RFP 121 - Subsystem Metadata and Lifecycle

Draft

21 Pages

~~*Text in Red is here to help you. Delete it when you have followed the instructions.*~~

~~*The <Document Title> and <Company Name> can be set from the File>Properties:User Defined menu. To update it onscreen, press F9. To update all of the fields in the document Select All (CTRL-A), then hit F9. Set the release level by selecting one from: Draft, Final Draft, Release. The date is set automagically when the document is saved.*~~ ~~***Note that FINAL DRAFT and RELEASE documents should only be released as PDF, NOT AS ODF DOCUMENTS***~~

Abstract

A mismatch exists between the OSGi concept of Bundles and how Deployers tend to think about Applications. The Application concept is much more coarse grained than the Bundle concept. This RFP introduces Subsystem Metadata to OSGi which will enable the Deployer to control the OSGi framework in a meaningful way.

0 Document Information

0.1 Table of Contents

0 Document Information......2
0.1 Table of Contents......2
0.2 Terminology and Document Conventions......3
0.3 Revision History......3

1 Introduction......4

2 Application Domain......4
2.1 Related OSGi Specifications......7
2.1.1 OBR (RFC 112)...... 7
2.1.2 Nested Frameworks (RFC 138)...... 7
2.1.3 Initial Provisioning...... 7
2.1.4 Deployment Admin...... 7
2.1.5 Application Admin...... 7
2.2 Terminology + Abbreviations......8

3 Problem Description......8
3.1 Scope......8

4 Use Cases......9
4.1 Deployer wants to check the applications in the OSGi framework......9
4.2 Deployer wants to be able to easily uninstall an app and everything that came with it......9
4.3 Impact analysis for upgrade of a bundle......9
4.4 Starting and Stopping Applications......9
4.5 Upgrading and patching an application......10
4.6 Purging of bundles......10
4.7 Installation of applications......10
4.8 Development Integration 10
4.9 Establish the Application Fingerprint......11
4.10 Tooling integration......11
4.11 Rollback of deployments......11
4.12 Sharing of bundles across applications......11
4.13 Configuring subsystems......11

5 Requirements......12

5.1 Subsystem Modeling....12
5.2 Dependency Management....13
5.3 Administration....13
5.4 Runtime....14
5.5 Development Support....14

6 Document Support....16
6.1 References....16
6.2 Author's Address....16
6.3 End of Document....17

0.2 Terminology and Document Conventions

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "NOT RECOMMENDED", "MAY" and "OPTIONAL" in this document are to be interpreted as described in 6.1.

`Source code is shown in this typeface.`

0.3 Revision History

The last named individual in this history is currently responsible for this document.

Revision	Date	Comments
Initial	March 6th, 2009	David Bosschaert, initial pre-draft version.
	March 2009	Tim Diekmann, first review and additions
0.3	March 12th, 2009	David Bosschaert, incorporating initial feedback from Austin F2F.
0.4	March 19th, 2009	David Bosschaert, incorporating feedback from Yan Pujante
0.5	April 8th, 2009	David Bosschaert, added use-case 10 and chapter 2 (domain description).
0.6	May 25th, 2009	David Bosschaert, feedback from Montpellier Bundlefest F2F and added use-cases 11 & 12. Document title changed from Application Metadata & Lifecycle to Subsystem Metadata & Lifecycle.
0.7	June 30th, 2009	David Bosschaert, initial requirements added.
0.8	August 7th, 2009	David Bosschaert, changes as discussed during the Dublin F2F.
0.9	September 2nd, 2009~~September 2nd, 2009September 2nd, 2009~~	Guillaume Nodet, incorporate feedback from the calls.~~Guillaume Nodet, incorporate feedback from the calls.Guillaume Nodet, incorporate feedback from the calls.~~

Revision	Date	Comments
1	September 24th, 2009	Guillaume Nodet, finalize document for review.
1.1	October 1st, 2009	Glyn Normington, describe an initial subset of the requirements and define the term *scope*.
1.2	October 14th, 2009	David Bosschaert. Moved requirement 7 to deferred list as agreed during concall on October 5th.
1.3	October 15th, 2009	Guillaume Nodet. Defer requirements 29 and 30, exclude subsystems scoping from the scope of this RFP, fix REQ 25, fix minor typos.
1.4	October 19th, 2009	Glyn Normington. Moved requirements 7, 29, and 30 to the supported list as agreed during concall on October 19th. Clarified requirement 7. Noted that “deferred” requirements are actually omitted from this RFP, but may be included in future RFPs.

1 Introduction

The OSGi platform provides a very appealing deployment platform for a variety of applications. Especially in Enterprise Scenarios an application may consist of a large amount of bundles. However, deployers tend to think in terms of applications being deployed rather than bundles, so there is a mismatch in concepts provided by the OSGi framework and the concepts familiar to the Deployer.

This document describes the problem with some use-case and declares the requirements that a solution to this problem needs to satisfy.

2 Application Domain

When System Administrators, Deployers and other people think about software they typically think in terms of systems performing a certain business function. A System Administrator's job might be to make sure that these system stay up and running. The sysadmin doesn't have intimate knowledge of how these systems are constructed, but (s)he does have knowledge of the applications that compose the system. There might be an ordering application, a payment application and a stock checking application. The sysadmin knows that these applications exist and that they must be monitored to ensure that they are in a healthy state.

Other people may think of a system yet on a different level. The CIO of the company may have a higher level view. He sees the system described above as a single application unit: the company's web shop and views all of the applications that together run in his organization as 'The System'.

On the other end of the spectrum, a developer might be tasked by implementing the payment system. This system might in turn be built up of other applications. There might be a currency conversion application, a credit card charge application and an electronic bank transactions application.

Bottom line is that what one person sees as an application, might be a mere component to the next person in the chain. Application is a nestable concept and as such it is referred to by the more general word ‘Subsystem’. What the people described above have in common is that a pure OSGi bundle is typically not what they view as an application. An OSGi bundle is a component used to build applications, but applications themselves, no matter from what level you look at it are always bigger than a bundle.

So from a conceptual point of view, applications are nestable. However, this does not mean that the runtime realization of these nested applications needs to exactly follow the nested structure. In OSGi systems, applications are built up from OSGi bundles, and although it would be desirable to view an OSGi system in a way that makes sense to the person looking at it, this does not mean that the bundles themselves need to be nested as well. The application view could be a virtual, organizational layer over a potentially flat bag of bundles that realize the applications at runtime. In some cases layering might be needed to enforce separation but in other systems the layering might not be used.

An application level view over an OSGi system might look like Figure 1.

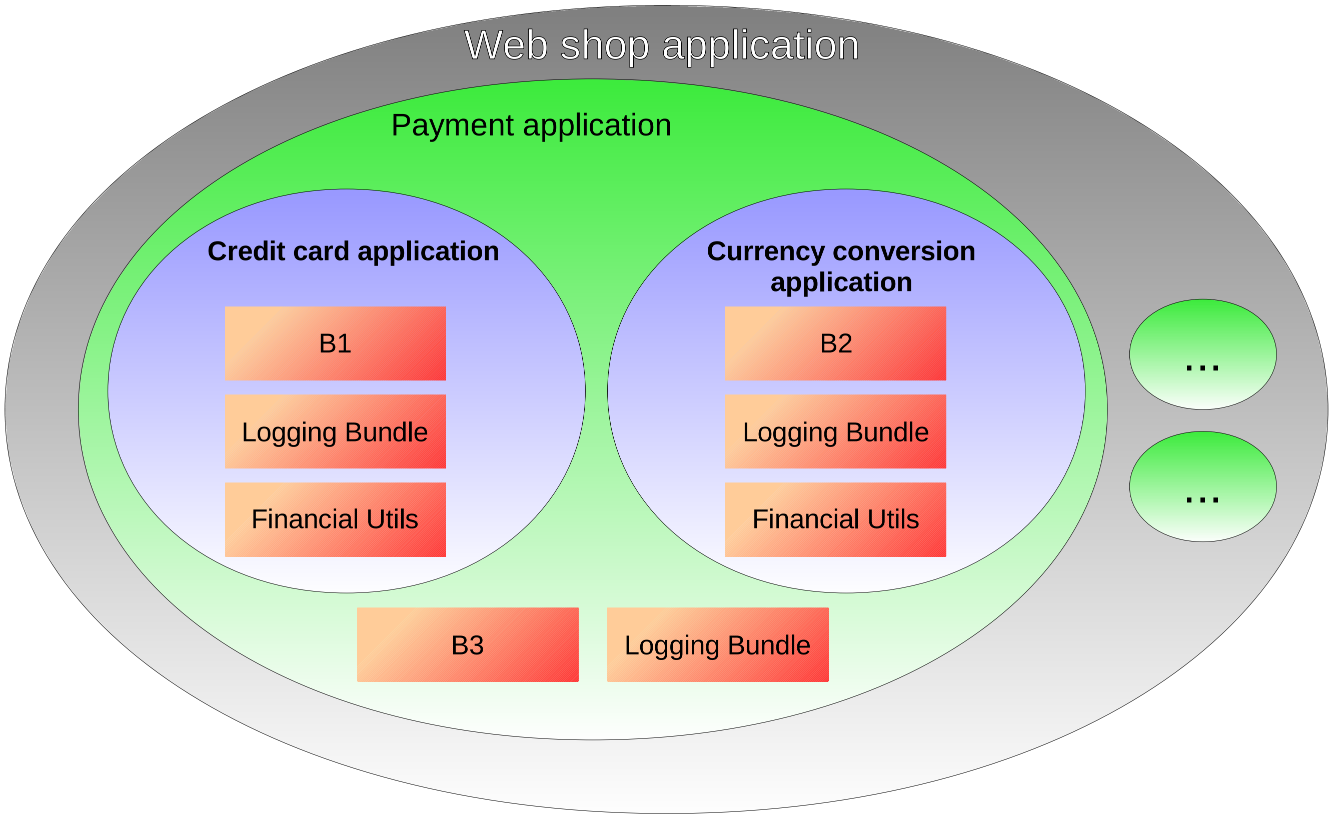

Figure 1: Application level logical view of a deployment

At runtime, the bundles composing this application might be deployed as a flat structure where the 'Logging Bundle' and 'Financial Utils' bundle are shared. This way of deploying the application is very efficient with regard to memory footprint. See Figure 2.

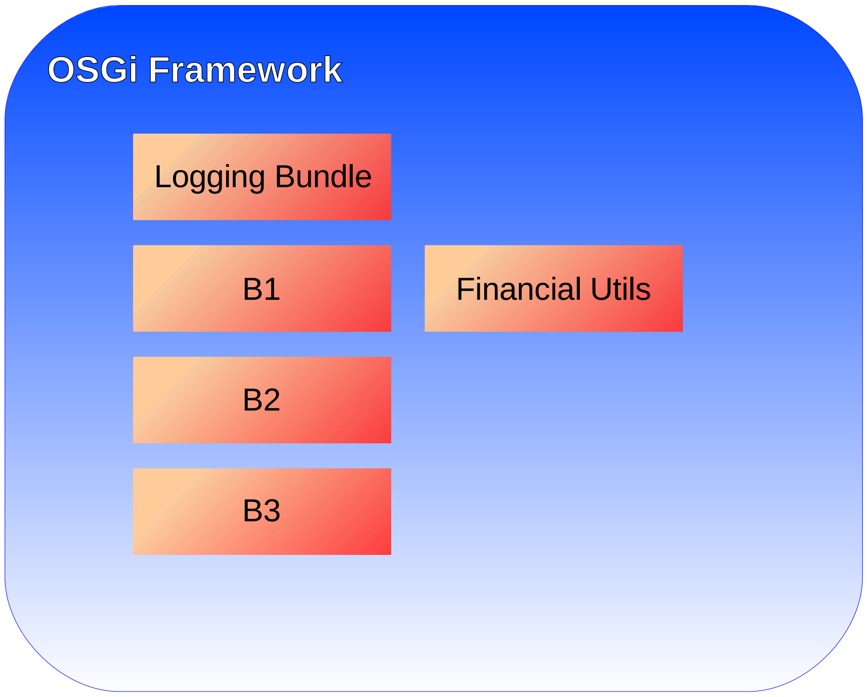

Figure 2: Runtime realization with sharing

Framework Nesting as defined by RFC 138 could be used to isolate certain sub-systems while still sharing other bundles, like the logging bundle. This provides another runtime deployment option for the same system, providing more isolation. See Figure 3.

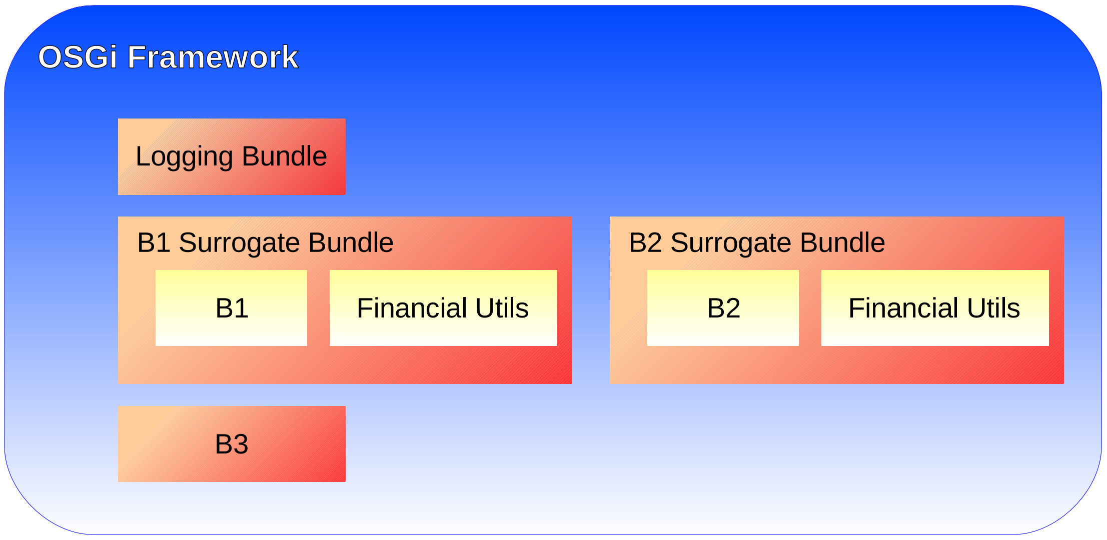

Figure 3: Runtime realization with less sharing and more isolation

Besides the applications installed in an OSGi framework, there are also bundles installed that are considered part of the OSGi framework infrastructure, but at runtime these bundles could become part of the application. Consider a bundle providing the logging service or a bundle providing the HTTP Service. OSGi Services are a great architecture for loose coupling, fostering sharing and re-use.

2.1 Related OSGi Specifications

2.1.1 OBR (RFC 112)

RFC 112 describes the OSGi Bundle Repository. This repository could provide an excellent place to store all the bundles required to deploy an application. It would therefore be important that any new designs resulting from this requirements document fully integrate and leverage the OBR.

However, using an OBR may not be right for every use-case so it should be optional. In some contexts development might start out relatively loose, using artifact repositories such as an OBR. Over time during the development process the dependencies in the project might harden and ultimately an OBR may not be appropriate any more. In a completely hardened situation dependencies might be obtained from a local directory created by an installer process or possibly a fixed corporate web site.

2.1.2 Nested Frameworks (RFC 138)

Nested Frameworks as proposed in RFC 138 provide a way of isolating or grouping subsystems within a single OSGi Framework. A nested framework could be used to map to an application at runtime, but there may also be implementations that do not require isolation so use of Nested Frameworks should be optional.

2.1.3 Initial Provisioning

The Initial Provisioning spec does not relate to an application concept at all. It rather concerns about how to deploy initial bundles on a device. This specification is unrelated to this RFP, but mentioned here to confirm that there is no potential overlap.

2.1.4 Deployment Admin

The Deployment Admin specification is about defining a file format to ship and deploy applications. Such an application is built up of multiple bundles and additional resources. Patching of applications, is also supported by this specification.

An additional feature provided by Deployment Admin is the capability to roll back a deployment. This very useful. See use-case 4.11.

A major drawback of the Deployment Admin specification is that it explicitly prohibits sharing of bundles, which, besides preventing memory efficiency, could cause real problems when two applications use the same third-party bundle, as installing the same bundle twice is not allowed. Even different versions of the same bundle can not be installed simultaneously with Deployment Admin.

2.1.5 Application Admin

Application Admin provides the concept of an Application in OSGi. This specification could potentially be the basis of a runtime API into the Application Metadata and Lifecycle system. It would most likely have to be extended to support all the use cases. Missing in the Application Admin spec is the Application Metadata, which should be declarative and available both inside the running OSGi framework as well as outside it to support OSGi tooling for subsystems.

Concepts missing from Application Admin:

- No file format to describe an application, purely API based
- No impact analysis for doing upgrades
- No interaction with OBR
- No application fingerprinting
- No runtime application extent analysis (what bundles beyond the core app set are used by this app)

2.2 Terminology + Abbreviations

- *Scope* – a unit of visibility, similar to a programming language scope, containing bundles and possibly other artifacts. The purpose of a scope is to prevent its contents from being accidentally interfered with or depended upon by other artifacts outside the scope. Unscoped artifacts in the service platform are visible to artifacts in a scope. If nested scopes are supported, artifacts in a containing scope are visible to the artifacts in a contained scope.

3 Problem Description

In today's OSGi framework, what is deployed is typically just a large set of bundles. To a person not familiar with the details of the design of these bundles it is often unclear what function these bundles perform and how they are related.

Some bundles might provide shared infrastructure (e.g. a bundle providing the OSGi Log service), while other bundles might together provide an application function (e.g. a set of bundles together implementing a web-based shopping application). Today it is not possible to find out from a high level what applications are installed in the OSGi framework. The only information available is the list of bundles.

The OSGi framework needs to be enhanced with a mechanism that makes it possible to declare applications. An application has a name meaningful to the deployer. It is typically composed of a number of key bundles and services, plus their dependencies. When the deployer requests the list of installed applications, he will get a manageable result that is typically much shorter than the list of installed bundles.

The deployer also needs to be able to perform actions on the application level. Installing, uninstalling, starting, and stopping should be possible on the level of an application.

A developer may wish to think in terms of an application consisting of a set of bundles and may wish to declare those bundles as belonging to the application.

3.1 Problem Scope

Graphical tools are out of scope for this RFP.

In scope would be the metadata needed to define the applications plus an API that would enable actions at the application level.

Out of scope would be the definition of any mechanism to restrict interactions between subsystems. Scoping of subsystems might be implemented using RFC 138 or another mechanism, but this RFP does not aim at defining such a mechanism.

4 Use Cases

4.1 Deployer wants to check the applications in the OSGi framework

A Deployer wants to see what applications are installed in an OSGi framework and wants to see the status of these applications. He expects to see that all the applications in the framework are properly running, but might find out that one of the applications wasn't able to start.

Today the deployer only see a large list of bundles, most of which don't mean much to the deployer.

4.2 Deployer wants to be able to easily uninstall an app and everything that came with it

The Deployer has installed an application. He has done this by providing the application metadata that defines the application in the OSGi framework via the OBR. The bundle has pulled in all its dependencies automatically from the OBR and is now properly running in the OSGi framework.

After a while the Deployer decides to uninstall the application. He expects that all of the bundles pulled in by the application that aren't used by any other application are automatically removed.

Note that some bundles might be shared across applications. These bundles can only be deinstalled once all of the applications using the bundle are uninstalled.

4.3 Impact analysis for upgrade of a bundle

The Deployer has received an update for a bundle that is being used in his OSGi system. He wants to upgrade the system, but first wants to find out what the impact of the upgrade is. For this the OSGi system needs to provide him with a list of the applications that are using the bundle.

The next thing the Deployer wants to know is whether he can do a no-downtime upgrade or not. If the bundle is used by the applications via OSGi Services, the Deployer can simply install the new bundle along side the old bundle and then uninstall the old bundle. However, if an application is linked to the old bundle via 'Import-Package' or Require-Bundle, the application needs to be temporarily stopped.

It may also be the case that upgrading a bundle introduces new dependences or removes old dependencies. The OSGi system upgrade process needs to consider these transitive dependencies.

Note the impact analysis system needs to be pluggable to support dependencies that are outside the realm of OSGi. E.g. a bundle might have a proprietary dependency on a database server.

4.4 Starting and Stopping Applications

The Deployer decides to upgrade a bundle in an application. He has found out that the application depends on this bundle via Import-Package so the application needs to be stopped in order to perform the update. Once stopped, the old bundle can be removed and replaced by the new bundle.

The Deployer has run the impact analysis as mentioned in use-case 4.3 to establish that the application depends on this bundle. Since the configuration of the OSGi framework could have changed in the mean time, the solution needs to provide a mechanism that allows the application of the bundle update to be done atomically given a particular state of the framework. A possible way to realize this could be that the impact analysis mechanism in 4.3 returns something that identifies the current state of the dependencies in the system (a dependency state identifier). When invoking the update mechanism, this identifier needs to be supplied in with the update command. If the state of the dependencies in the system has changed since doing the impact analysis, the update should fail and the impact analysis will need to be done again before the update can be applied.

After applying the update, the application can be started up again.

Note it should be possible to define interceptors for application-level operations so that bundles can take a custom action when this happens.

4.5 Upgrading and patching an application

At any point in time the deployer may have installed multiple applications. The supplier has delivered a patch for an existing application, which needs to be applied to the runtime. Similar to 4.3, the impact of patching the application has to be evaluated.

The patch itself may or may not replace the full set of bundles included by the original application deployment. In addition, the patch itself is versioned. An inquiry of the installed applications needs to return the list of applications as well as their patches.

4.6 Purging of bundles

The deployer has patch or upgraded an application, which replaces some bundles with newer versions. The impact analysis mentioned in 4.3 and 4.5 has determined that some bundles are no longer required in the runtime as their newer versioned replacements are also used in the wiring inside the runtime. The older bundles need to be purged from the runtime configuration to conserve resources.

4.7 Installation of applications

A deployer is given an application to deploy. The application consists of a set of bundles and some additional metadata. The deployer installs the application as a whole and needs to ensure the bundles install, resolve and start as expected.

It may be the case that the application includes all non-system provided transitive dependencies and does not wish to have more recent or missing dependencies automatically provisioned via some other means, such as a bundle repository.

It should also be possible to provide a 'partial' application and have its transitive dependencies provisioned from a bundle repository.

4.8 Development Integration

Integration with a Development Environment is very important. It should be possible to easily redeploy the OSGi application being developed in the target OSGi framework, where only the minimum of bundles need to be updated, ideally leaving the rest of the application running. This could tie in with use-case 4.3In this case it is unlikely that the version number of the OSGi bundle has changed. We are operating in a 'snapshot' mode here.

4.9 Establish the Application Fingerprint

An OSGi application is different to a traditional application. A traditional application has a version number, optionally with a patch level that clearly defines the exact version of the application and all of its sub-components. In OSGi, an application has its own bundles, but typically shares a number of bundles with other applications.

An application descriptor should have its own version number, but what it is really constituted of is defined by the bundles and services that it is bound to at runtime. It should be possible for a deployer to check whether two applications are identical and if not, what the differences are. This should include bundles bound to via import-package, require-bundle and other wiring (such as dynamic-import-package) as well as the services in use by the bundles part of that application.

4.10 Tooling integration

A variety of tools is available for OSGi today, which ranges from Eclipse Developer tools and maven or ant-based command-line tools to deployment and management tools. Currently the tooling is fragmented and many of these tools mostly operate on the bundle level. Where they extend beyond it, they use their own proprietary metadata. Standardized Application Metadata could provide a rendezvous point for these tools and allow them to collaborate on the application level. Since many bundles are developed, deployed and managed in a larger context spanning a number of bundles, this higher level view and the availability of associated metadata in a standardized form for higher-level tools is very desirable.

4.11 Rollback of deployments

An administrator has installed a new version of an application. However the new version doesn't turn out to be working appropriately, so the administrator decides to roll back the installation.

Today, the Deployment Admin specification supports the concept of deployment rollback. In the future it would be good if this rollback functionality was also provided outside of Deployment Admin and in the context of this RFP.

4.12 Sharing of bundles across applications

The concepts behind and design of the OSGi Framework are all about sharing and reuse of components. It is therefore highly likely that bundles are shared across subsystems. While in some cases it might be necessary to load two instances of the same bundle in a single VM (either to resolve class space consistency issues or for some other reason such as running nested frameworks) a deployer could have a very strong preference for sharing the same bundle across all subsystems using the bundle. While the design of the Subsystem component should not prevent all these deployment scenarios, it should also not prevent complete sharing of all bundles in the system, if this is possible within the class loading constraints of the VM.

4.13 Configuring subsystems

A certain subsystem has configuration data associated with it. In this case it's the port number that consumers connection to when accessing the functionality over the network. A number of other configuration items are also present. The subsystem is configured using the OSGi Configuration Admin Service.

Default values for these configuration items should be provided as part of the subsystem definition. On the one hand this serves as a convenience to the user, as it relieves the user from the potentially tedious job of setting all the configuration items. It also provides a clear list of all of the configuration items that can be changed.

It should be possible to override the configuration values by providing new values in the OSGi Configuration Admin Service.

Additionally, the configuration *defaults* might change when the subsystem is used as part of another subsystem. So when a subsystem *contains* another subsystem as part of its definition, it should be possible to provide new defaults as part of the enclosing subsystem definition.

4.14 Application Assembly

Development of large applications is often undertaken by different teams. Each team is responsible for a 'subsystem' (or sub-application) of the application (similar to the view shown in figure 1). Each team has aspects of their sub-system that they share within the team (i.e. packages and services shared between bundles within the subsystem) but to aid application assembly only a subset of those packages and services are exposed by the subsystem's bundles are made available to the application assembler. This differentiates the subsystem externals from the internals used by the team. The externals form the contract with the assembler in the same way a bundle only exposes a subset of its packages.

In packaging the contents of the subsystem, the team also want to ensure consistent resolution and potentially restrict the visibility of packages and services being imported from a parent. This can be for a number of reasons:

- prevent accidental enablement off some optional functionality provided by a re-used internal bundle,

- ensure the wiring of the internal bundles is not unduly influenced by packages available outside the bundle.

- ensure services matching those used internally are not accidentally picked up from outside the subsystem

- ensure child subsystems of that subsystem do not unitentionally pick up pakcages and services (the subsystem developer is unlikely to have detailed knowledge of what is used within the subsystems they re-use).

5 Requirements

~~List the detailed requirements that should hold for the solution. This should be described as statements that can be tested in some way.~~

5.1 Subsystem Modeling

REQ 1. The solution MUST provide a means to describe a subsystem which can be accessed both inside a running OSGi framework (e.g. through an API) as well as outside of a running framework (e.g in a file).

REQ 2. The solution MUST define a subsystem definition format.

REQ 3. It MUST be possible to define a subsystem in terms of OSGi bundles.

REQ 4. It MUST be possible to make artifacts other than OSGi bundles part of a subsystem definition.

REQ 5. It MUST be possible to include ~~another~~a subsystem definition in another subsystem definition.

REQ 6. It MUST be possible to reference another subsystem in a subsystem definition, which makes the other subsystems a dependency.

REQ 7. ~~d - see below.Deferre subsystems.nestIt MUST be possible to~~ It MUST be ~~possible to~~simple to define a scoped ~~scope~~ subsystem~~s~~ of the type defined in requirement 8 but with a default sharing policy that hides everything in the given subsystem from the parent of the given subsystem and makes all bundles (for wiring purposes only), packages and services visible inside the given subsystem from its parent.~~. Scopes form a hierarchy. The subsystems, bundles, packages, and services belonging to a given scoped subsystem are visible only within the scope of the given subsystem and any scopes nested within that scope. Subsystems, bundles, packages, and services belonging to unscoped subsystems are visible within all (scoped or unscoped) subsystems.~~

REQ 8. It MUST be possible to scope subsystems. Scoped subsystems form a hierarchy. The subsystems, bundles, packages, and services belonging to a given scoped subsystem are visible within the scope of the given subsystem. A subsystem MUST have control over a sharing policy to selectively make packages and services visible to the parent of that subsystem. A subsystem MUST have control over a sharing policy to selectively make bundles (for wiring purposes only), packages and services visible inside that subsystem from its parent. Subsystems, bundles, packages, and services belonging to unscoped subsystems are always available to all peer subsystems (scoped or unscoped) and bundles.

REQ 9. ~~Subsystems MUST have a Symbolic Name, Version and Display Name as part of their attributes.~~

REQ 10. ~~It MUST be possible to store subsystem definitions in a bundle and deploy these definitions into the OSGi Framework as such.~~

REQ 11. ~~It MUST be possible to store subsystem definitions outside of a bundle and deploy these definitions into the OSGi Framework as such.~~Subsystems MUST be uniquely identifiable and versioned.

REQ 12. Subsystem definitions MUST be extensible~~ in a proprietary way~~. This will allow tools to store associated data along side the subsystem definitions.

REQ 13. ~~Deferred -UST~~

REQ 14. ~~The solution MAY provide a file format to contain bundles and other files that together form a subsystem (like the Deployment Admin file format), however use of this file format MUST NOT be required.~~Omitted - see below.

5.2 Dependency Management

REQ 15. ~~Deferred -The solution MUST be capable of using OBR to resolve dependencies, but use of a repository resolver~~OBR MUST NOT be mandated.this resolver ~~The solution MUST also be capable of operating without OBR.~~Omitted - see below.

REQ 16. It MUST be possible to use ~~the OBR capabilities model~~constraints to declaratively identify bundles and other artifacts in a subsystem definition.

REQ 17. ~~Deferred -IIt MUST be possible to use URLs to denote bundles and other artifacts in a subsystems definition.~~Omitted - see below.

REQ 18.

REQ 19. ~~requirements discussed until here @@@ - @@@ Dublin F2F 2009 ~~It MUST be possible to define a subsystem ~~based on~~as a number of key bundles. The transitive dependencies of the subsystem are~~, plus their transitive dependencies, where the transitive dependencies are~~ inferred from the meta-data of the key bundles.

REQ 20. ~~Deferred -It MUST be possible to disable the dynamic resolution of (transitive) dependencies and define a subsystem purely based on bundles and other subsystems listed.~~Omitted - see below.

REQ 21. ~~Deferred -It MUST be possible to use the following to declare dependencies: import-package, require-bundle, an extender, a service, OBR 'capabilities' or a URL. on packages, bundles, OSGi services and extenders.It MUST be possible to declare dependencies ~~Omitted - see below.

5.3 Administration

REQ 22. The solution MUST provide an API to query the subsystems available in the OSGi container, their associated state (as a snapshot) and dependencies.

REQ 23. The fidelity of the subsystem states MUST be sufficient to capture the possible states of its constituents

REQ 24. The solution MUST provide an API to ~~install/uninstall and start/stop a subsystem~~drive the subsystem lifecycle.

REQ 25. ~~The solution MUST provide an API to update a subsystem.~~Deferred – see below.

REQ 26. ~~ @@@ we need to investigate the level of sharing we want to address @@@The solution MUST provide a means to update a set of bundles that are used by one or more subsystems.~~.

REQ 27.

REQ 28. ~~The solution SHOULD clean up bundles that aren't needed any more when uninstalling or updating a subsystem, this MUST exclude bundles that are marked as system bundles.~~

REQ 29. ~~installation and/or update actions.roll backThe solution SHOULD provide an API to~~ The solution SHOULD provide an API to do impact analysis regarding replacing one or more bundles or subsystems. This impact analysis should list the subsystems affected and describe how these subsystems are using the affected bundle: either by importing an interface or a class from it, purely through services or through the extender pattern. It should also describe existing transitive dependencies and new transitive dependencies.

REQ 30. ~~Deferred -~~

REQ 31. ~~The solution MUST provide an API that provides a 'fingerprint' of a given subsystem. The fingerprint is a unique identifier that is based on the bundles and their dependencies (including services used) that constitute the subsystem. hanism for updates. provides an optimistic mecThe APIs that modify the system (e.g. install/uninstall/update) MUST accept a fingerprint and fail the operation if the state of the system does not match the state when the fingerprint was obtained.~~Omitted - see below.

REQ 32. The APIs that modify the system SHOULD not leave the system in an inconsistent state. ~~do so atomically/transactionally~~.

REQ 33. ~~DeferredThe solution SHOULD provide RFC 147 commands for its administrative APIs.~~Omitted - see below.

5.4 Runtime

REQ 34.

REQ 35. ~~The solution MAY utilize nested framework, but nested frameworks MUST NOT be mandated.~~The solution MUST allow ~~sharing of bundles across subsystems~~ a single bundle to be part of multiple subsystems.

REQ 36. The solution MUST allow multiple versions of the same bundle ~~being installed~~.

REQ 37. Subsystems MUST have a well-defined life-cycle.

5.5 Development Support

REQ 38. The solution MUST support the development process by allowing bundles to be updated with other bundles that have the same version as the previous, similar to how Maven supports SNAPSHOT versions of artifacts.

5.6 RFC 138 Compatibility ~~REQ 29. The solution MUST be compatible with the scoping mechanism defined by RFC 138.~~

REQ 39. The solution MUST be compatible with the scoping mechanism defined by RFC 138.

REQ 40. ~~Deferred – see below.The solution SHOULD NOT require each scoped or unscoped subsystem to reside in its own RFC 138 composite bundle.~~

REQ 41. ~~Deferred – see below.The solution MUST be compatible with the scoping mechanisms defined by RFC 138.~~REQ 30. The solution SHOULD NOT require each scoped or unscoped subsystem to reside in its own RFC 138 composite.

5.7 ~~Deferred~~Omitted Requirements

The above requirements constitute a consistent initial subset based on existing implementations of and user feedback on constructs similar to subsystems. The following requirements were omitted from this subset, but may be ~~addressed~~included in ~~the future~~future RFPs.

5.7.1

5.7.2 Subsystem Modelling ~~It MUST be possible to scope subsystems.REQ 7.~~

REQ 1~~0~~1. The solution MUST provide a file format to contain bundles and other files that together form a subsystem (like the Deployment Admin file format), however use of this file format MUST NOT be required.

5.7.3 Dependency Management

REQ 1~~1~~2. The solution MUST be capable of using a repository resolver to resolve dependencies, but use of this resolver MUST NOT be mandated. (Note: this requirement appears to be an implementation detail.)

REQ 1~~3~~4. It MUST be possible to use URIs to denote bundles and other artifacts in a subsystems definition.

REQ 1~~5~~6. It MUST be possible to disable the dynamic resolution of (transitive) dependencies and define a subsystem purely based on bundles and other subsystems listed.

REQ 1~~6~~7. It MUST be possible to declare dependencies on packages, bundles, OSGi services and extenders.

5.7.4 Administration

REQ 2~~0~~1. The solution MUST provide an API to update a subsytem.

REQ 2~~2~~2. The APIs that modify the system (e.g. install/uninstall/update) MUST provides an optimistic mechanism for updates.

REQ 2~~4~~5. The solution SHOULD provide RFC 147 commands for its administrative APIs.~~Installing an App – touch points with OBR?~~

~~Should apps be nestable?~~

~~Maybe error information / logs can be grouped per application so that the deployer can retrieve them on an application level in case of issues.~~

~~There needs to be a possibility of system bundles, which are never automatically uninstalled.~~

~~Applications need to be versioned.~~

~~Applications need to have a well-defined lifecycle~~

~~Maybe we should not be using the word Application. “System” or maybe “Assembly” might be better.~~

REQ 42. ~~The application metadata should be declarative such that it can be used and evaluated outside of the framework, i.e. during the build process and or deployment/packaging phase~~

REQ 43. ~~Backup – thoughts~~

~~David B: no real list of requirements here yet. This is just a parking space for wild thoughts at this stage.~~

~~This 'Application Metadata Service' should provide the following~~

~~- what applications are currently deployed~~

- ~~what patch levels~~

~~- which apps are successfully installed, which ones failed (missing dependencies, bundle failed to start etc)~~

~~- what bundles is application X using in what version~~

~~- I want to add / remove an application~~

~~- Given a bundle, which of the installed applications is using it?~~

~~- This spec should have the full capability to share bundles across applications but not require that applications which depend on the same bundle share the same instance~~

~~It needs to be able to describe an application:~~

– ~~key bundles~~

– ~~key services~~

– ~~display name~~

– ~~the list of apps should be much smaller than the list of bundles and should be meaningful to the end user~~

– ~~possibly the aggregation of applications into products~~

~~Metadata in a bundle~~

~~Should have an API~~

- ~~query registered apps~~
- ~~query installation status~~
- ~~send events for changes that can be listened to~~
- ~~given an app, what bundles is it using?~~

REQ 44. ~~Given a bundle, what apps are using it?~~

REQ 45. ~~REQ 29. The solution MUST be compatible with the scoping mechanism defined by RFC 138.~~

REQ 46. ~~REQ 30. The solution SHOULD NOT require each scope or unscoped subsystem to reside in its own RFC 138 composite .~~

~~**RFC 138 Compatibility**~~

REQ 47.

6 Document Support

6.1 References

[1]. Bradner, S., Key words for use in RFCs to Indicate Requirement Levels, RFC2119, March 1997.

[2]. Software Requirements & Specifications. Michael Jackson. ISBN 0-201-87712-0

~~*Add references simply by adding new items. You can then cross-refer to them by chosing <Insert><Cross-Reference><Numbered Item> and then selecting the paragraph.* ***STATIC REFERENCES (I.E. BODGED) ARE NOT ACCEPTABLE, SOMEONE WILL HAVE TO UPDATE THEM LATER, SO DO IT PROPERLY NOW.***~~

6.2 Author’s Address

Name	David Bosschaert
Company	Progress Software
Address	158 Shelbourne Road, Ballsbridge, Dublin 4, Ireland
Voice	+353 1 637 2000
e-mail	dbosscha@progress.com

Name	Guillaume Nodet
Company	Progress Software
Address	158 Shelbourne Road, Ballsbridge, Dublin 4, Ireland
Voice	+33 6 60 48 00 39
e-mail	gnodet@progress.com

Name	
Company	
Address	
Voice	
e-mail	

6.3 End of Document