# About the OSGi Service Platform

Technical Whitepaper

Revision 4.1
7 June 2007

[This page intentionally left blank.]

# Executive Summary

The OSGi™ specifications define a *standardized*, *component-oriented*, computing environment for *networked services* that is the foundation of an enhanced *service-oriented architecture*. Adding an OSGi Service Platform to a networked device (embedded as well as servers), adds the capability to manage the lifecycle of the software components in the device from anywhere in the network. Software components can be installed, updated, or removed on the fly without ever having to disrupt the operation of the device.

Software components are libraries or applications that can dynamically discover and use other components. Software components can be bought off the shelf or developed in house. The OSGi Alliance has developed many standard component interfaces that are available for common functions like HTTP servers, configuration, logging, security, user administration, XML, and many more. Plug-compatible implementations of these components can be obtained from different vendors with different optimizations to suit the needs of different markets.

Software component architectures address an increasing problem in software development: The large number of configurations that need to be developed and maintained. The standardized OSGi component architecture simplifies this configuration process significantly.

The OSGi specifications were initially targeted at residential Internet gateways with Home Automation applications. However, the attributes of the standard made it applicable, and attractive, to other markets. For example, Nokia and Motorola drove an OSGi technology standard for the next generation of smart phones. The vehicle industry adopted the OSGi specifications by making them an intrinsic part of the GST specifications, which is supported by many car manufacturers. The OSGi Service Platform has become a standard part of the BMW high-end telematics platform and is finding its way into many Volkswagens, and standards like CVIS and VII are taking OSGi technology into their considerations.

OSGi technology is valuable on the desktop, with Eclipse as an example how the OSGi specifications solved the problem of dynamically updating the plugins that Eclipse is built upon. The Eclipse Rich Client Platform (RCP) drives many companies to implicitly adopt the OSGi plug-in architecture for desktop applications.

Recently, there has been a surge in interest in OSGi technology from enterprise developers. The simplicity and leanness of OSGi technology is refreshing in comparison with the established J2EE standard. The combination of Spring and OSGi technology has enticed many enterprise developers to adopt OSGi technology.

The OSGi specifications are so widely applicable because they form a small layer that allows multiple, Java™-based, components to efficiently cooperate in a single Java Virtual Machine (JVM). The technology provides an extensive security model so that components can run in a shielded environment. However, with the proper permissions, components can reuse and cooperate, unlike other Java application environments. The OSGi Framework provides an extensive array of mechanisms to make this cooperation possible and also secure. The presence of OSGi technology-based middleware in many different industries is creating a large software market for OSGi software components. The OSGi Service Platform enables components to run on a variety of devices, from very small to very big.

In summary, the OSGi specifications create universal middleware that is cross platform as well as cross industry. Adoption of the OSGi specifications reduces software development and maintenance costs and provides new business opportunities.

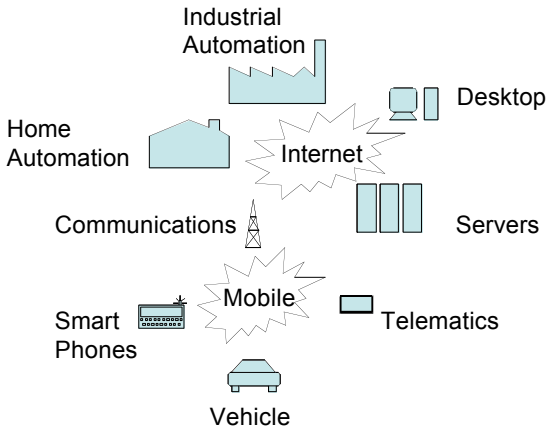
This white paper explains in detail the features of the OSGi specifications as well as the technical architecture of the OSGi Service Platform.

# Introduction

The OSGi Service Platform is the optimal Java-based application server for networked devices, however small or large they are. This non-proprietary service platform spans:

- Digital mobile phones
- Vehicles
- Telematics
- Embedded appliances
- Residential gateways
- Industrial computers
- Desktop PCs
- High-end servers, including mainframes.

*The different areas where the OSGi Service Platform is being used*




The *scope* of the OSGi Service Platform is:

- A standard, non-proprietary, software component framework for manufacturers, service providers, and developers. The fact that the OSGi specifications are an open standard enables a fair playing field for all participants.
- A powerful model for co-existence of different components/applications in a single JVM. Running multiple applications in the same JVM minimizes the memory footprint, increases performance, and provides near zero-cost inter-application communication.
- A flexible deployment Application Programming Interface (API) that controls the life-cycle of applications. Applications are installed with a standardized deployment format and can then be started, stopped, updated, and uninstalled without requiring the JVM to be restarted. This is a must for continuous computing.
- A secure environment that executes applications in a sandbox so that these applications cannot harm the environment, nor interfere with other resident applications. This model allows less trusted applications to run inside the JVM without compromising the overall integrity of the system.
- A cooperative model where applications can dynamically discover and use services provided by other applications running inside the same OSGi Service Platform. This cooperative service model allows OSGi applications to be much smaller than applications for other Java application servers. A key benefit for mobile devices.
- A flexible remote management architecture that allows platform operators (the organization that manages the platform) and enterprises to manage thousands or even millions of Service Platforms from a single management domain. The OSGi Service Platform architecture allows the operator to control a platform in fine detail by using a model where the operators can ensure that their required policies are implemented, however secure or flexible those policies are desired to be: from a walled garden to the anarchy of a PC.
- A number of standardized, optional services: Logging, Configuration, HTTP, XML, Wiring, IO, Event Handling, Device Discovery and driver loading, User Authentication

and Authorization, Preferences, Device Management, Component Wiring, Event Management, UPnP, and many more. Additionally, it provides a number of standardized utilities.

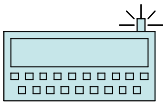
This is a broad scope. The following use cases further elucidate the scope of the OSGi Service Platform.

**Enterprise Solutions**

One of the surprising aspects of the OSGi specifications is that they were targeted at small, embedded devices but have become popular in high-end servers, up to mainframes. A large part of the success in this market is driven by the complexity of the Java Enterprise Edition (J2EE), J2EE did not heed many of the lessons that make frameworks hard to use, like inversion of control (IoC) and dependency injection. In contrast, the OSGi framework tries to stay out of the way as much as possible, allowing code to be more reusable in other applications.

However, the OSGi specifications do not address many of the issues that were addressed in J2EE like databases, transactions, messaging, etc. -- APIs that are crucial for most enterprise applications. Interestingly, in this area J2EE was competing with Spring. Spring is a framework that simplifies application development by focusing on configuring objects that have no coupling to any framework (POJOs, plain old Java objects), as well as providing libraries that abstract the POJOs from any dependency on other frameworks. Their customers pushed the company that shepherds Spring, Interface21, to integrate Spring with OSGi technology. This combination turned out to be very useful. Both Spring and OSGi technology are very orthogonal; there is little overlap. As Rod Johnson (Interface21) says: “[OSGi technology] will strengthen Spring's value as a basis for server infrastructure and offer benefits to users in the area of componentization, versioning and dynamic deployment.”

The adoption of OSGi technology by the enterprise developers has not gone unnoticed by the OSGi Alliance. They started an Enterprise Expert Group late 2006 that will work on Enterprise specific applications in 2007. This group will focus on the needs of enterprise developers. It is likely that this group will develop specifications for distribution of OSGi, management, integration of existing standards, and much more.



**Next Generation Smart Phones**

Java is tremendously successful in today’s low- and mid-range mobile phone market with more than 1 billion phones (and more than 300 models) that can download and run Java applications. However, these mobile Java applications must conform to the Mobile Information Device Profile (MIDP) specification based on the Connected Limited Device Configuration (CLDC VM), which is suitable for games and small applications but is too limited for compelling system and business applications. The market for mobile enterprise applications and more complex consumer applications that utilize the capabilities of smart phones has, therefore, never taken off. Mobile phone manufacturers, faced with more mobile platforms and custom software adaptations than they can manage, need a service platform that is scalable, flexible, reliable, and, most importantly, has a small footprint. Also, enterprises that want to deploy companywide applications are faced today with a bewildering array of choices. A unified mobile platform will allow a new breed of applications to emerge.

The OSGi specifications fulfill these requirements. It was, therefore, decided to develop additional specifications in the OSGi Alliance together with the Java Community Process (JSR 232) to fully match the requirements of the mobile phone manufacturers, operators and enterprises. As Jon Bostrom, Director of Java Technology, Nokia, says: “The OSGi Service Platform is an open deployment platform which offers life-cycle management for mobile applications and services. It enables operators, enterprises, and mobile device and application manufacturers to dynamically extend the platforms features after manufacturing. For example, an IT Manager can install new APIs or applications to employees’ existing mobile devices over-the-air independent of the mobile phone model. This increases efficiency for enterprises and offers new business opportunities for

developers. The OSGi Service Platform also enables the new business model of client *middleware;* this model will become a driving force in the industry".

In the OSGi specifications, vendors and platform operators find a solution to the software manageability problem. Mobile software developers will find a rich, extendable environment on which to base their business and other applications.

**serve@home**

Siemens is providing a complete range of household devices that are prepared for home automation. The appliances are linked via a power-line interface; no new cabling is required. A central gateway running an OSGi Service Platform provides the aggregated services to the user. It is thus possible to verify that the oven was really switched off when you are driving away on vacation, start the oven to bake the fresh croissants while lying in bed on Sunday morning, or be notified of required maintenance. The advantage of using OSGi technology is clear: drivers and scenarios can be updated and added after the initial sale. Not only can drivers for the appliances be updated and added, but also interfaces to portable devices or standards like UPnP.

**Eclipse**

Eclipse is an open-source Integrated Development Environment (IDE) that originated inside of IBM as the n-th successor to Visual Age. Eclipse has an innovative architecture that is built around a very small extensible runtime core. All functionality, including the Graphical User Interface (GUI), the workbench, the compilers, and the support tools, consists of plugins that can be managed separately. This powerful IDE began as a development environment, but it soon became clear that its extensible core is very useful for building rich client platforms. For example, the IBM Workplace Client Technology is based upon Eclipse.

One requirement, however, was not fulfilled in Eclipse 2.0. Plugins could only be installed and updated with a restart. Eclipse could have developed dynamic plugins in a proprietary way but decided to follow the standards route with the OSGi Alliance. This allows them to take advantage of mature tools and knowledge. The new 3.x version, therefore, is built upon an OSGi Service Platform. As Jeff McAffer, one of the lead Eclipse developers, says: "The adoption of the OSGi framework has been a major step along the path to a fully dynamic Eclipse platform".

Eclipse has taken a large step by adopting a mature and well-tested dynamic component model with an OSGi Service Platform. With the OSGi Service Platform, they can now deploy and update plugins on the fly without restarting.

**BMW and the OSGi Service Platform in the 5 Series**

The automobile of the future will increasingly become an integral element within a network of information and services. ConnectedDrive is an innovative, all-encompassing concept from the BMW Group, which connects the system 'automobile' with the outside world and road traffic processes to provide the driver with as much information as he needs and wishes for in an individualized, ergonomic form. The information supplied is drawn from telematics services, online services, and driver assistance systems.

BMW uses the OSGi specifications as the base technology for its high-end infotainment platform in several high-end models. BMW is interested in the possibility to centralize functionality and, at the same time, to have the option to upgrade functionality in an easy way. This will reduce complexity by reducing the number of control units in the car. With that, it's possible to save costs by sharing a platform instead of having many. Additionally, it becomes easier to manage different configurations for different markets, even across different product lines.

For a company like BMW that sells in many foreign markets, it is vital that the technology used is not proprietary but standardized and that implementations are available from several vendors.

### Other Use Cases

There are many more products that are based on the OSGi Service Platform which are just as interesting as the previous use cases. Unfortunately, there is insufficient space in this whitepaper to discuss them all. The OSGi Alliance, however, maintains fact sheets with use cases on OSGi technology based solutions and products at http://www.osgi.org.

# Key Features of the OSGi Service Platform

### Software Component Management

The most common remark of members during their first OSGi Alliance meeting is: “We were building something very similar to the OSGi technology when we discovered the OSGi specifications”. Invariably, the motivating factor for them to build their own framework was to be able to maintain their software components and configurations. Using a standard OSGi environment not only simplifies this process, it also allows the usage of standard tools and patterns.

The OSGi Framework handles the life-cycle management of applications and components. Therefore, it provides the following functions:

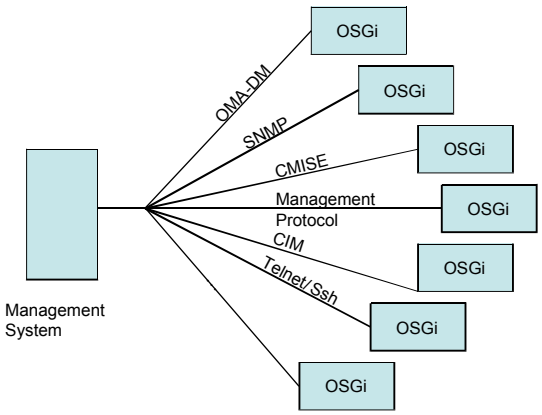
- A packaging format for the applications. The OSGi specifications provide the Bundle format. Bundles are applications packaged in a standard Java ARchive (JAR) file, which format is fully compatible with the ubiquitous ZIP files.
- Install a bundle. The bundle must be prepared, and the diverse components installed in the OSGi Framework, so it is ready to be executed.
- Start/stop a bundle. Installed bundles can be started and stopped in an OSGi Framework. Starting a bundle makes certain resources available, stopping a bundle cleans up. In an OSGi Service Platform, all applications are started in the same JVM, thereby saving memory, resources, and CPU cycles.
- Update a bundle. The OSGi Framework first stops existing applications, after which their resources are cleaned up. Code is unloaded and replaced with the updated code. After the code is updated, the bundle is restarted. This all happens without restarting the JVM.
- Uninstall a bundle. At the end of a bundle’s life, the code and its resources are removed from the system.

Today, when a device leaves the factory the functions and features of that device are defined and fixed. Having the capability to manage the code afterwards, allows different parties (platform operators, manufacturers, enterprises, and the end users) to customize the device to their needs. This saves cost and enables new types of applications as well as enhancing the relations with the customer.

### Remote Component Management

The OSGi Service Platform is specifically designed for devices that can operate unattended or under control of a platform operator. These are the devices that need remote management. Managing devices remotely requires a protocol. Selecting an appropriate protocol is difficult because there are so many choices: SNMP, CMISE, CIM, OMA DM, and more.

*Management protocols*




The OSGi Alliance decided that no management protocol can be preferred over others because no protocol is suitable for all cases. The OSGi Alliance therefore chose an architecture that provides a management API to be used by an authorized bundle. This authorized bundle can then map a protocol to API calls.

This is a very powerful concept that offers the same interoperability as a standard protocol. However, the benefits of this concept are not always immediately obvious. The benefit of a standard protocol is that any device can be managed by any management system. With the OSGi remote management architecture, the same advantage is achieved with an API and bundles which provide a higher level of abstraction with respect to a protocol. As long as a management system can provide a standardized OSGi protocol bundle, it can operate with any manufacturer's device, regardless of the underlying protocol.

By not specifying a protocol, the management system can optimize for the carrier characteristics. For example, a mobile phone with GPRS has high latency, expensive bandwidth, and is not always online. A residential gateway is always online, has low latency, and near zero cost bandwidth. Obviously, a protocol suitable for a mobile phone might not leverage the more privileged position of the gateway. Vice versa, a protocol that works well for a residential gateway does not work for a mobile phone. The OSGi management model therefore allows both industries to use their optimal management protocol without wreaking havoc on the interoperability.

The flexible OSGi remote management model allows the OSGi Service Platform to be used in many different deployment scenarios. For example, the mobile industry and the automotive industry are converging on OMA DM as a standard management protocol for mobile devices. The OSGi management architecture maps very well to this management protocol. Vendors develop OMA DM bundles that allow any conformant OSGi Service Platform to be managed by any OMA DM management system.

**Secure Execution Environment**

Executing code on a networked device requires the device to be protected. Therefore, the OSGi specifications offer a comprehensive security model.

The first level of defense is the VM with its security mechanisms. The JVM is designed to restrict the capabilities of a Java application during run time. Dangerous constructs, like pointer manipulations and unchecked access to arrays, are removed from the programmer's vocabulary. This choice makes it impossible to use, for example, buffer overruns to attack the environment. Buffer overruns are exploited by malicious viruses like Melissa, MSblaster, and many others that attack Microsoft Windows-based computers. Buffer over-run attacks are just not possible in Java.

The second level of defense is the safety features of the Java language. Java access modifiers declare the permitted callers of the code. This access control mechanism allows multiple Java applications to reside in the same VM and cooperate while providing a protection shield for code that should be kept contained.

The third (optional) level of defense is Java 2 code based security. Each bundle can be limited in its capabilities by giving it permissions for only those resources that it should have access to.

The final level of defense is the OSGi Framework that strictly separates bundles from each other. Bundles need appropriate permissions to interact with other bundles on the OSGi Service Platform.

Delegating the management policy to the platform operator allows the OSGi Service Platform to be used in many security scenarios, as well as scenarios where system security is handled outside the OSGi Service Platform.

The OSGi and Java security architectures are a comprehensive model that allows the platform operator to create an environment that is probably more secure than any other environment in the market today.

**Cooperation Between Applications**

There are many Java application server models available, such as MIDP 2, J2EE, Avalon, PicoContainer, JMX, and others. Except for the OSGi Service Platform, all these models provide a closed container in which the application runs in isolation. Applications can use libraries from the run-time environment but they are not able to provide no-overhead integration with other applications.

In contrast, the OSGi Service Platform allows bundles to *contribute* code as well as *services* to the environment. Contributing code is important because it allows libraries with shared functionality to be downloaded. In the closed container model, each application must carry all its code, even if it is already present elsewhere in the device environment. With the download of libraries, applications can share common code, making applications smaller. The power of this model is demonstrated by the Apache Software Foundation's selection of OSGi technology for some of its future container work.

For example, many enterprise applications require Java Messaging Service (JMS), a communication API to the enterprise back-end system. This is about 30-40 KB of code that must be included in each MIDP application because MIDP does not provide this functionality. This means that if there are six enterprise applications running, up to 200 KB is wasted. This expensive memory is saved in an OSGi Service Platform because a JMS library can be installed once and then shared by all enterprise applications.

To simplify collaboration, the OSGi Service Platform provides a lightweight publish, find, and bind *service* model for services inside the JVM with the OSGi Framework service registry. This supports loosely coupled application designs using a *service-oriented* architecture (SOA).

In the OSGi specifications, the word service has a very precise meaning, indicating a mechanism that allows one bundle to provide functionality to other bundles. An OSGi service is an object from one bundle that is made available to other bundles. Services can represent large-grained services such as the Http Service, or they can be fine-grained like a Bluetooth device that is discovered nearby.

The implementations of the service objects can be provided by different vendors; one vendor can offer different service object optimizations while another adapts it to different devices. The OSGi service model allows complicated applications to be split up in small services that are composed into a constellation for a specific task. For example, an enterprise application could have multiple UI components for different Graphic User Interfaces (GUI), a business component, a database component, and a number of

communication components. The deployer of this enterprise application can then mix the different components to match the target environment.

Cooperative applications simplify the development of mobile applications, as well as making the applications smaller and more flexible.

**Commercial Off the Shelf Components**

A standardized environment has the advantage that many parties can cooperate without extra communications. OSGi Alliance member companies like Espial, Gatespace Telematics, IBM, ProSyst Software, and others already deliver many basic building blocks off the shelf. Collectively, they have implemented virtually every popular protocol in existence as an OSGi bundle. The excellent characteristics of the cooperative models allow these vendors to develop small bundles that provide a set of highly cohesive functions that work seamlessly with other bundles.

The market of OSGi technology based Commercial-Off-The-Shelf (COTS) software components is small today, but is growing rapidly. This market is attracting many developers, as is demonstrated by the overwhelming success of OSGi technology in the Open Source community. This competition will keep the prices of common components low while maintaining high quality and choice.

**Simplified Deployment**

Installing an application in an environment is a difficult problem that is often underestimated. Java applications are notoriously hard to install because the language attempts to abstract from the environment as much as possible. Issues include which VM to use, how to configure the application correctly with the environmental parameters, and how to ensure the application can be started, stopped, and monitored.

The technologies provided by the OSGi Alliance simplify the process of deployment. Once an OSGi Service Platform is installed in an environment, deploying applications is smooth because the rigorously standardized OSGi Service Platform hides the differences of the underlying environments.

Using the OSGi Service Platform for deployment pays off because it can give more flexible deployment schemes and prevent the discrepancies in the underlying JVM and operating system from being visible to the applications. These features minimize deployment problems significantly.

**Multi Vendor Interoperability**

A unique selling point of Java is that its evolution is driven by many companies that are providing implementations of the same libraries, tools, and JVMs. This vibrant eco-system provides optimizations in many directions while maintaining compatibility for application developers. This is also the case for implementations of the OSGi specifications. The OSGi Alliance creates specifications and test suites that allow interoperability between different implementations.

For example, future generations of smart phones will contain an OSGi Framework. Complex applications or libraries written for a Nokia phone will then run unmodified on a Motorola phone.

Multi-vendor interoperability gives the platform operator as well as the manufacturer a choice. The operators can buy components from different vendors and run them on the same service platforms. The resulting competition increases quality and lowers prices.

**Dynamic Nature**

One expert in component-oriented programming called the OSGi specifications “Java’s best-kept secret”. His enthusiasm for the OSGi Service Platform is driven by the dynamic nature of the OSGi Framework. Installing a new bundle, registering a new service, or updating an existing bundle does not require a restart of the JVM. The affected

components are informed of the new state and adapt accordingly. For example, when a bundle with a servlet is updated, the servlet is automatically unregistered and the new version of the servlet is immediately made available through the web server.

Hot updates have long been available to large telephony systems, using proprietary technology, to assure 24/7 availability. The OSGi Service Platform brings standardized hot update technology to small, embedded devices, desktops and servers, enabling these devices to run continuously, even when their software is configured, updated or augmented.

**Policy Free**

The OSGi Service Platform provides many mechanisms but does not dictate how those mechanisms must be used. It allows considerable freedom for deployers to define their own policies. The remote management architecture with its pluggable management agent is a prime example. The flexible security policy that allows both unmanaged systems and walled gardens is another example. The OSGi Service Platform allows the operator to establish its policy using the mechanisms provided by the Service Platform.

Policy free leaves the platform operator in charge. The OSGi Service Platform can be used in a completely managed environment like a mobile phone, as well as running on the desktop without any supervision other than the user.

**Certification**

The OSGi Alliance has created a certification program to assure the compliance of Service Platforms and OSGi-specified services. Only OSGi Alliance members' implementations can be certified. These members can charge a fee for their implementations to offset their investments.

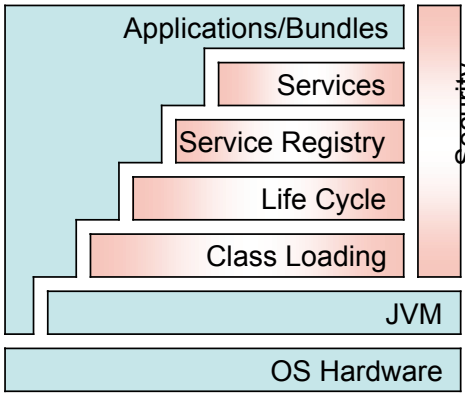
The advantage of certified implementations is that applications do not have to be written for a specific vendor's implementation. If an application runs on one vendor's service platform, it should run unmodified on another vendor's service platform.

Certification allows developers to defer the decisions for a particular vendor to late in the development process. Certification enables compliance, which promotes competition and thus lowers prices and improves quality.

# Architecture

The OSGi specifications have a number of layers as depicted in the following figure.

*OSGi Architecture*



The OSGi Service Platform provides a computing environment for applications, called bundles, which execute together in a single JVM. Bundles can be installed, updated, and uninstalled dynamically. Installed bundles find a rich environment to execute in.

The following sections describe the different layers that are necessary to provide this environment.

### JVM

The OSGi specifications are based on a JVM because that was the most logical choice when the OSGi Alliance was founded. The Java environment provides all the required features for a secure, open, reliable, well supported, mature, rich, and portable computing environment. Interestingly, today many other languages are becoming available on the Java VM, making the VM a portable environment and not just a language.

Today a possible candidate might be Microsoft .NET because it can provide similar features. However, the balance is still in favor of Java because Microsoft .NET is only available from one source. A widely accepted open standard, for as many environments as the OSGi Alliance is targeting, requires an open, multi-vendor platform.

### OSGi Framework

Running multiple applications in a single JVM creates a number of sharing and coordination issues that must be addressed. The OSGi Framework is the entity that is responsible for addressing these issues. It has a number of distinct responsibilities that are explained in the following sections.

### Modularization

The main contents of the bundle are the class files. Class files are the executable part of a bundle. In Java, classes are grouped in a package. Packages have unique names like, for example, `java.lang`.

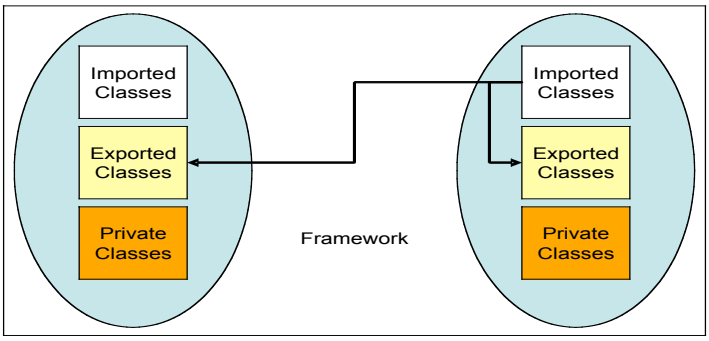
The first problem the OSGi Framework must handle is how to treat these classes with respect to other bundles. Should these classes be kept private or should they be shared with other bundles?

Restricting access to the classes to local only (bundle private) is the easiest solution. This model is pursued in other application server standards like Java 2, Enterprise Edition (J2EE) and Mobile Information Device Profile (MIDP). However, sharing classes allows bundles to contain libraries for other bundles, a key advantage with respect to memory footprint. Being able to rely on resident functions (or download them on demand) allows bundles to be much more flexible and smaller.

Sharing classes is the best solution. However, sharing classes means that multiple bundles can provide the same class, potentially causing version incompatibility between the same classes of different bundles. A related problem is the dependency on other bundles. It is also possible that different sets of bundles use the same class but require different versions of that class. The OSGi Framework handles all these problems in a rigorously specified and deterministic fashion.

Every bundle can export and import packages. Exporting means that packages (a set of classes) are made available to other bundles. Importing means that packages need to be available (exported) from other bundles. If multiple bundles are exporting the same package (with a different version, for example), the Framework selects an appropriate version for each bundle that imports that package.

*OSGi Class Sharing*




A package is always exported with a unique version. The importer can specify a range of versions that it can accept. The Framework tries to minimize the number of exports but it supports multiple *class spaces* where multiple versions of the same class can be in use at the same time. To prevent clashes, the Framework thoroughly verifies that a bundle cannot inadvertently get class cast exceptions.

If a bundle exports a Java package and is subsequently uninstalled, then the OSGi Framework ensures that importers are restarted so that they can bind to a new package exporter. This whole process is transparent to the bundles because it happens when they are stopped.

Managing Java class-loading in a VM for multiple independent bundles is not a trivial task. Compliance between different vendors in this complex area is achieved by specifying in minute detail what the OSGi Framework must do in all cases.

**Life-Cycle Management**

Installing a new bundle in a remote JVM (and uninstalling it) provides the basis for networked services. Installing a bundle has two important aspects:

- The file format of a bundle
- Access to the install function

A bundle is typically stored in a Java Archive, also called a JAR. Every JAR contains a Manifest. A JAR Manifest stores information about the contents of the JAR in headers. Some headers are pre-defined by the JAR Manifest specification but the total set of headers is extendable and the values can be localized. The OSGi Alliance has defined a number of additional Manifest headers to allow the JAR file to be used in an OSGi Service Platform.

Access to the bundle installation function is done with an API that is available to every bundle. This may seem odd because it implies that only a bundle can install another bundle. This raises a problem of self-reference: how is the primordial bundle installed? This bootstrap problem is solved with the Initial Provisioning specification or by using command line parameters when starting an OSGi Framework implementation.

The API of the OSGi Framework is defined in the `BundleContext` object. The OSGi Framework supplies this context object to the bundle when it is started. The context object has a number of methods to install new bundles and list existing bundles.

The `installBundle` method takes a URL or InputStream as a parameter. The OSGi Framework inspects the headers and the code and installs the code in the OSGi Service Platform. After the bundle is installed, a `Bundle` object is returned.

Once a bundle is installed, it can be started, but before classes in the bundle can be executed, the bundle must be *resolved*.

A bundle can have a number of dependencies on the external environment. For example, a bundle can import a Java package. The Framework analyzes these dependencies and binds the dependents before the bundle is given control. These bundle dependencies can be indirect and cyclic. Resolving one bundle may initiate resolving other bundles as well.

The installed bundle is given control by instantiating a class from the bundle. This class must implement the `BundleActivator` interface, which has `start(BundleContext)` and `stop(BundleContext)` methods. The name of the instantiated class is specified by the Manifest header `Bundle-Activator`.

The start method must return quickly to allow other bundles to be managed as well. If the bundle requires background processing, it must create threads for these tasks. The Framework moves the state of the bundle to *active* once the start method has returned.

During the active life of a bundle, many events happen in its Framework. Other bundles might get stopped, updated, uninstalled, installed, and started. This could mean that certain dependencies of the bundle can no longer be fulfilled. For example, the exporter of a Java package can become uninstalled.

These changes never affect the bundle directly nor can they happen at a random moment. If a bundle is uninstalled, its exported packages remain available as long as there are other resolved bundles depending on them. However, the Management Agent can initiate a *refresh* action. This action stops all affected bundles, removes the dependencies, resolves again, and restarts the earlier stopped bundles. Bundles, therefore, need not be concerned about their imported and exported packages changing on them while they are not paying attention. A bundle is always stopped before any of its package dependencies are changed.

The start and stop state of a bundle can be recorded persistently. If a bundle is started persistently and then the OSGi Framework is shut down, the bundle will be automatically restarted when the OSGi Framework is restarted. However, this does not have to happen immediately after starting the Framework because each bundle can be assigned a *start level*. When starting up, the Framework can transition between start levels, automatically starting and stopping bundles as required by the start level and persistent start state.

The life-cycle management is the most prominent feature of the OSGi Framework. Life-cycle management provided as an API enables reliable remote management without mandating protocols. It provides the necessary mechanisms to allow remote management in a wide variety of management models.

**Service Registry**

Active bundles can use all standard Java mechanisms to implement their functionality, just like any other Java environment. However, an interesting aspect of the OSGi Service Platform is its dynamic nature. A bundle can suddenly become active, providing a function that could be useful for other parts in the system. For example, a bundle is installed that needs to be visible on the local web server. The web server bundle and the new bundle must now suddenly cooperate.

Other scenarios found in home automation, mobile and vehicle environments are also dynamic. A mobile phone might have an accessory plugged in. A residential gateway could detect a new media server or a car could detect the presence of a Bluetooth mobile phone that comes into range.

Each example above could be implemented with existing Java patterns of registering listeners with domain objects, like, for example, adding a Mouse Listener to a Frame. However, this results in hundreds of private registries that all have their own idiosyncrasies. This could be acceptable if the OSGi Service Platform was a static environment that only required structures to be built dynamically, like most Java environments. However, it is *not*. The system is not only built up, but it must also be possible to remove parts of it without disrupting the overall environment. This is a crucial

difference, with far-reaching consequences, between an OSGi application and a traditional Java application!

These dynamics require that all the hundreds of listener registries must become aware of the OSGi Service Platform dynamics. A bundle that is stopped should automatically be removed from each registry where it has a listener registered. Vice versa, listeners must be aware that they are no longer registered because their subject is gone.

The OSGi Service Registry was developed to make this problem easy to manage. It dynamically links different bundles together while tracking their state and dependencies. With the Service Registry, bundles can:

- Register objects with the Service Registry.
- Search the Service Registry for matching objects.
- Receive notifications when services become registered or unregistered.

Objects registered with the Service Registry are called *services*. Services are always registered with an interface name and a set of properties. The interface name represents the intended usage of the service. The properties describe the service to its audience. For example, the OSGi Log Service would be registered with the `org.osgi.service.log.LogService` interface name and could have properties such as `vendor=acme`.

The Service Registry is an *in-memory registry*. Registrations are dynamic and are dependent on the execution state of the bundles. The OSGi Framework automatically unregisters all services from a stopped bundle, notifying all its dependents. The Service Registry is designed to have very low cost per registered service. The use of the Declarative Service specification can reduce even this low cost by delaying loading classes from a bundle until services from the bundle are actually used; often achieving an efficiency that a more traditional environment cannot.

Discovering services is done with notifications, or by actively searching for services with specific properties. A simple, but powerful, filter language is used to select exactly the services that are needed. A utility class, the Service Tracker, makes it easy to write code for this highly dynamic environment.

The Service Registry is innovative and therefore not familiar to many Java programmers. It has no counterpart in other Java application environments. It is absent in these other environments because they lack the dynamic aspects that make OSGi technology so powerful. It is this dynamic nature, installing and *uninstalling* bundles, as well as the connection to the real world, that make the Service Registry such a powerful aid to the bundle programmer.

The Service Registry enables the OSGi Service Platform to support applications built using a service-oriented architecture (SOA). It allows application programmers to develop small and loosely coupled components, which can adapt to the changing environment in real time. The platform operator uses these small components to compose larger systems. The Service Registry is the glue that binds these components seamlessly together.

**Security**

One of the goals of the OSGi Service Platform is to run applications from a variety of sources under strict control of a management system. A comprehensive security model, present in all parts of the system, is therefore a necessity. The OSGi specifications use the following mechanisms:

- Java 2 Code Security
- Minimized bundle content exposure
- Managed communication links between bundles

Java 2 Code Security provides the concept of Permissions that protect resources from specific actions. For example, files can be protected with File Permission classes. Permission classes take the name of the resource and a number of actions as parameters. Each bundle has a set of permissions. This set of permissions can be

changed on the fly. New permissions are immediately effective once set. However, this permission assignment can also be done prior to, or just in time during the install phase.

Permissions are roughly used as follows: When a class wants to protect a resource, it asks the Java Security Manager to check if there is permission to perform an action on that resource. The Security Manager then ensures that all callers on the stack have the required permissions. Checking the callers on the stack protects the resource from attackers. For example, if M calls T and T accesses a protected resource, both M and T need to have access to the resource. This is a very secure strategy that prevents much mayhem.

The access modifiers in Java classes are used to protect access to code. Classes, methods, and fields can be made private, package private (accessible only by classes in the same package), protected (accessible by sub-classes) or public. The OSGi Service Platform adds an extra level of module privacy by making packages only visible within the bundle. These packages are freely accessible by other classes inside the bundle, but they are not visible to other bundles.

Package-sharing between bundles is a possible attack route for malicious bundles. The OSGi specifications therefore contain Package Permissions to limit exports and imports to trusted bundles. Package Permission is a fine-grained permission that allows importing or exporting for a specific package, or for all packages.

Another security mechanism is Service Permission. This permission gives bundles the possibility to register, or get a service, from the Service Registry. Service Permissions are extensively used to ensure that only the appropriate bundles can provide or use certain services.

The security mechanisms in the OSGi Service Platform provide the operator of the Service Platform with powerful tools to control the security operation of the device.

# Standard Services

The following sections give a short overview of the OSGi Service Platform Release 4.1 services. More information can be found at the OSGi web site (www.osgi.org) and in the OSGi Service Platform Release 4 Core Specification and Service Compendium. These services are optional and can be deployed as desired by a given solution. Note that each service is defined abstractly and is independently implemented by different vendors.

**Framework Services**

The OSGi Framework may provide a Permission Admin service, a Conditional Permission Admin service, a Package Admin service, a URL Handler service, and a Start Level service. These services are optional and direct the operation of the Framework.

- *Permission Admin* – The permissions of current or future bundles can be manipulated through this service. Permissions are activated immediately once they are set.
- *Conditional Permission Admin* – This service extends the Permission Admin with permissions that can apply when certain conditions are true or false at the time the permission is checked. This enables for example permissions that depend on: prompting the user, the signer of the bundle, or the location of the bundle. The number of conditions can be extended with custom code. For example, a mobile operator could permit certain operations when the mobile is in a specific region.
- *Package Admin* – Bundles share packages with classes and resources. The update of bundles might require the system to re-calculate the dependencies. The Package Admin service provides information about the actual package sharing state of the system and can also refresh shared packages, that is break the dependencies and recalculate the dependencies.
- *Start Level* – Start Levels are a set of bundles that should run together or should be initialized before others are started. The Start Level service sets the current start level, assigns a bundle to a start level, and interrogates the current settings.

- *URL Handlers* – Java 2 provides mechanisms to add new scheme or content handlers to the URL class. However, the mechanisms is not dynamic because the operation to provide the hook for adding handlers can only be set once and handlers cannot be withdrawn. The URL Handlers service enables bundles to dynamically contribute new scheme or content handlers to the URL class.

**System Services**

System Services provide horizontal functions that are necessary in many systems. The Log Service, Configuration Admin service, Device Access service, User Admin service, IO Connector service, Event Admin service, and Preferences Service are examples of system services.

- *Log Service* – The logging of information, warnings, debug information or errors is handled through the Log Service. It receives log entries and then dispatches these entries to other bundles that subscribed to this information.
- *Configuration Admin* – This service provides a flexible and dynamic model to set and get configuration information.
- *Event Admin* – Provides a general and flexible publish and subscribe event mechanism. Supports both synchronous and asynchronous delivery of events.
- *Device Access* – Device Access is the mechanism to match a driver to a new device and automatically download a bundle implementing this driver. This is used for Plug and Play scenarios.
- *User Admin* – This service uses a database with user information (private and public) for authentication and authorization purposes.
- *IO Connector* – The IO Connector service allows bundles to provide new and alternative protocol schemes for the Generic Connection Framework (`javax.microedition.io` package).
- *Preferences Service* – A service that provides access to hierarchical database of properties. Similar to the Windows Registry or the Java Preferences class.
- *User Admin* – Provides access to user authorization and authentication information.
- *Device Management Tree* – Provides an abstract tree containing management information, services are used to provide the content for this tree. The concepts closely follow the concepts in the OMA DM protocol. The OSGi Alliance also developed a management tree for mobile devices.
- *Deployment Admin* – Provides a service that can deploy multiple artifacts from one file. A flexible scheme allows the artifacts to be processed by external *Resource Processors*.
- *Application Admin* – OSGi bundles are continuously running while more traditional models use a start-stop model for applications. The OSGi Application service provides a model whereby applications can register and get activated on demand.
- *Monitoring* – Provides a standard way for bundles to provide their performance data.
- *Foreign Applications* – The OSGi Framework is very suitable to deploy non-OSGi applications like Midlets into. The foreign application access specifications offers other application models an API to access OSGi specific functions.

**Protocol Services**

The OSGi Alliance has defined some services that map an external protocol to an OSGi service.

- *Http* Service – The Http Service is, among other things, a servlet runner. Bundles can provide servlets, which become available over HTTP. The dynamic update facility of the OSGi Service Platform makes the Http Service a very attractive web server that can be updated with new servlets, remotely if necessary, without requiring a restart.
- *UPnP* – Universal Plug and Play (UPnP) is an emerging standard for consumer electronics. The OSGi UPnP service maps devices on a UPnP network to the Service Registry. Alternatively, it can map OSGi services to the UPnP network.

**Miscellaneous Services**

- *Wire Admin* – Normally bundles establish the rules to find services that they want to work with. However, in many cases this should be a deployment decision. The Wire Admin service therefore connects different services together as defined by a configuration. The Wire Admin service uses the concept of a Consumer and Producer

service that interchange objects over a wire. The Wire Admin service is supported with the Position, Measurement and State utility classes.
- *XML Parser* – The XML Parser service allows a bundle to locate a parser with desired properties and compatibility with JAXP.

**Programming Support**

The OSGi Service Platform is a very dynamic environment, much more dynamic than traditional Java application models because bundles can be installed and uninstalled at any time. This dynamism creates additional complexity for the programmer. The OSGi specification addressed this problem with the following utilities:

- *Service Tracker* – The Service Tracker significantly simplifies the handling of services by providing a class that tracks the services for the application. The application provides 3 methods that are called: for existing or new services, when the properties of a service are modified, and when the services are unregistered or the tracker is closed.
- *Declarative Services* – The Service Component Runtime is a subsystem that can read an XML declaration from a bundle with service registration and service dependencies. The Service Component Runtime will then register the declared services on behalf of the bundle. The bundle is only initialized when the declared service is actually needed by another bundle. Declarative services can significantly reduce the footprint of a device because only actually needed bundles consume resources.

# Future directions

The adoption of the OSGi specifications in different industries is creating a demand for new services (vertical) as well as new features in the OSGi Framework (horizontal). Therefore, new services and framework extensions are the different areas in which enhancements will take place.

On the horizontal level, there will be a need for more control over the low-level details of the Framework. Desktop applications like Eclipse, and the use of the OSGi specifications in Enterprise applications, will inevitably drive new requirements. However, this should not significantly increase the footprint of the core Framework.

There is ongoing work in the following areas:

- *Remote Management* – The OSGi Service Platform has been carefully crafted to be agnostic of the management protocol. However, the market might require a preferred standardized management protocol to minimize the number of options and improve inter-operability. An abstraction is planned that supports OMA DM very well.
- *Power Management* – Mobile devices require careful management of the power consumption. Today, many peripherals provide the capability to reduce their power consumption without completely loosing their context. The OSGi Alliance is investigating standardizing the interfaces to enable optimized power management.
- *Web Services* – Amazon, Google, Microsoft.NET, and many more Web services are becoming more popular each day. The OSGi Service Platform is an excellent platform for Web Services. Its superb dynamic update facilities, the rich software environment that Java offers, and cooperative facilities that the OSGi Service Platform offers are an ideal combination with Web Services.
- *Application Servers* – The OSGi Service Platform is an excellent Java application server. Research is being performed to decide if the OSGi Framework is applicable in J2EE environments. The OSGi Service Platform's superior life-cycle management architecture could add significantly easier remote management to J2EE servers.
- *Connectivity* – Embedded applications are more and more confronted with portable devices. iPods, mobile phones, PDAs are just a few examples of devices that people expect to collaborate with their car and home computers. Managing this complexity is one of the main goals of the OSGi Alliance.
- *Distribution* – The current specifications assume a single VM for the service collaboration model. There is an increasing interest in using the service model for distribution.
- *Native code* – Java has many advantages over other environments but many systems require integration with legacy code or code that has special requirements.

Many companies would like to adopt OSGi but need a standardized solution for this integration problem.

# Conclusion

The OSGi specifications provide a comprehensive operating environment for Java applications. Companies that adopt the specifications will benefit on many different fronts.

- *Development cost* – Devices with the OSGi Service Platform become easier to develop because of the component architecture. Components can be developed with smaller teams and/or bought externally, saving significant cost. Additionally, components developed for one product are likely to be reusable for other similar products. This reuse not only saves development time/cost, it also saves learning time/cost.
- *Customization* – The hardest problem for many manufacturers is device customization for different markets and customers. This customization can make the number of configurations skyrocket. Software configuration management quickly becomes a significant development cost in these situations. Components can simplify this process significantly and at the same time increase customization without additional costs.
- *Deployment cost* – The standardized environment that the OSGi Service Platform provides to bundles minimizes deployment problems significantly.
- *Remote Management* – The OSGi Service Platform provides policy-free remote management capabilities to a device. The clean architecture allows the device to be managed with a suitable protocol such as OMA DM.
- *Unification* – The increasing popularity of the OSGi specifications enables the unification of completely different environments. In the future, applications written for a residential gateway can also run inside a vehicle, PDA or desktop. This will enable a large market for OSGi applications.
- *Providing the glue* – The OSGi Service Platform can provide the glue for embedded devices to gracefully interact with their environment.

The number of adopters of the OSGi specifications is increasing rapidly. The new business opportunities, the cost savings, and the technical advantages are providing great incentives to incorporate OSGi technology into your products.

If this white paper has raised your interest, please contact the OSGi Alliance at www.osgi.org for more information or send a mail to marketinginfo@osgi.org.

The OSGi Alliance is a worldwide consortium of technology innovators that advances a proven and mature process to assure interoperability of applications and services based on its component integration platform. The alliance provides specifications, reference implementations, test suites and certification to foster a valuable cross-industry ecosystem. OSGi technology is delivered in many Fortune Global 100 company products and services. Member companies collaborate within an egalitarian, equitable and transparent environment and promote adoption of OSGi technology through business benefits, user experiences and forums. For more information, visit http://www.osgi.org.

**HOW TO REACH US:**

**OSGi Alliance**
Bishop Ranch 6
2400 Camino Ramon, Suite 375
San Ramon, CA 94583 USA

Phone: +1.925.275.6625
E-mail: marketinginfo@osgi.org
Web: http://www.osgi.org