: the OSGi components. Bundles are modules. They hide their internals from other bundles and communicate through well defined *services*. Hiding internals means more freedom to change later. This not only reduces the number of bugs, it also makes bundles simpler to develop because correctly sized bundles implement a piece of functionality through well defined interfaces. There is an interesting blog that describes what OSGi technology did for their development process.
- **Reuse** - The OSGi component model makes it very easy to use many third party components in an application. An increasing number of open source projects provide their JARs ready made for OSGi. However, commercial libraries are also becoming available as ready made bundles.
- **Real World** - The OSGi framework is dynamic. It can update bundles on the fly and services can come and go. Developers used to more traditional Java see this as a very problematic feature and fail to see the advantage. However, it turns out that the real world is highly dynamic and having dynamic services that can come and go makes the services a perfect match for many real world scenarios. For example, a service could model a device in the network. If the device is detected, the service is registered. If the device goes away, the service is unregistered. There are a surprising number of real world scenarios that match this dynamic service model. Applications can therefore reuse the powerful primitives of the service registry (register, get, list with an expressive filter language, and waiting for services to appear and disappear) in their own domain. This not only saves writing code, it also provides global visibility, debugging tools, and more functionality than would have implemented for a dedicated solution. Writing code in such a dynamic environment sounds like a nightmare, but fortunately, there are support classes and frameworks that take most, if not all, of the pain out of it.
- **Easy Deployment** - The OSGi technology is not just a standard for components. It also specifies how components are installed and managed. This API has been used by many bundles to provide a management agent. This management agent can be as simple as a command shell, a TR-69 management protocol driver, OMA DM protocol driver, a cloud computing interface for Amazon's EC2, or an IBM Tivoli management system. The standardized management API makes it very easy to integrate OSGi technology in existing and future systems.
- **Dynamic Updates** - The OSGi component model is a dynamic model. Bundles can be installed, started, stopped, updated, and uninstalled without bringing down the whole system. Many Java developers do not believe this can be done reliably and therefore initially do not use this in production. However, after using this in development for some time, most start to realize that it actually works and significantly reduces

deployment times.

- **Adaptive** - The OSGi component model is designed from the ground up to allow the mixing and matching of components. This requires that the dependencies of components need to be specified and it requires components to live in an environment where their optional dependencies are not always available. The OSGi service registry is a dynamic registry where bundles can register, get, and listen to services. This dynamic service model allows bundles to find out what capabilities are available on the system and adapt the functionality they can provide. This makes code more flexible and resilient to changes.
- **Transparency** - Bundles and services are first class citizens in the OSGi environment. The management API provides access to the internal state of a bundle as well as how it is connected to other bundles. For example, most frameworks provide a command shell that shows this internal state. Parts of the applications can be stopped to debug a certain problem, or diagnostic bundles can be brought in. Instead of staring at millions of lines of logging output and long reboot times, OSGi applications can often be debugged with a live command shell.
- **Versioning** - OSGi technology solves JAR hell. JAR hell is the problem that library A works with library B;version=2, but library C can only work with B;version=3. In standard Java, you're out of luck. In the OSGi environment, all bundles are carefully versioned and only bundles that can collaborate are wired together in the same *class space*. This allows both bundle A and C to function with their own library. Though it is not advised to design systems with this versioning issue, it can be a life saver in some cases.
- **Simple** - The OSGi API is surprisingly simple. The core API is only one package and less than 30 classes/interfaces. This core API is sufficient to write bundles, install them, start, stop, update, and uninstall them and includes all listener and security classes. There are very few APIs that provide so much functionality for so little API.
- **Small** - The OSGi Release 4 Framework can be implemented in about a 300KB JAR file. This is a small overhead for the amount of functionality that is added to an application by including OSGi. OSGi therefore runs on a large range of devices: from very small, to small, to mainframes. It only asks for a minimal Java VM to run and adds very little on top of it.
- **Fast** - One of the primary responsibilities of the OSGi framework is loading the classes from bundles. In traditional Java, the JARs are completely visible and placed on a linear list. Searching a class requires searching through this (often very long, 150 is not uncommon) list. In contrast, OSGi pre-wires bundles and knows for each bundle exactly which bundle provides the class. This lack of searching is a significant speed up factor at startup.
- **Lazy** - Lazy in software is good and the OSGi technology has many mechanisms in place to do things only when they are really needed. For examples, bundles can be started eagerly, but they can also be configured to only start when another bundle is using them. Services can be registered but only created when they are used. The specifications have been optimized several times to allow for these kind of lazy scenarios that can save tremendous runtime costs.
- **Secure** - Java has a very powerful fine grained security model at the bottom but it has turned out very hard to configure in practice. The result is that most secure Java applications are running with a binary choice: no security or very limited capabilities. The OSGi security model leverages the fine grained security model but improves the usability (as well as hardening the original model) by having the bundle developer specify the requested security details in an easily audited form while the operator of the environment remains fully in charge. Overall, OSGi likely provides one of the most secure application environments that is still usable short of hardware protected computing platforms.
- **Humble** - Many frameworks take over the whole VM, they only allow one instance to run in a VM. The flexibility of the OSGi specifications is demonstrated by how it can even run inside a J2EE Application Server. Many developers wanted to run OSGi but their companies did not allow them to deploy normal JARs. Instead, they included an OSGi framework in their WAR file and loaded their bundles from the file system or over the network. OSGi is so flexible that one application server can easily host multiple OSGi frameworks.
- **Non Intrusive** - Applications (bundles) in an OSGi environment are left to their own. They can use virtually any facility of the VM without the OSGi restricting them. Best practice in OSGi is to write Plain Old Java Objects and for this reason, there is no special interface required for OSGi services, even a Java String object can act as an OSGi service. This strategy makes application code easier to port to another

environment.

- **Runs Everywhere** - Well, that depends. The original goal of Java was to run anywhere. Obviously, it is not possible to run all code everywhere because the capabilities of the Java VMs differ. A VM in a mobile phone will likely not support the same libraries as an IBM mainframe running a banking application. There are two issue to take care of. First, the OSGi APIs should not use classes that are not available on all environments. Second, a bundle should not start if it contains code that is not available in the execution environment. Both of these issues have been taken care of in the OSGi specifications.
- **Widely Used** - The OSGi specifications started out in the embedded home automation market but since 1998 they have been extensively used in many industries: automotive, mobile telephony, industrial automation, gateways & routers, private branch exchanges, fixed line telephony, and many more. Since 2003, the highly popular Eclipse Integrated Development Environment runs on OSGi technology and provides extensive support for bundle development. In the last few years, OSGi has been taken up by the enterprise developers. Eclipse developers discovered the power of OSGi technology but also the Spring Framework helped popularize this technology by creating a specific extension for OSGi. Today, you can find OSGi technology at the foundation of IBM Websphere, SpringSource dm Server, Oracle (formerly BEA) Weblogic, Sun's GlassFish, and Redhat's JBoss.
- **Supported by Key Companies** - OSGi counts some of the largest computing companies from a diverse set of industries as its members. Members are from: Oracle, IBM, Samsung, Nokia, Progress, Motorola, NTT, Siemens, Hitachi, Deutsche Telekom, Redhat, Ericsson, and many more.

The OSGi specifications were started in 1998 and were intended for the home automation market, trying to solve the problem how to build applications out of independent components. In the this past decade, the software industry was fundamentally changed because of the explosion in open source projects. Ten years ago, an application consisted mostly of specifically written code. Today, most software is largely wiring up open source artifacts that were often not designed to work together. This is similar to the problem that OSGi was designed to solve. Many open source projects are therefore adopting the OSGi specifications because they see that they can focus on the real problem and worry less about infrastructure, as well as becoming easier to use in other projects. This trend is accelerating.

If you are developing software in Java then OSGi technology should be a logical next step because it solves many problems that you might not even be aware can be solved. The advantages of OSGi technology are so numerous that if you are using Java, then OSGi **should** be in your tool chest.

Further reading

- The OSGi Architecture
- How to get Started with OSGi?
- Links