Using Apache Felix: OSGi best practices

Marcel Offermans
luminis®

ApacheCon Europe 06

About me

- Marcel Offermans
- Software architect at luminis®
  - Consultancy & product development
  - Over 4 years of experience with OSGi
- Committer on the Apache Felix project
- marcel.offermans@luminis.nl

My name is Marcel Offermans. I am a software architect at luminis® (http://www.luminis.nl/), which is a small company based in Arnhem, The Netherlands. We do software consultancy and product development, predominantly for large companies that create (hardware) products containing embedded and often distributed software. I have over 4 years of experience using OSGi in commercial projects and I'm a committer on the Apache Felix project.

Agenda

- Birds-eye view of the OSGi framework
- Building, partitioning and provisioning
- Dependency management
- Software testing
- Leveraging standard OSGi services
- Extending the service registry: JMX & SOAP
- Embedding OSGi

I'll start off with a very short, birds-eye view of OSGi. In talks earlier today, Richard Hall has already explained how Apache Felix works and what OSGi is so I won't be going into great detail about that.

This talk is about best practices, and these are mainly things we have learned by using OSGi in commercial projects. Since there are many different areas, from embedded to desktop to enterprise, this won’t be an exhaustive overview.

You will learn a bit about different build environment, how to partition software into bundles, how to provision your software and how to dynamically manage service dependencies.

Then we'll look at software testing and see what generic and specific tests should be used.

Next up, we'll look at a some of the standard OSGi services and their application. Because systems hardly ever exist in isolation, we'll have a look at some remote aspects such as management and SOAP access and how to easily integrate those into the framework.

Finally we'll have a look at some of the possibilities that exist for embedding OSGi.

Birds-eye view

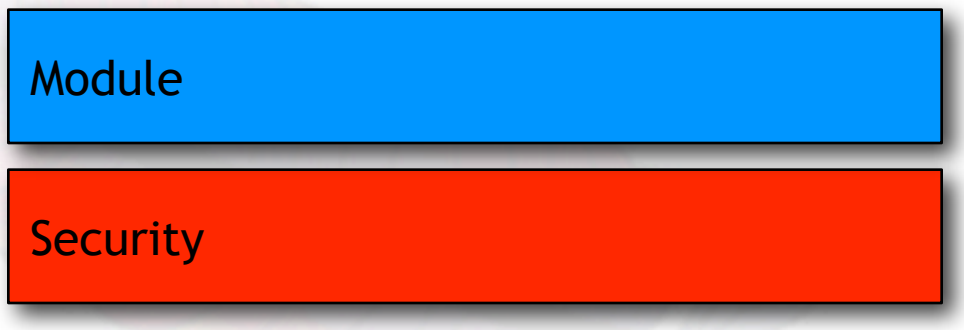
- Security layer:
  - similar to the Java 2 security model;
  - bundles can be signed;
  - PermissionAdmin service to manipulate permissions.

In short, OSGi is a component based, service oriented framework consisting of 4 layers.

First there's the security layer, which is similar to standard Java security, so I guess most of you are already familiar with that. You can use policy files to determine what software bundles can and cannot do. Furthermore, there is an option to sign bundles, similar to the way you sign jars for Java Web Start. Finally, there is a service called PermissionAdmin, which allows you to dynamically manipulate permissions. You can use this to change policies on the fly or add new policies for newly installed components.

Birds-eye view

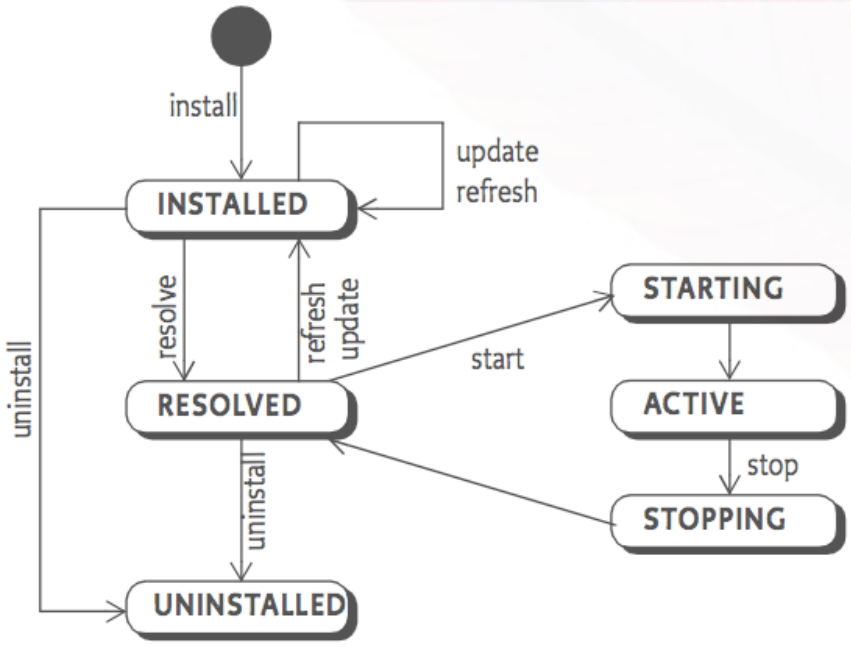
- Security layer
- Module layer:
  - unit of deployment is the bundle;
  - bundles can export packages, specifying a version for each package;
  - bundles can import packages, specifying a version range;
  - resolved at run-time.

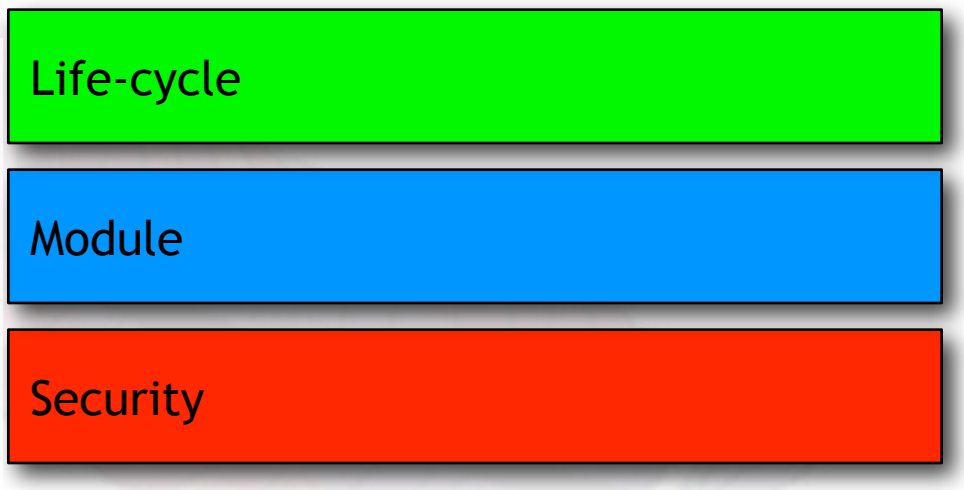


Second up is the module layer. At the module layer, the software is grouped into bundles, where each bundle contains zero or more java packages and optionally some other resources or even native libraries. Some of these packages are exported, meaning they can be used by other bundles, others are private and only visible within a specific bundle. Packages that are exported by one bundle, can be imported by another. The exporter specifies a version number for a package. The importer can specify a range of versions it is compatible with. At run-time the framework will resolve such dependencies. All this allows you to keep implementations private, only exposing API’s.

Birds-eye view

- Security layer
- Module layer
- Life-cycle layer:

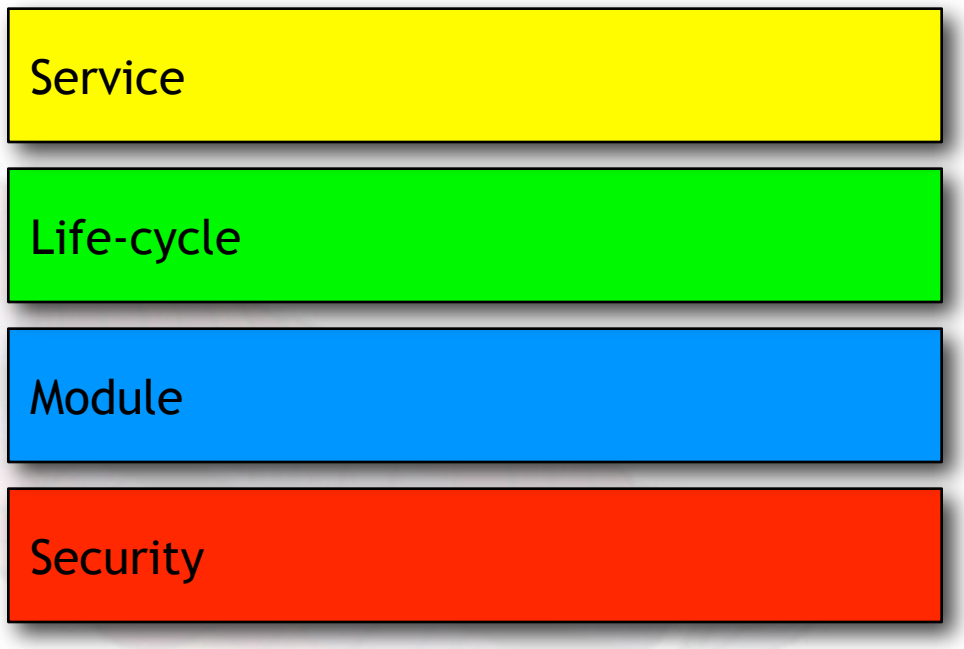




On top of that, the life-cycle layer determines the life-cycle of individual bundles. Basically, bundles can be installed, started, stopped and uninstalled. Furthermore, an installed bundle can be updated anytime. Finally, you can uninstall a bundle if you don’t need it anymore. The “starting” and “stopping” states have hooks for you as the bundle developer to actually make your bundle do something. By implementing the BundleActivator interface, the framework will invoke your start() and stop() methods. You will also get a BundleContext which is basically a reference to the API to talk to the container.

Birds-eye view

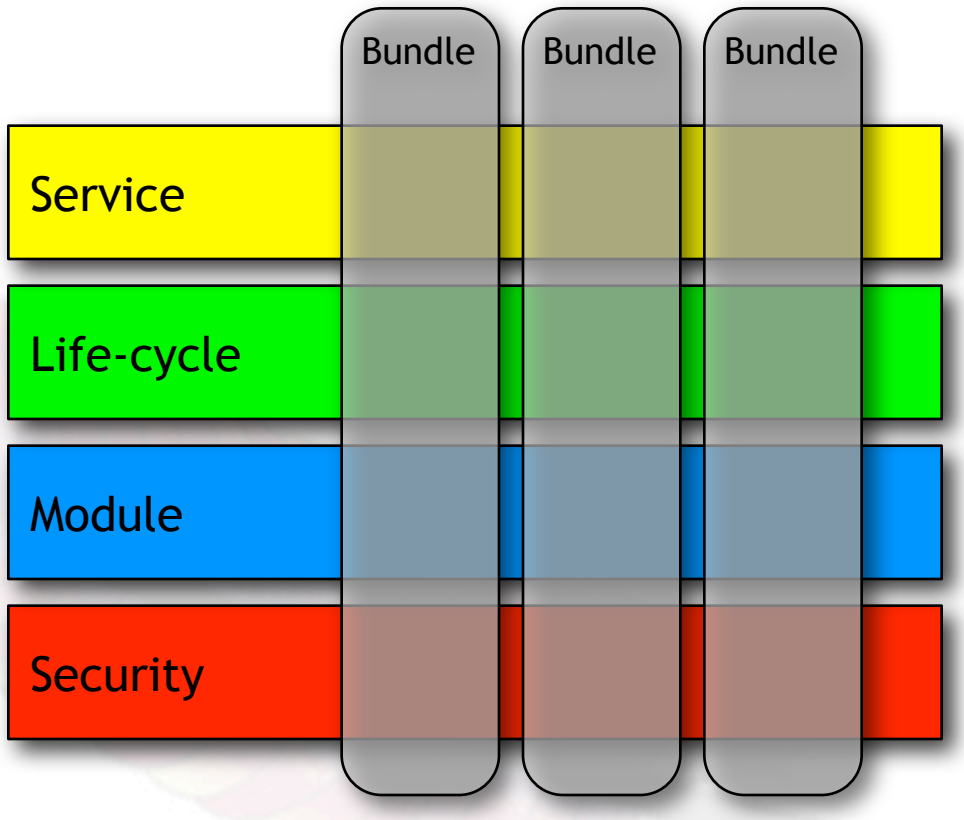
- Security layer
- Module layer
- Life-cycle layer
- Service layer:
  - registry;
  - publish;
  - query.



Finally, the service layer contains a registry where services are published. Each bundle can register any number of services in the service registry. A service is registered using the fully qualified name of its interface. Furthermore, you can add any number of properties to this service. Both the name and properties can then be used again by others that query the registry. A fast and powerful LDAP based query language is used for that.

Birds-eye view

- Security layer
- Module layer
- Life-cycle layer
- Service layer
- Actual services



On top of these four layers, which together form the container, you can install your bundles containing the actual services.

OSGi frameworks

- Apache Felix
  - Successor to Oscar
  - Over 4 years in development
- Gatespace Telematics' Knopflerfish
  - Open sourced
  - Commercial support
- Eclipse Equinox
  - Foundation for Eclipse

Let’s have a look at the different available OSGi implementations. Apart from several commercial implementations, there are three major open source ones:

Apache Felix, which is new to Apache, but is the succesor to Oscar, a project that has been under development for over 4 years now. We are still in incubation right now, but we are working hard to get out of that because we feel we have a very stable framework already.

Gatespace Telematics' Knopflerfish, an open source project that was initially a commercial product. In fact, Gatespace is still offering commercial support for it, but the framework itself is free.

Equinox, the foundation on which Eclipse is build. Around version 3.0, Eclipse actually switched to OSGi when they were looking for a more dynamic, standard replacement for their plugin system. At that time they considered both JMX (which JBoss uses too) and OSGi. They eventually chose OSGi and although at first they did not make a big deal about it, they are now offering it as a separate project. Actually, a lot of the new features introduced in OSGi release 4 came from Eclipse, some of them mainly to support the migration of “legacy code”.

Of course, at an Apache conference the politically correct thing would be to say to just use Felix and forget about the rest, but actually we have had a lot of benefit by developing and testing on multiple frameworks.

Build System

- Ant
  - use macro features for repetitive tasks
  - use “mangen” to automatically create imports
- Maven
  - used within Felix
  - one bundle per “project” model
- Eclipse
  - one bundle per project

There are three major build systems you can use with OSGi:
Ant, which has been traditionally used by many Java projects. When using Ant, make sure to structure your project in a smart way so you can use macro features for all repetitive tasks. Also, there is a tool called “mangen” which you can use to automatically generate imports based on byte code analysis of your actual code. The advantage of using Ant is the fact that you are free to organize your project in any way you want (but that can also be a disadvantage if you make a mess of it).
Maven, or actually Maven 2, has been used in Felix. It recommends you to use a “one bundle per project” model, which is a very reasonable way of working, even though we have traditionally used a “one subsystem per project” model, creating multiple bundles during packaging. The Maven 2 OSGi plugin is available, but is still being perfected, and we’re talking to the Maven developers too to make sure OSGi is supported well in Maven.
Finally, you can use Eclipse as a build environment. It too uses a “one bundle per project” model. Creating, building, running and debugging bundles is very easy, because you can run your bundles directly in the IDE and use things like hot code replace. On the other hand, there is a disadvantage of having to use Eclipse to do builds, so I would not recommend it for more complicated projects.

Partitioning software

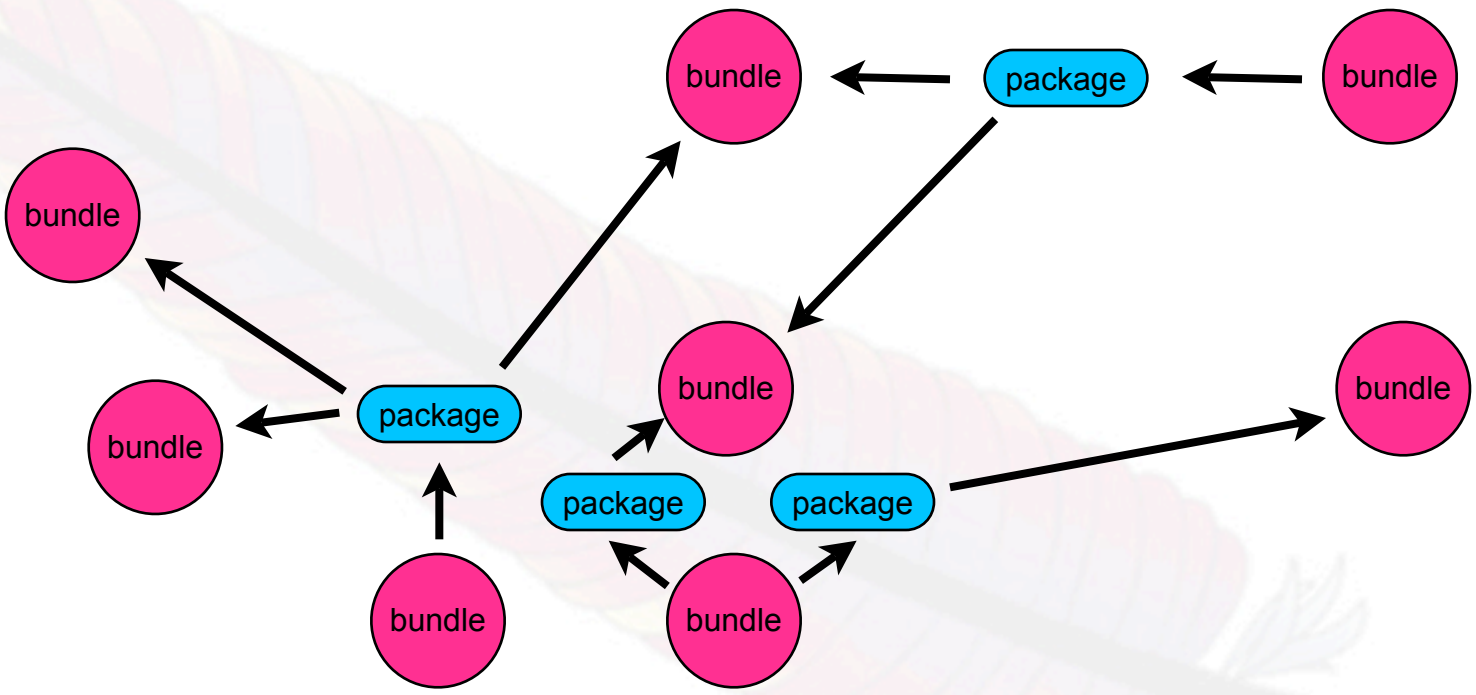
- Minimizing dependencies
- Considering rate of change
- Minimizing complexity
- Maximizing architected flexibility

Basically, I'm not going to say anything new here, because the points I make here are common sense when designing software. However, it is interesting to see how the OSGi framework enables us to really use partitioning even at run-time to keep the system modular. In "traditional" environments you'll often see that although the design is modular, in practice some of these modularization is broken because in general each piece of code has full visibility of all other code on the classpath.

Other important things to consider when drawing these models is the rate of change of components. For example, for most applications, the user interface will change far more often than the business logic, so keeping those separate is desirable from the point of view of updating components.

Partitioning software

Package dependencies at the module layer

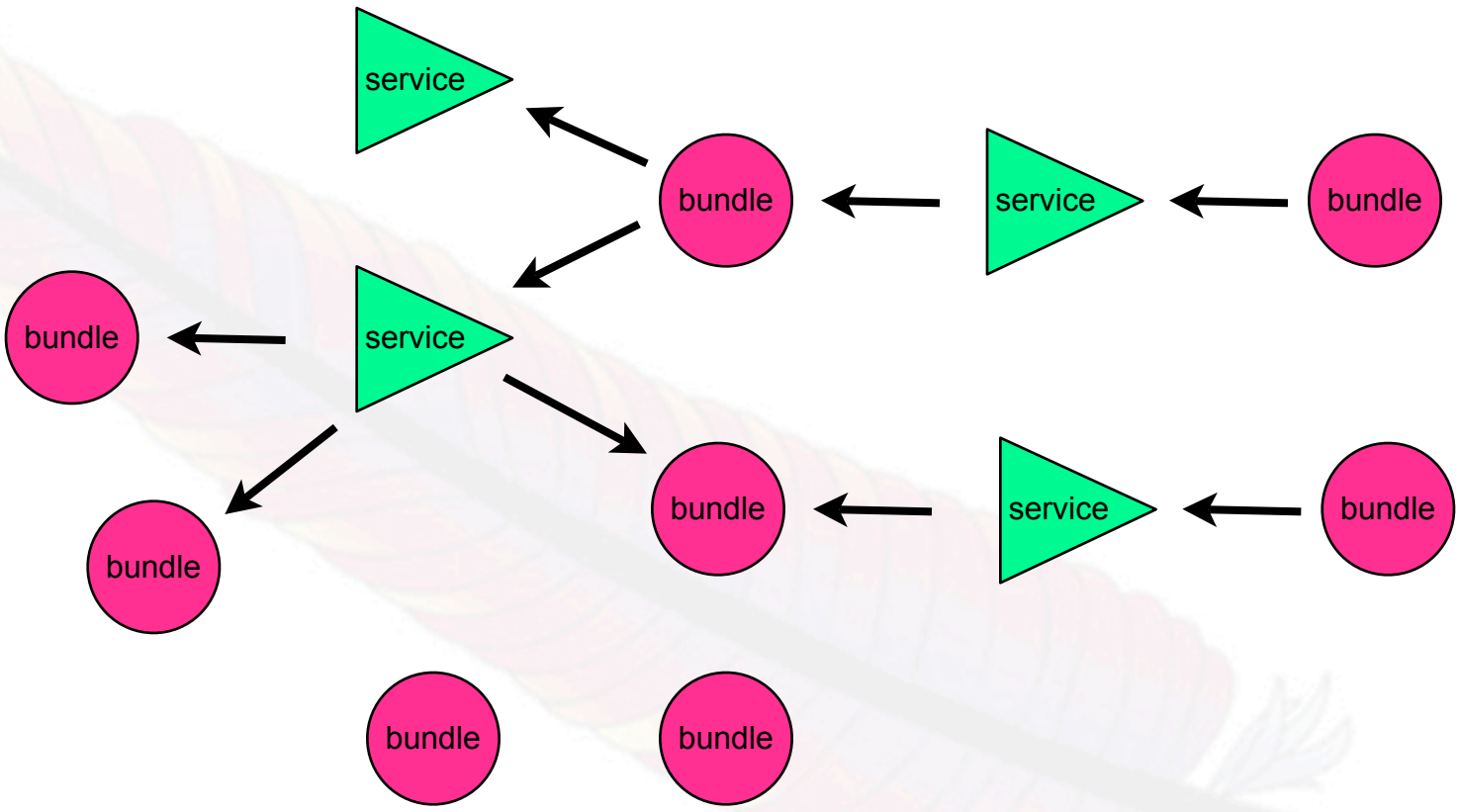


So, because OSGi makes the dependencies and components explicit, it’s fairly easy to get an overview of the services, packages and components or bundles in a design. Creating a graph of these is very helpful, because it gives you a very quick overview of all dependencies. This allows you to minimize those dependencies and really evaluate the way you’ve created components.

You can create two graphs, this is the graph of all package dependencies at the module layer. It clearly shows who imports and exports what.

Partitioning software

Service dependencies at the service layer



The second graph is arguably even more important and shows the service dependencies at the service layer. It’s obviously useful to overlay both graphs to get an overall impression of the dependencies.

Currently, there are no tools to automatically generate such graphs (either at design time or at run-time) but that’s definitely something we would be interested in because these graphs are very helpful to discuss the architecture of large systems and to get insight into their operation while testing.

Provisioning

- Don't think about provisioning as an afterthought!
- OSGi spec covers it:
  - Initial provisioning bootstraps your systems
    - allows you to use off the shelf OSGi hardware
    - connects to a provisioning server and installs the management agent
  - Management agent handles updates
    - installs the actual software
    - keeps system up to date

So we've talked a bit about partitioning, the next thing to consider is provisioning. Software provisioning is all about getting the right software bundles onto the right platforms. Especially for embedded systems, but also for others, it is worth thinking about how you are going to do this. In the past you often saw service engineers connecting their laptops to some machine, uploading new firmware over a serial line. Similarly, on the desktop, software companies would just send out new CD's every time a new version came out.

The first thing worth mentioning is that the OSGi spec covers provisioning. There is an "initial provisioning" system for bootstrapping your system. The pre-condition for initial provisioning is a device that already has a JVM and an OSGi framework installed, but no bundles. The mechanism describes how a framework should connect to a provisioning server and recursively get all the bundles necessary to run the so called "management agent". The management agent then takes over and communicates with the provisioning server to further install and update the required software.

For the embedded market this is very important, because it allows you to use "off the shelf" OSGi hardware which you can just power-on and your software will automatically install itself (usually a process which is done in the factory where the machines with embedded software are assembled).

Dependency management

- Design using POJO's
- Use import/export instead of “require bundle” to avoid too tight coupling
- Use a library, don't build it yourself! Available options:
  - service binder;
  - dependency manager;
  - declarative services;
  - iPOJO.

Up til now we’ve mostly talked about dependency management from the design and deployment point of view. Let’s now look at the run-time aspect. Summarizing dependency management, there are two types of dependencies: Package dependencies, which are explicitly declared in the manifest and resolved somewhere between the installation and the startup of a bundle. Second, there’s service dependencies, which are completely dynamic, since any service can appear and disappear at any point in time. This creates a very flexible system, but at the same time places a heavier burden on the developer, since he has to cope with changes in the availability of the services he uses. There are two types we can distinguish: required dependencies, which simply must be available, otherwise the component cannot do its job, and optional dependencies, which can be used if available, but are not necessary for the basic operation of the component.

The OSGi framework itself provides basic support for managing dependencies, but still leaves you as the developer with some repetitive work. Luckily, there are many available libraries to manage these dependencies for you. Of these, declarative services is part of the R4 spec, the others are simply bundles which can be used in any framework. In general, these libraries allow you to simply design your services as POJO’s and declare their dependencies and have them injected. The main differences between these implementations is how these dependencies are declared and injected. For example, the dependency manager uses a Java API whilst the service binder uses an XML file. We intend to start merging these different libraries into one though. The main point I want to make is that you can write your services as POJO’s, which is convenient and does not tie you to a framework like OSGi, so there’s no “standard lock-in” ;)

Software testing

- Unit tests
  - Build on your POJO's
  - Mock service dependencies
- Integration tests
  - Use a test framework
  - Create tests that validate the life-cycles
  - Test on multiple frameworks
- Continuous build
  - catch bugs early

From using POJO's it's only a short step towards testing, because when you can write your software using POJO's, it makes it a lot easier to test. Again, a lot of these tips are neither rocket-science, nor OSGi specific.

Using the POJO's you can write a lot of unit tests. The service dependencies can then easily be mocked. We have good experience using the EasyMock library, but there are others which will no doubt work just as well. For each test, simply inject the mocked services and you can easily write tests. Also, these tests will usually run very quickly, which for big projects is a great advantage. I've recently been involved in a large J2EE project that wanted to start using a continuous build process. The only problem, their build and test cycle took somewhere between 4 and 6 hours so every time their nightly build failed, they had only a couple of hours to fix the problem and restart a build during the day time if they wanted to check if the problem would really be gone once the next nightly build would run. This might be a bit extreme, but I've seen more projects that take a long time to go through all their tests.

So basically my first advice is to test as much as possible using unit tests. The next step is to invest in an automated integration test framework. You can use this to test the collaboration of a whole group of components, and to do bundle life-cycle tests: do all my bundles start, stop and update correctly. Since the OSGi framework itself is fairly lightweight, it's possible to simply start a whole framework, deploy some bundles and run a test. You can use something like JMX or RMI to remotely control the framework and get the results of the tests.

Finally, use a continuous build. This is simply another means of detecting problems as early as possible.

Leveraging standard services

- OSGi compendium – catalog of standard service descriptions
- Repository at bundles.osgi.org – over 1400 bundles, implement compendium and other services
- More and more projects are made OSGi compatible (often extending the standard manifest is enough)

In OSGi, there are several ways to use standard services and components.

First of all, the second part of the OSGi specification, called the compendium, contains a set of standard services. These cover areas from logging to configuration settings, from user management to a HTTP service. Often, multiple different implementations exist for any of these services, so there's plenty to choose from based on your specific requirements.

A great source for bundles is the bundle repository on the OSGi website. Currently it contains over 1400 bundles, which implement compendium and other services. There is a web interface to query the repository but even better, there is a bundle that allows you to query the repository from within an OSGi framework. Using that bundle, you can either manually or automatically install certain bundles from this repository. Because the repository also has knowledge about a bundle's dependencies, these get installat automatically too.

Looking into the future it will probably be possible to extend the repositories that Maven currently uses to include OSGi manifest information. This would even further increase the database of available pluggable components. At the moment, within Felix we're still working hard on creating all the necessary Maven plugins to create bundles and we're very open to cooperating with other projects, helping them expose their jars as bundles.

The OSGi compendium

Wire Admin

User Admin

Initial Provisioning

Log

Device Access

XML Parser

Service Tracker

Measurement and State

Preferences

Position

UPnP™ Device

Configuration Admin

Metatype

Declarative Services

HTTP

Event Admin

IO Connector

Execution Environment Spec

This is a quick overview of the services in the compendium. It's a broad mix, so we won't discuss all of them. Instead, we'll look at three fairly generic ones, which we've used in our projects a lot.

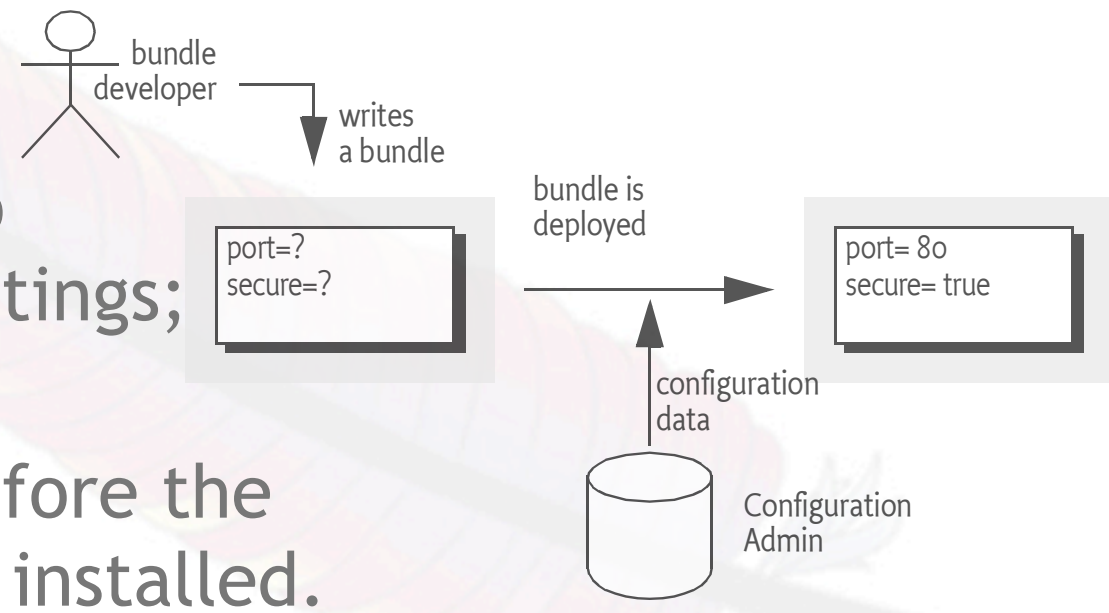
User Admin

- Used in any application that needs role based access control
- Provides: users, roles and groups
- Can authenticate users
- Can determine autorization for authenticated users
- Fairly easy to plug-in to HTTP, SOAP, RMI, JMX or anything else

First of all is UserAdmin. This is a service to do role-based access control. You can define users, groups and roles and perform authentication (is this user really who he says he is) and authorization (is this user allowed to perform this operation). There are several implementations of this service available. There is a relatively simple one that uses an object persistency mechanism to store data. There also is a version that talks to the Apache Directory Server (at least, there is one in development at the moment). Obviously, since the service interface is always the same, it’s fairly easy to switch implementations.

Config Admin and Preferences

- Configuration Admin:
  - contains externally configurable settings for a service;
  - allows management systems to configure all settings;
  - settings can be created even before the actual bundle is installed.

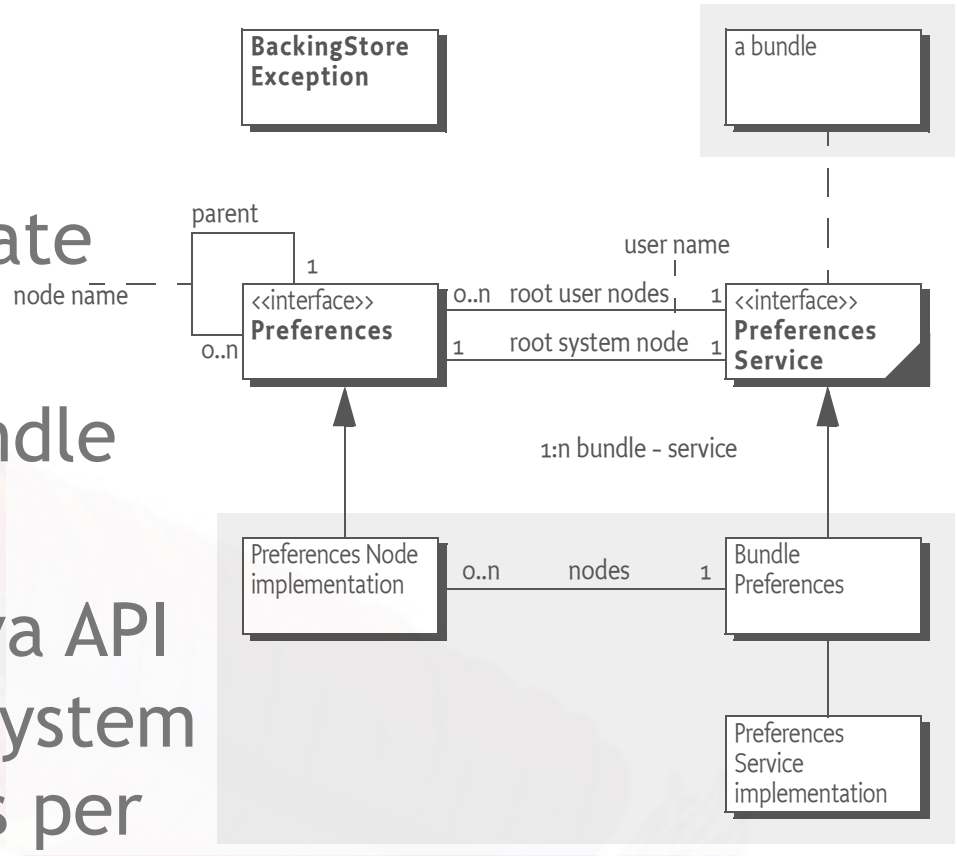


Then there's ConfigAdmin and Preferences. These are two services that at first glance seem to have some overlap, since both are concerned with some kind of "preferences" but upon closer examination, they are complementary:

ConfigAdmin contains all externally configurable settings for a service. For example, let's say we have a HTTP service. This is a service with several settings. You can specify the port to listen to, and if the service should use SSL or not. ConfigAdmin allows you to externally configure these settings, even if the actual HTTP service is not even installed yet. This allows a management system to centrally manage settings for all services.

Config Admin and Preferences

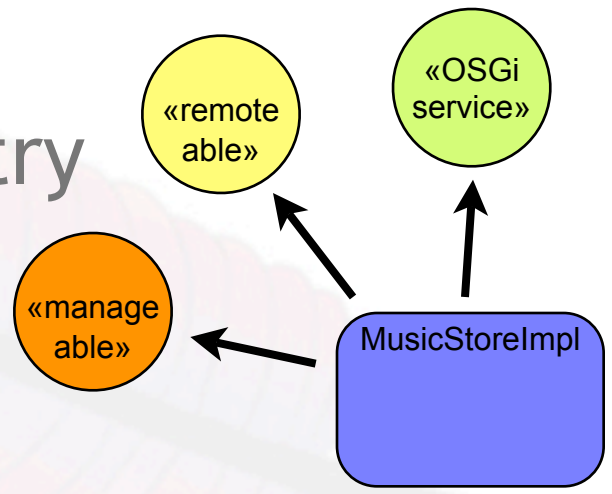
- Preferences:
  - contains bundle private settings;
  - is coupled to the bundle life-cycle;
  - like the standard Java API there is a notion of system and user preferences per bundle.



Preferences is very similar to the standard Java API with the same name. Each installed bundle has its own set of preferences (both user and system preferences, but the actual interpretation of what a user in the context of that bundle is, is left to the developer). These preferences are persistent, and they also “survive” updates of the bundle. However, there is no mechanism to externally manipulate them. So typically these preferences are used for internal settings. For example, imagine a bundle that shows a Swing GUI, then the preferences for this bundle could contain the window positions (per user if relevant).

Extending the service registry

- Plain OSGi: service implementation + interface(s)
- Add interfaces to the implementation
- Leveraging the life-cycle events of the service registry
  - Java 5 annotations
  - XML descriptors
  - Super interfaces
  - …



Next up, let’s discuss extending the service registry. Normally, in plain OSGi, you have a service implementation and an interface that you will register in the OSGi service registry. What you can do is use the event mechanism of this registry to notify your own listener. If you then decorate your service implementations with extra interfaces, you can give them to a factory and, for example, create JMX MBeans for a management interface. In the next slides, I’ll show some examples of what we’ve done.

Remote management with JMX

- Spec globally describes management, does not choose any specific technology
- JMX is the Java standard for remote management, so using it makes sense
- Built into Java 5, MX4J for many other cases

When you want to do remote management, the OSGi spec globally defines how you can do it, but it does not prescribe any specific technology. A lot of times, it makes sense to use JMX because it’s the standard Java way of doing management. We use Java 5, so we can take advantage of the built-in JMX implementation. Extending the service registry like explained before, we can generate MBeans automatically and publish them into the MBean server. For developers creating manageable views of services, this is a very easy model to use. We actually use annotations to provide the meta-data for the factory that will automatically generate the MBean.

Remote access with SOAP

- System rarely exists in isolation, therefore we need remoting
- SOAP can be plugged in easily
- We used XFire, but others have integrated Apache Axis too
- JSR-181 web service annotations @WebService

Another example is SOAP. The reason for using SOAP is that it is a great way to provide access to your framework from other platforms and environments.

We're using XFire, but others have integrated Axis too. In the end, we can simply annotate an interface with the @WebService annotation and have the XFire bundle pick that up and turn the interface into a SOAP interface. So all you need to do as a developer is tag the interfaces you want to have available remotely.

Embedding OSGi

- Embedding OSGi
  - in Java Web Start
  - in a servlet container
  - in a J2EE server
  - in OSGi
- OSGi embedding
  - makes existing frameworks more flexible:
  - Wicket, Cocoon, Spring (SPR-1802), (?), …

Another interesting subject is the embedding of OSGi. Basically there are two things you can do. Either you embed OSGi in some other container or framework, or you embed something else in OSGi.

OSGi itself can be embedded in other containers. For an open source desktop application I have created a Java Web Start application that bootstraps an OSGi framework that subsequently installed the rest of the application. For a customer of ours we're currently embedding OSGi in a servlet container to create a pluggable web application where you can update parts of the application without having to restart the whole web application. Also, there have been various examples of using an OSGi framework inside a J2EE application container.

The other way round, you can embed other applications in OSGi. There are several projects considering such a thing and I have personally experimented a bit with Wicket, creating pluggable web applications.

Talking to an embedded Felix

- WARNING: this is implementation specific!

```
public class Main implements BundleActivator {
    public static void main(String[] args) {
        (new Main()).init();
    }
    public void init() {
        Felix fw = new Felix();
        List activatorList = new ArrayList();
        activatorList.add(this);
        fw.start(propertyResolver, activatorList);
    }
    public void start(BundleContext ctx) {
        // context is your hook into the system!
    }
    public void stop(BundleContext ctx) {
        // indicating the framework is stopped
    }
}
```


Finally, this is an example of how to embed Felix in an application and how to communicate with it. Basically, you provide a BundleActivator that you pass to the framework when you start it. Your activator's start() method will then be called, allowing you to get hold of the BundleContext. This is your hook into OSGi.

Wrapping it up…

- Share some experiences!
- Any questions?
- Get on the Felix mailing list and ask!