RFC 88 — MEG Deployment

Confidential, Draft

47 Pages

Abstract

The OSGi Mobile Expert Group (MEG) aims to apply and to extend the OSGi specifications for the operational management of mobile device platforms. This document is part of that overall effort and addresses technical solution for a software component deployment model.

0 Document Information

0.1 Table of Contents

All Page Within This Box

All Page Within This Box

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 [1].

Deployment Package – A JAR file containing a collection of resources, including but not limited to bundle JAR files. A deployment package is used to deliver applications and other software components to a device. A deployment package may be signed and is not known by the OSGi Framework.

Fix Pack – A special type of Deployment Package used for carrying out optimized update operations on already installed deployment packages. For size efficiency, a fix pack may only contain the changes required to update already installed resources and bundles, omitting any material not changed between versions.

0.3 Revision History

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

Revision	Date	Comments
Initial	*Oct 26 2004*	*Initial Draft after Sophia MEG meeting, based on the discussions at the meeting and earlier work in MEG.* *Olivier Gruber, IBM, ogruber@us.ibm.com*
0.1	*Nov 05 2004*	*Proposed modifications added for review.* *Gabor Paller, Nokia, gabor.paller@nokia.com*
0.2	*Nov 09 2004*	*Decisions of the 2004 Nov. 8 phone conference incorporated* *Gabor Paller, Nokia, gabor.paller@nokia.com*
0.3	*Nov 11 2004*	*Writing up agreement from the phone call of the 11-11-2004* *Merged ResourceProcessor and DataCustomizer. keeping processor concept only, usable for both.* *Supporting in-RP delivery of processors* *Matching processor and resources by filters, one filter for the OSGi registry and one filter on resources.* *Olivier Gruber, IBM, ogruber@us.ibm.com*

Revision	Date	Comments
0.4	*Dec 01 2004*	*Editing changes.* *Addition of security-related text for Resource Processors from Joe Rusnak, JavaDoc interfaces, and DMT Deployment Management objects.* *Mark Hansen, Motorola, mark.hansen@motorola.com*
0.41	*Dec 06 2004*	*Decisions of Dec 3rd phone conference incorporated. Added revised text for Resource Processor security from Joe Rusnak.* *Mark Hansen, Motorola, mark.hansen@motorola.com*
0.50	*Dec 21 2004*	*Reformatted APIs back to JavaDoc format.* *Copied relevant requirements from RFP-53.* *Added revised text for Resource Processor security from Joe Rusnak.* *Added information on service filtering from Gabor Paller.* *Editing of sections 5.2 and 5.3.* *Mark Hansen, Motorola, mark.hansen@motorola.com*
0.60	*Jan 23 2005*	*Removal of Resource Processor registration/unregistration/listing APIs from JavaDoc.* *Revisions from Nokia review and Jan 17 conference call.* *Addition of OMA DM scripting example from Motorola to Section 5.2.* *Mark Hansen, Motorola, mark.hansen@motorola.com*
0.65	*Feb 03 2005*	*Continuation of revisions from Jan 17 conference call.* *Addition of Deployment Permissions to Javadoc from Gabor Paller.* *Mark Hansen, Motorola, mark.hansen@motorola.com*
0.70	*April 1, 2005*	*Extensive changes based on the review of Peter's draft specification.* *Joe Rusnak, IBM, jgrusnak@us.ibm.com*
0.71	*April 8, 2005*	*Minor changes to fix mistakes caught by Nokia* *Joe Rusnak, IBM, jgrusnak@us.ibm.com*

All Page Within This Box

1 Introduction

This document is part of the overall effort of the *Mobile Expert Group* (MEG) under OSGi to specify technical solutions for operational management of mobile device platforms based upon CDC configuration of J2ME and OSGi. It defines a unit of deployment and management, called a *Deployment Package*, that is used for installing, updating, and removing bundles and other artifacts on a mobile device platform. It is a first step toward smart infrastructure to manage such mobile devices.

2 Application Domain

Mobile devices are rapidly becoming more complex as their capabilities increase. One such capability is the ability to install new software components after the time of manufacture. For example, device users already are, or soon will be, accustomed to installing applications or services on their devices from remote servers, either wirelessly or via local connectivity. Operators and other providers also wish to perform software repair and upgrades remotely to subscriber devices. To enable these deployment capabilities, a powerful management framework is required. The OSGi Service Platform, i.e. the OSGi Framework, provides the foundation to such a powerful management. This document will address the use-cases and requirements for MEG and define the management agent framework necessary for software component deployment for mobile devices.

3 Problem Description

In R3, the OSGi Service Platform didn't specify the management agent itself, only the core mechanisms and concepts that an agent would use. The platform manages the lifecycle of bundles, configuration objects, and permission objects but the overall consistency of the runtime configuration is the responsibility of the agent. In other words, the management agent decides—as it sees fit—to install, update, or uninstall bundles as well as create or delete configuration or permission objects. The goal here is to specify a default management agent and a provisioning infrastructure for mobile devices.

In the future, a possible solution to this problem may be to have a "smart" infrastructure for robust and automated provisioning. This "smart" infrastructure would require a complete understanding of OSGi concepts and enough knowledge of the current device states for making correct decisions. When a new functionality is required, or conversely not needed anymore, the infrastructure could generate a plan for installing or removing such

functionality. This plan could then be downloaded as a "*Smart Package"*—a multi-step deployed process effecting potentially complex changes on the overall runtime configuration of a device, created with full knowledge of the state of the device.

While powerful and flexible, this approach suffers from requiring the presence of a "smart" infrastructure and its understanding of the relatively fine-grain concepts of the current OSGi platform. To avoid these two issues in the OSGi R4 timeframe, a higher-level concept capturing packaged resources is specified. A *Deployment Package*, or DP, groups resources as a unit of management. A deployment package is something that can be installed, updated, and uninstalled as a unit—following a clean, simple, and robust process. This construct allows manual management and can serve as a stepping stone for future optimization.

The essential difference between smart packages and deployment packages is that a smart package represents a complete plan to effect changes—a plan that is tailored for a particular device at a particular time for effecting a given change. A deployment package is only a set of related resources that need to be managed as a unit rather than individual pieces. It is not a plan from one consistent state to another; several deployment packages may be needed to achieve a new consistent state. Like bundles, a deployment package might not be a totally self-contained unit—it might have dependencies on Java packages and services provided by bundles belonging to other deployment packages. Nevertheless, a deployment package primarily groups one or more bundles and other related resources, such as an RFC 94 AUTOCONF.XML file for example, into one entity that is easier to manage.

As an example, consider a suite of games sharing some common parts. Let's say we have two games, Chess and Backgammon. Both share a top-score database as well as a 3D graphics library. The 3D graphics library comes from a third-party provider and would therefore be a deployment package of its own, composed of several bundles and possible configurations. The top-score database would also be its own deployment package, and would in fact be optional. It offers a service for storing top scores, but games may function without this service. Finally, each game could be a deployment package allowing them to be installed independently. Alternatively, the two games could be packaged into the same deployment package, but in this case they must be installed and removed together.

Deployment packages are represented as first-class objects on devices. This document does not specify by whom and when deployment packages are created, named, versioned, and signed. This document does specify the rules governing the format, correctness, and processing of deployment packages.

4 Requirements

This section lists the requirements used as guidelines for the design described in this document. Most of the requirements concerning Deployment originated from RFP-53 [4]. Unfortunately, in many cases the terminology used in this document deviates substantially from that used in [4], although the intent has largely remained the same. The terminology section of this document and RFP-53 should be examined before any comparison of requirements to Deployment design. For example, in the original RFP-53 terminology, a "Unit of Deployment" (UDP) was considered to be equivalent to a bundle, and "Unit of Delivery" (UDL) was considered to be a composite construct consisting of one or more bundles, termed a "bundle suite" for some period of time. In this RFC, only one deployment construct has been identified, a deployment package, which resembles RFP-53's UDL package.

4.1 Basic Operations

REQ-DEP-01-01. It MUST be possible to install an UDP and an UDL on the device.

REQ-DEP-01-02. It MUST be possible to uninstall an UDP and an UDL from the device.

REQ-DEP-01-03. It MUST be possible to upgrade an existing UDP to a higher version number.

REQ-DEP-01-04 It MAY be possible to upgrade an existing UDL to a higher version number.

REQ-DEP-01-05. It SHOULD be possible to downgrade an existing UDP to a lower version number.

REQ-DEP-01-06 It MAY be possible to downgrade an existing UDL to a lower version number.

REQ-DEP-01-07. It MUST be possible to reinstall an existing UDP.

REQ-DEP-01-08 It MAY be possible to reinstall an existing UDL.

REQ-DEP-01-09. It MAY be possible to control the point of time when the downloaded component becomes accessible to other components.

REQ-DEP-01-10. It MUST be possible to control access to the deployment functionality by a flexible policy system. *For example it is possible to restrict, who can install what software over local API calls or what management server can update what software remotely.*

4.2 Off-line Operations

REQ-DEP-02-01. It MUST be possible to install an UDP or an UDL from a PC without a management server being involved, such as from a CD-ROM drive.

REQ-DEP-02-02. It MUST be possible to install an UDP or an UDL locally, without the help of the management server. *For example it is possible to download a package from the Internet and install it without management server interaction*

REQ-DEP-02-03. It SHOULD be possible to identify all the parts of an installed software component (code, resource, database, etc.) using dependency information stored in the management framework. *For example, a selective backup utility may use this information to save only the relevant parts during the backup process.*

REQ-DEP-02-04. It MUST be possible to access management functionalities over a local API. *For example it is possible to write a manager that allows installation of software components, provided that the invoking user-application has installation rights.*

4.3 Server-Assisted Installation

REQ-DEP-03-01. It MUST be possible to install/update UDPs and UDLs over the network from a server.

All Page Within This Box

REQ-DEP-03-02. It MAY be possible to install/update UDPs and UDLs on the user's request. The user selects the component to download, and the installation happens automatically

REQ-DEP-03-03. It MUST be possible for a server to initiate the installation/update of an UDP or UDL independently of the user's actions. *For example a server administrator can install new software on the device.*

REQ-DEP-03-04 If a server initiates the installation/update of an UDP or UDL independently of the user, device MAY require user confirmation before completing the installation.

REQ-DEP-03-05. It MUST be possible that a server can assemble the contents of an UDL or UDP package with parts (i.e. binaries, settings, resources) custom-selected for a specific device. For example, the relevant application parts may be selected based on the device capability, pre-installed software, etc.

REQ-DEP-03-06. Management model MUST be supported where the secure communication between the remote manager and the mobile device can be established.

REQ-DEP-03-07. During an UDP installation or update operation, the management system SHOULD support dependency resolution to initiate the acquiring and installation of a missing component.

REQ-DEP-03-08. It MAY be possible for software components to start management sessions if they detect a malfunction that may be corrected by management interaction. *For example, if an e-mail user-application detects that the e-mail server cannot be contacted, it may start management session with the management server to "repair" the e-mail server settings.*

REQ-DEP-03-09. It MUST be possible for the mobile device to be managed by more than one remote manager (or local manager, e.g. the user). Such managers should possibly have different rights, e.g. manage different applications or perform different operations.

REQ-DEP-03-10. SyncML DM as management protocol MUST be supported.

REQ-DEP-03-11. Java MIDlet OTA MUST be supported as download protocol for MIDlets.

REQ-DEP-03-12. OSGi-based provisioning MUST be supported for applications (non-MIDlets).

REQ-DEP-03-13. The management protocol MUST be supported over wide-area radio networks (like cellular networks).

REQ-DEP-03-14. The management protocol MAY support short-range protocols like WLAN, Bluetooth, etc.

REQ-DEP-03-15. The management protocol MAY support local cable connections.

4.4 Deployment/Undeployment/Upgrade Customization

REQ-DEP-04-01. It MUST be possible to customize the installation process by scripts, add-ons, etc

REQ-DEP-04-02. It MUST be possible to assign custom installer, uninstaller, upgrader and downgrader logic to installation packages that can add component-specific intelligence to the installation process. *For example, the custom installer can modify settings so that the user doesn't have to bother with configuration. The custom installer/uninstaller/upgrader/downgrader logic can be implemented in any convenient language like Java.*

All Page Within This Box

REQ-DEP-04-03. A scripting language MAY be available that provides conditional installation and user interaction (prompting, selection) features.

REQ-DEP-04-04. It MUST be possible to restrict the access rights of the install script. The install script doesn't run on behalf of the installer and may have limited rights.

REQ-DEP-04-05. It SHOULD be possible to associate component-specific data to installed components. This data is loaded into the appropriate data store during installation process. *For example, it is possible to declare that 3 database tables need to be created during installation and they are initialized from initial data that the package contains.*

4.5 Packaging

REQ-DEP-05-01. It MUST be possible to package all data (code, settings, resources, other component-specific data) into one file that can be expanded into individual files on the device.

REQ-DEP-05-02. The UDL MAY contain multiple software components. *For example, it is possible to package multiple units of execution within a single UDL package, such as a GUI-oriented user application and a background service associated with it.*

REQ-DEP-05-03. It MUST be possible to digitally sign UDLs and UDPs to check their authenticity. *For example it is possible to add company X's signature to the package so that the user or the management system can be sure of the package's origin.*

REQ-DEP-05-04. It MUST be possible to support of digital content signing compatible with MIDP2.0. Specifically, it should be possible to assign roles to signers (e.g. operators, manufacturers).

4.6 Software Component Discovery and Subscription

REQ-DEP-06-01. It MUST be possible to list available software components from the specified component repository or management server by a device.

REQ-DEP-06-02. It MAY be possible to subscribe for a specified software component if the component repository supports a subscription model.

REQ-DEP-06-03. It MUST be possible to unsubscribe for a subscribed software component if the component repository supports a subscription model and the specified software component is provided by the selected component repository.

4.7 Operator and Device Characteristics

REQ-DEP-07-01. It MUST be possible to attach an operator-specific subscriber identifier (e.g. IMSI) to an UDL/UDP.

REQ-DEP-07-02. It SHOULD be possible to attach an operator identification code to an UDP/UDL.

REQ-DEP-07-03. It MAY be possible to attach a device identification code (e.g. IMEI) to an UDP/UDL.

REQ-DEP-07-04. The operator or device characteristic MUST be attached to the UDL/UDP in such as way that it cannot be removed or thwarted by a reasonable effort from a user.

REQ-DEP-07-05. If the operator or device characteristic of a UDP/UDL does not match the current aspects of the device (e.g. the user changed subscription plan), the management agent MUST guarantee that the UDP/UDL is removed after a reasonable time.

---

5 Technical Solution

A Deployment Package (DP) is a reified concept, like a bundle, in an OSGi Service Platform. It is not known by the OSGi Framework, but it is created and managed by the Deployment Admin service. A DP groups resources into one management unit -- the associated resources are installed, updated, or uninstalled as one.

A deployment package is a stream of resources (including bundles) which, once processed, will result in new artifacts being added to the OSGi platform. These new artifacts can include installed Bundles, new Configuration objects added to the Configuration Admin service, new Wire objects added to the Wire Admin service, or changed system properties. All the changes caused by the processing of a deployment package are persistently associated with the deployment package, so that they can be appropriately cleaned up when the deployment package is uninstalled.. In order to simplify cleanup, there is a strict “no overlap” rule imposed on deployment packages. Two DPs are not allowed to create or manipulate the same artifact. Obviously, this means that a bundle cannot be in two different DPs. But it also means, for example, that an AUTOCONF.XML configuration file contained by one DP cannot manipulate a Configuration object created by another DP. Any violation of this “no overlap” rule is considered an error and the install or update of the offending deployment package must be aborted.

The strong no-overlap rule ensures a clean and robust lifecycle. It promotes the simple cleanup rule: *the deployment package that creates an artifact is the one that destroys it*. Generally speaking, this certainly restricts what deployment packages can be, but in practice, we do not believe it to be a problem, especially with the smaller applications envisioned for MEG on OSGi R4. In fact, it is more a mindset than a restriction. A deployment package groups those resources designed to be managed together. Sharing is achieved through the normal OSGi mechanisms: Java package imports and OSGi services (declarative or not).

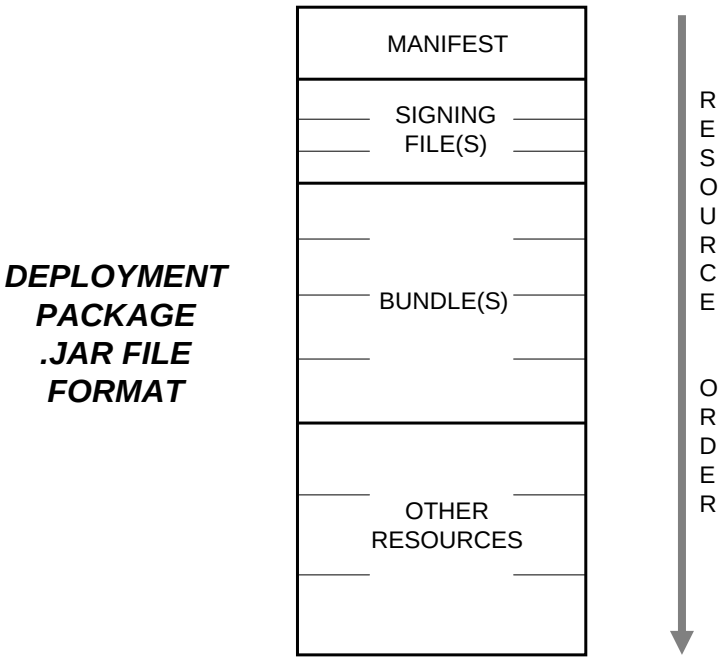
5.1 Deployment Package Format

5.1.1 Naming

A Deployment Package has a name and version associated with it. Package authors should try for uniqueness in naming through the usage of reverse domain naming introduced for Java packages. The version follows the scheme proposed in RFC 79 for implementation versions. Together, a name/version pair specifies a unique Deployment Package; a device will consider any Deployment Package with identical name/version pairs to be the same.

5.1.2 Structure and Manifest

A Deployment Package is a standard JAR file as specified in the J2SE 1.3 documentation [3], which has an optional file extension of “.dp”. It has a standard Java manifest file (/META-INF/MANIFEST.MF), which must be the first file in the JAR. A Deployment Package must be formed in such a way that it can be read with a Java JarInputStream, while supporting getManifest() and the verification of the entries. The order of resources in the JAR file is important – it allows the JAR file to be processed as a stream, avoiding the need to store the complete deployment package on the device. A deployment package JAR file has the following format:



The manifest of a Deployment Package consists of a global section and separate sections for each resource contained within it. The global section must contain the following fields:

`DeploymentPackage-Name`: The name of the deployment package as a reverse domain name, as specified by the package section of the Java Language Specification [2].

`DeploymentPackage-Version`: The version of the deployment package

The following headers may also appear in the global section:

`DeploymentPackage-Copyright:` Copyright specification for this deployment package.

`DeploymentPackage-ContactAddress:` How to contact the vendor/developer of this deployment package.

`DeploymentPackage-Description:` A short description of this deployment package.

`DeploymentPackage-DocURL`: A pointer ot any documentation that is available for this deployment package.

`DeploymentPackage-Vendor:` The vendor of the deployment package.

As with any Java manifest, other headers may be added but may be ignored by the Deployment Admin service. If any fields have human readable contents, natural language translation may be provided through property files as described in RFC-74 (Manifest Localization).

In addition to the global section, a manifest has a section, called a *Name* section, for each resource contained within it as defined by the Java 2 manifest format. The Name section starts with a Name header that contains the path name of the resource in the manifest. Additional optional headers may also be present.

All Page Within This Box

An example manifest of a deployment package containing two bundles, “foo” and “bar”.

```
DeploymentPackage-Name: com.mycompany.mydeploymentpackage
DeploymentPackage-Version: 2.3.1

Name: bundles/foo.jar
Bundle-SymbolicName: com.mycompany.bundles.foo
Bundle-Version: 2.3.1

Name: bundles/bar.jar
Bundle-SymbolicName: com.mycompany.bundles.bar
Bundle-Version: 1.5.3
```

The “no overlap” model for deployment packages means that a bundle must belong to one and only one deployment package. Although bundles are identified by a name/version pair, deployment packages “own” bundles by name alone. Two deployment packages cannot contain different versions of a bundle with the same name. Bundles that are pre-installed on the device (i.e., that were not dynamically deployed as part of a DP) are automatically members of a default “System” deployment package, which included in the results returned by `DeploymentAdmin.listDeploymentPackages()`.

Deployment packages may have other resources as well. Resources are available only during the initial install and subsequent update operations on their deployment package. They are also described in the “Name” sections of the manifest.

5.1.3 Types

A Deployment Package may take one of two stream formats: a “full” format or a “fix-pack” format, which can be distinguished through the presence of a manifest header.

A fix-pack is an optimized Deployment Package format which may not contain all of the resources associated with it. It assumes that a particular deployment package is already installed and contains updated versions of resources pertinent to only that package. A fix-pack will specify the range of versions for a particular deployment package for which it is relevant. If the already installed deployment package is outside of a specified fix-pack’s range of versions, then the installation of the fix-pack should not occur.

A fix-pack can be distinguished from a full format through the presence of a manifest header, `DeploymentPackage-FixPack`, contained within the global section of the manifest.

```
DeploymentPackage-FixPack: <starting major version>.<starting minor
version> , <ending major version>.<ending minor version>
```

An example:

```
DeploymentPackage-FixPack: [1.3,3.4]
```

In this example, the manifest header denotes a fix-pack which is only applicable to versions 1.3 through 3.4, inclusive, of a given deployment package. The syntax for the range should be borrowed from the ranged versions for Java package import statements.

A fix-pack may not contain all of the bundles associated with the deployment package it is updating, and may only include updated versions of bundles or new bundles requiring installation. However, the manifest must still list all

All Page Within This Box

resources, bundles or otherwise, in order to distinguish between a resource that should be dropped from a deployment package because it is no longer required and a resource not requiring an update. The `DeploymentPackage-Missing` manifest header is used to indicate that some resources are not present in the deployment package. This header is mandatory for a fix-pack deployment package, i.e, it must be present if the `DeploymentPackage-Fixpack` header is present. The header must not be used if the `DeploymentPackage-Fixpack` header is not present.

```
DeploymentPackage-Missing: <true / false>
```

If a bundle or other resource is listed in the manifest, but is not present in the fix-pack DP, then the identified bundle or resource contained within the currently installed deployment package is assumed to be up-to-date.

5.1.4 Signing

Deployment packages are signed by JAR signing, compatible with the operation of the standard java.util.jar.JarInputStream class. This compatibility requires that the manifest must be the first file in the input stream, and the signature file must be the second. All of the files in the deployment package are signed, and the signers of all the files must be the same as the signer of the manifest file. Multiple signers are possible.

If an implementation supports signature checking and encounters a deployment package not conforming to these signing rules, then Deployment Admin must reject it and abort the installation operation.

5.2 Resource Processor

5.2.1 Overview

There are three operations that can be performed on a deployment package: installation, update, and removal. The installation or update of a deployment package can be customized using an extensible, plugin-style architecture called *Resource Processors*. Resource processors are OSGi services which can act upon any resource files that might follow the bundle(s) in a deployment package. Resource processors are also responsible for undoing any changes they make to the platform when a deployment package is uninstalled. Resource processors implement the `ResourceProcessor` interface to provide methods that Deployment Admin invokes at various points in a deployment package's life-cycle.

Only one deployment package will be processed at any time so resource processors do not need to be re-entrant or multi-threaded.

5.2.2 Associating Resources with Resource Processors

A deployment package must associate (non-bundle) resources with the resource processors it wishes to operate upon them. This association is done through a special header within the deployment package manifest file. The header, `Resource-Processor`, is in the resource's "name" section of the manifest file. The format of the header is:

```
Resource-Processor: <some PID>
```

For example:

```
Name: META-INF/autoconf.xml
Resource-Processor: org.osgi.deployment.config_resource_processor
```

When Deployment Admin encounters this resource in the deployment package stream, it will invoke the resource processor service that registered the specified PID in the service registry to process the resource. If there is no

All Page Within This Box

service registered with the specified PID, the deployment operation must be aborted, since continuing would have unpredictable consequences.

5.2.3 Pre-Installed vs. Deployed

Resource processors may be pre-installed on a device by its manufacturer or could be delivered as a part of a deployment package. Pre-installed processors are generic, are in the service registry, and are used by any deployment package that specifies their PID. An example of a pre-installed resource processor might be a configuration resource processor that processes RFC 94 AUTOCONF.XML files [5]. On the other hand, some resource processors might be application-specific, delivered within a deployment package to customize something unique to that deployment package. An example of such an application-specific resource processor is a service to reformat or expand an application's database when the application is updated. These application-specific resource processors are called *"customizers"* – they are delivered in a bundle and are identified by a header, `DeploymentPackage-Customizer`, in the deployment package manifest file. Multiple customizers, and therefore multiple headers, are allowed. The header uses the following format:

```
DeploymentPackage-Customizer: <bundle symbolic name>
```

For example, a resource processor contained within the bundle com.mycompany.bundles.foo could be identified by the following header:

```
DeploymentPackage-Customizer: com.mycompany.bundles.foo
```

A resource processor specified in this manner must be one of the bundles in the deployment package. Deployment Admin will install and start the customizer RP at the appropriate time in the deployment operation (see Section xxxx for more details). When it is started, the customizer RP is expected to register a services implementing the `ResourceProcessor` interface; however, Deployment Admin does not actually manage the lifecycle of these non-declarative resource processors in order not to disrupt any that do long-running background processing. These services must also be registered with a service PID that can be used to identify them in Resource-Processor manifest headers.

5.2.4 Resource Processing

Processing a resource is involves a series of steps. Regardless of the operation (install, update, or remove), the first thing that happens is the `begin()` method is called on the resource processor that has been matched with the resource. When Deployment Admin calls `begin()`, it passes a `DeploymentSession` object that encapsulates the state of the deployment operation. The `DeploymentSession` object provides access to the "new" DP being deployed, and, in the case of a update, to the "old" DP being updated.

Next, for installation or update, the `process()` method is called on the same resource processor. During this step, the resource processor should perform its intended customization operation using the resource file. The `DeploymentPackage` object representing the DP being deployed provides methods that the resource processor can use to access the resource in the DP stream and the manifest headers in the DP manifest. If the resource processor encounters any problem processing the resource, it should throw the appropriate `DeploymentException`.

It is the resource processor's responsibility to persistently remember all the changes it makes so that it is able to undo the changes later. This can occur when

- the DP is uninstalled
- the DP is updated, and the new version does not include a resource that was in previous versions

All Page Within This Box

- the current deployment operation is aborted for some reason.

This history can be thought of as a `Dictionary` whose search key is the concatenation of DP name and resource name, and whose value is a `Vectors` of artifacts or changes that need to be undone.

Recall from Section 5.1.3 that an "update" DP might not include a resource that an earlier version of the DP needed. In this case, any artifacts created by the earlier resource must be cleaned up by the resource processor that handled the (now unnecessary) resource. To this end, the `DeploymentPackage` object must persistently remember all the resource processors that were invoked during its installation and subsequent updates. This list is not static and may change during the lifecycle of the deployment package. For example, some processors invoked during the installation of a deployment package may be no longer required after an update operation, requiring those processors to be dropped from the list. A new update could also add new processors to the list. If a resource is removed from the DP, then the `dropped()` method is invoked on the resource processor that processed the resource. The name of the resource is passed as a parameter. The resource processor should appropriately undo any changes it made when it processed the resource.

Next, the `prepare()` method is called on the resource processor, giving the resource processor a chance to do any last-minute processing, and to throw a `DeploymentException` if it is not able to commit the changes it made. If no resource process throws a `DeploymentException`, the `commit()` method is called on each resource processor, who should commit their changes to the environment. Upon a `commit()`, all side effects are confirmed, and necessary metadata should be stored to be able to carry out an uninstallation operation at a later time. If, on the other hand, some RP throws a `DeploymentException`, Deployment Admin will call `rollback()` on all the resource processors associated with the DP. They must clean up any changes they made.

Uninstall processing also beings with a call to the `begin()` method of each resource processor associated with the DP being uninstalled. Instead of `process()`, however, the resource processor's `dropAllResources()` method is called, instructing the resource process to prepare to undo every change made for any resource associated with that DP. If a resource processor encounters a problem undoing something, it should throw a `DeploymentException`. If no resource process throws an exception, then `commit()`is called on each resource processor associated with the DP, and the cleanup is committed. If, on the other hand, a resource processor throws a `DeploymentException`, Deployment Admin calls the `rollback()` method of each RP associated with the DP. The changes that were about to be undone are reinstated, and the uninstall operation fails.

If an orderly uninstall of a DP fails, Deployment Admin offers a "forced uninstall" capability. If operating in "forced uninstall" mode, Deployment Admin will call commit() on all the RPs associated with the DP, even if one of them threw a `DeploymentException` earlier. This might result in some "orphaned" artifacts (changes) that are not cleaned up (e.g., changes that the resource processor that threw the exception could not clean up). It is up to the implementor whether these "orphans" are cleaned up, and how.

5.2.5 Example: Configuration Processor

One application of how resource processors could be used is the management of configurations in the ConfigAdmin service. In a deployment package, a specific resource file could be created called the "autoconf" resource. This resource would contain a declarative markup language for managing configurations and would be identified as follows in the manifest of the deployment package containing it:

```
Name: META-INF/autoconf.xml
Resource-Processor: org.osgi.deployment.config_resource_processor
```

Details of the markup language can be found in RFC 94 ([5]). This language allows the creating, updating, and deleting of configuration objects.

All Page Within This Box

This example helps illustrate the no-sharing principle and how it is enforced by processors. The overall no-sharing rule translates in several finer-grain rules to enforce an ownership model of resources. A deployment package owns a set of bundles by name (e.g. symbolic name). The autoconf processing may manipulate configuration objects for owned bundles, but also for referred bundles. Referred bundles are referred to by name-version pairs, thereby uniquely identifying the referred bundles. Given this distinction between *owned* and *referred* bundles, the autoconf processor follows these rules:

- For bundles that a deployment package owns:
  - A deployment package may configure services provided by bundles that it owns:
    - managed services
    - managed service factories
    - declarative service wiring (service references)
  - A deployment package may customize freely bundles that it owns
    - Contents
    - Private data area
- For bundles that a deployment package refers to:
  - It may **not** customize bundles it refers to
  - It may **only** configure services provided by bundles it refers to through managed service factories
  - It may **only** create its own configuration objects for such factories, not modifying existing ones

Note: This model allows using managed services for owned bundles and avoids forcing the mandatory use of managed service factories if configuring is necessary. This is important in preserving the OSGi programming model.

5.2.6 Example: OMA Device Management Scripting

Another example of how resource processors may be used is in device management customization. When a new deployment package is installed, it is often necessary to configure some nodes in the device management tree (DMT). When it is uninstalled, the nodes have to be removed from the tree. A typical example is one of network-enabled application whose Internet connection profile has to be provisioned. Internet profiles are usually defined in the DMT independently from an application model, so Config Admin can not be used to define such parameters. Another example is defining a menu item through which the installed application will be available, provided main menu settings are available through the DMT (and thus OTA-provisionable)

An resource processor implementation could use standard OMA DM scripts, with some relaxation of the rules that are designed specifically to support an OTA usage pattern. In this respect, the relaxation continues the simplification methods used in the now defunct plain profile bootstrapping for OMA DM. The difference is that under the approach proposed for scripting most optional elements are simply disregarded when present.

Let's take the bootstrap syntax example from the OMA DM spec, and see how it applies to scripting:

```
<SyncML>
   <SyncHdr>
      <VerDTD>1.1</VerDTD>
```

All Page Within This Box

```
      <VerProto>DM/1.1</VerProto>
      <SessionID>0</SessionID>
      <MsgID>0</MsgID>
      <Target>
         <LocURI>My_SyncML_DM_Device</LocURI>
      </Target>
      <Source>
         <LocURI>http://www.TheUltimateManagementServer.com</LocURI>
      </Source>
   </SyncHdr>
   <SyncBody>
      <Add>
         <CmdID>1</CmdID>
         <Item>
            <Target>
               <LocURI>
                  ./SyncML/DMAcc/UltimateManagement
               </LocURI>
            </Target>
            <Meta>
               <Format xmlns="syncml:metinf">node</Format>
            </Meta>
            <Data/>
         </Item>
         <Item>
            <Target>
               <LocURI>./SyncML/DMAcc/UltimateManagement/Addr</LocURI>
            </Target>
            <Data>www.TheUltimateManagementServer.com</Data>
         </Item>
         <Item>
            <Target>
               <LocURI>./SyncML/Con/My_ISP</LocURI>
            </Target>
         </Item>
      </Add>
      <Final/>
   </SyncBody>
</SyncML>
```

The elements in the script are treated as follows:

SyncHdr – optional and disregarded
CmdId – optional and disregarded
Meta – optional, if not present the missing metadata is derived from other sources available to the implementation (such as DDF)
Data – optional
Final – optional and disregarded

The mandatory elements are SyncML, SyncBody, commands (Add, Remove, Replace, Exec), Item, Target and LocURI.

5.2.7 Security

5.2.7.1 Limiting the permissions for resource processors

All Page Within This Box

Since the MEG Deployment Admin runs with `AdminPermission`, any resource processors it calls will run with administrative privileges. Generic resource processors (resource processors that are not application-specific and simply process metadata in deployment packages) might unwittingly disrupt the device by processing incorrect or malicious metadata in a deployment package. Application-specific resource processors (e.g. "customizers" consisting of executable code) in a deployment package could also be incorrect or malicious. In order to protect the device, resource processors' capabilities should be limited by the permissions granted to the signer of a deployment package. The MEG Deployment Admin can enforce this limitation by performing two operations:

- The MEG Deployment Admin should construct an `AccessControlContext` that represents the permissions associated with the signer of the deployment package. If the MEG implementation on the device includes the Conditional Permission Admin (which is optional in MEG), the MEG Deployment Admin can invoke `getAccessControlContext()` on Conditional Permission Admin to get such an `AccessControlContext`. Otherwise, the MEG Deployment Admin must construct the AccessControlContext itself, using permission information in the DMT.

- Before calling each resource processor, the MEG Deployment Admin will perform a `doPrivileged(AccessControlContext)` operation. This will limit the permissions of the Resource Processor to the permissions granted to the signer of the Deployment Package.

For example, suppose there are two entities that could deploy deployment packages onto a mobile phone. The wireless carrier AcmeWireless.com wants to deploy games, address book/PIM applications, and other "carrier applications". An enterprise SpacelySprockets.com wants to deploy mobile enterprise applications on a phone.

Among the permissions granted to AcmeWireless.com, as a deployment package signer, might be granted `ConfigurationPermission("com.AcmeWireless.*, SET_ACTION")`. This permission would allow AcmeWireless to configure any managed service provided by it on the platform. In particular, AUTOCONF.XML files (see RFC-94 [5]) contained in deployment packages signed by AcmeWireless.com could configure any managed service provided by AcmeWireless. However, deployment packages signed by AcmeWireless could not be used to configure any managed services provided by SpacelySprockets.

SpacelySprockets might be granted `ConfigurationPermission("com.SpacelySprockets.*",SET_ACTION)` and `ConfigurationPermission("com.AcmeWireless.fooService",SET_ACTION)`. These permissions would allow deployment packages signed by SpacelySprockets to configure any managed service from SpacelySprockets and also the "fooService" provided by AcmeWireless. However, configuring other managed services provided by AcmeWireless.com would not be allowed.

The previous example involved a generic resource processor – a configuration resource processor that processed an RFC-94 AUTOCONF.XML metadata in a deployment package. This security mechanism also applies to application-specific resource processors.

For example, suppose a deployment package contains an application specific "data customizer" resource processor that reformats a database on the device. If this resource processor was allowed to run with full `AdminPermission`, a malicious programmer could tamper with permissions via Permission Admin. With the mechanism outlined above, the resource processor would be limited to the permissions granted to the signer of the deployment package (which presumably would NOT include full `AdminPermission`). So any attempt to manipulate Permission Admin would result in a `SecurityException`.

5.2.7.2 Security considerations for the getDataFile() method

When a bundle in a DP is updated, it might be necessary to manipulate the bundle's private data area so that existing data can be converted into a form usable by the new version of the bundle. The `DeploymentSession` interface provides a method, `getDataFile()`, that provides a customer this access. Since the location of a

All Page Within This Box

bundle's private data area can be different on different frameworks, it is impossible to know a priori the location of a bundle's private data area, and therefore impossible to send an appropriate `FilePermission` along with the customizer/DP. Therefore, before calling the customizer, Deployment Admin must (in an implementation-specific way) figure out the location of the bundle's private data area, construct a `FilePermission` that allows the customizer to access only that directory, and add the `FilePermission` to the set of permissions that the customizer has. When the deployment operation is over, DeploymentAdmin must rescind this `FilePermission`.

5.3 Deployment Package Lifecycle

In this subsection, lifecycle of deployment packages will be discussed in more detail. Three lifecycle operations for deployment packages have been defined: install, update, and uninstall. The implementation of these operations should make all effort to have transactional semantics for the deployment operation. This means that the overall process should be isolated and atomic, but these requirements are optional.

5.3.1 Installing/Updating a Deployment Package

Deployment Admin has a single method, `installDeploymentPackage()`, that can be used for both the initial installation and subsequent updates of a DP. An initial installation of a DP is treated as a special case of updating a DP – an update from a default "empty" or "null" DP. The `DeploymentSession` interface provides a method, `getTargetDeploymentPackage()`, that returns the currently-installed DP. In the case of an update operation, this method will return the current ("old", "stale") DP. In the case of an initial installation, this method will return null.

When updating a deployment package, the first step is to stop all bundles associated with the DP. (Obviously, if this is an initial installation, the current DP will be null, and there will be no bundles to stop.) Once they are all stopped, Deployment Admin needs to figure out:

- Which bundles must be uninstalled
- Which bundles must be updated
- Which bundles must be installed

To compute this, the update operation uses both the current deployment package (which bundles belong to the package before the update operation starts) and the new version of the deployment package being installed. The new version is either a full-format deployment package or a fix-pack. In the case of a full-format package, the new deployment package contains all its resources, bundles and others. In the case of a fix pack, the fix pack still lists all the resources (bundles and otherwise) that make up the fix-pack DP, but some of those resources may not be present in the DP stream. The fact that some resources are missing is signaled by a `DeploymentPackage-Missing` header(Section 5.1.3). The following table describes the processing rules. The most likely installation scenario is highlighted. Note that "update" and "install" are intended to specify OSGi framework operations.

All Page Within This Box

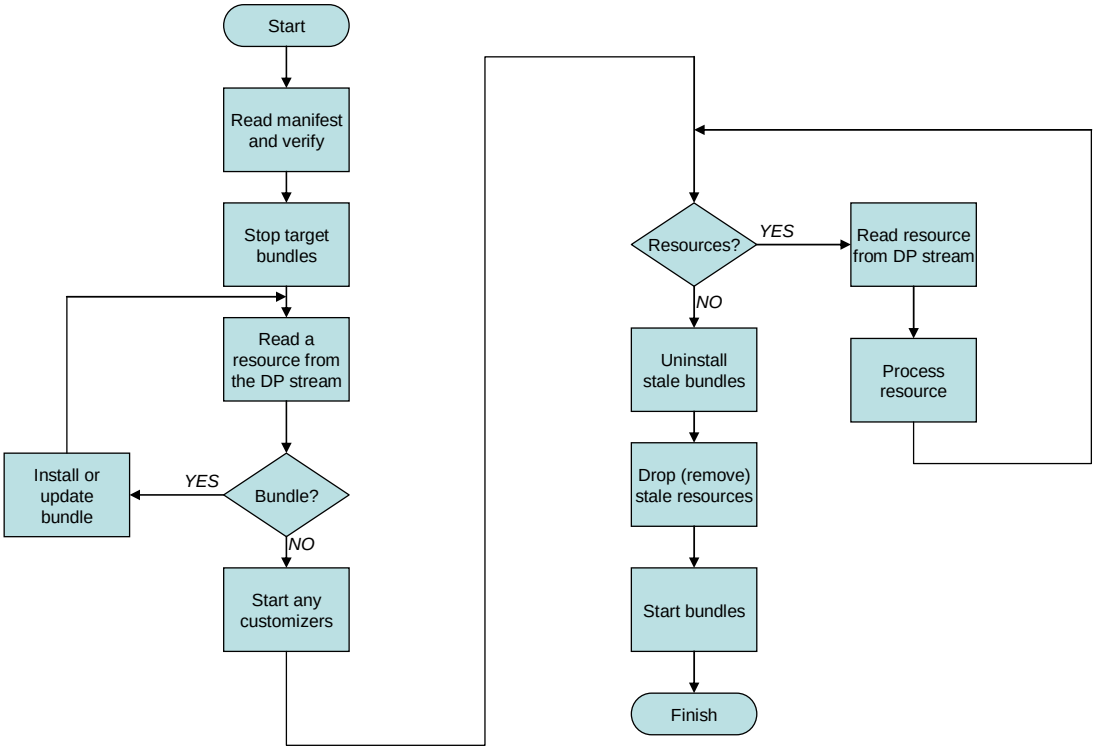
Is this a FixPack DP?	Is the bundle currently installed?	Is the bundle in the DP manifest?	Is the bundle in the DP InputStream?	Action
Y	Y	Y	Y	Update the bundle if the new version is higher, otherwise ignore
Y	Y	Y	N	Leave existing bundle intact
Y	Y	N	Y	Error
Y	Y	N	N	Bundle is no longer part of the DP, mark for uninstall
Y	N	Y	Y	Addition to the DP, install
Y	N	Y	N	Error
Y	N	N	Y	Error
Y	N	N	N	Meaningless case
N	Y	Y	Y	Update the bundle if the new version is higher, otherwise ignore
N	Y	Y	N	Error
N	Y	N	Y	Error
N	Y	N	N	Bundle is no longer part of the DP, mark for uninstall
N	N	Y	Y	Addition to the DP, install
N	N	Y	N	Error
N	N	N	Y	Error
N	N	N	N	Meaningless case

The next step is to call the resource processors associated with any resources that follow the bundles in the DP stream. The details of resource processing were described in Section 5.2. Note that as a result of an update, the set of active processors for a given deployment package may change.

All Page Within This Box

The final steps are to uninstall all bundles marked for removal, and to start the bundles in the DP.

Updating (or installing) a DP is summarized in the following diagram:



5.3.2 Uninstalling a Deployment package

Uninstalling a deployment package involves removing all the effects of its installation. This is achieved through collaboration between the deployment package object and its set of cached resource processors - the resource processors that were invoked at install time. During install, the deployment processor object remembers all the resource processors it has invoked. At uninstall time, it will call those same resource processor services to uninstall any artifacts they might have created.

The first step in the uninstall process is for the deployment package object to stop all the bundles it owns.

The second step is for the deployment package object to call all the resource processors that were used so far (for the original install and subsequent updates of the DP). For the details of resource processing, refer to Section 5.2.

The third and final step is for the deployment package object to uninstall the bundles that it owns.

Uninstalling a deployment package may break the overall runtime configuration. There is no attempt to ensure that a deployment package being uninstalled is not necessary as a provider of Java packages or services. The

uninstall semantic for a deployment package is equivalent to the uninstall semantic for a single bundle: it is mandatory, applies to all resources owned by the deployment package, and may break runtime dependencies.

It is an error condition if the customizing services are no longer present when uninstalling or updating a deployment package. Reacting to this error condition is quite delicate. As a default rule, a request to carry out any such operation on a deployment package must be refused (no uninstall and no updating) until the customizing services are available. However, the case where these services will never be available again needs to be handled. Hence, the `DeploymentPackage` interface provides a method, `uninstallForced()`, that forces removal of a DP. It is likely that the uninstallation may be incomplete, and some residue of deployment package may remain on the platform. Whether this residue is eventually cleaned up, and how, is left up to the implementor.

5.4 Deployment-related Management Objects

Deployment operations may be accessed from within a device's Device Management Tree for remote management. This section defines the management objects used for remote deployment operations and their mapping to the DeploymentAdmin interface.

5.4.1 Node: ./OSGi/deploy

This permanent node is the parent node of all the Deployment Manager management objects.

AccessType: Get

Format: node

Occurrence: One

Scope: Permanent

5.4.2 Node: ./OSGi/deploy/DownloadAndInstall

This interior node is a node on which an Exec can be invoked to initiate a download and an immediate install for the specified deployment package. The URL of the deployment package must be set first with a Replace operation on the PkgURL child node. Then, the installation can be started with an Exec operation.

AccessType: Exec

Format: node

Occurrence: One

Scope: Permanent

5.4.3 Node: ./OSGi/deploy/DownloadAndInstall/PkgURL

This node contains the URL where the installation package or the download descriptor is stored to be used when performing an Exec operation on ./OSGi/deploy/DownloadAndInstall.

All Page Within This Box

AccessType: Replace

Format: chr

Occurrence: One

Scope: Permanent

5.4.4 Node: ./OSGi/deploy/Uninstall

This interior node is a node on which an Exec can be invoked to initiate the immediate uninstallation of a deployment package. The deployment package name must be placed into the PkgName child node with a "Replace" operation first. Then, the deployment package uninstall can be initiated with an "Exec" command on the node.

AccessType: Exec

Format: node

Occurrence: One

Scope: Permanent

5.4.5 Node: ./OSGi/deploy/Uninstall/PkgName

This node contains the symbolic name of the deployment package to be uninstalled.

AccessType: Replace

Format: chr

Occurrence: One

Scope: Permanent

5.4.6 Node: ./OSGi/deploy/Inventory

This node is the parent of inventory-related operations. Its child nodes represent deployment packages.

AccessType: get

Format: node

Occurrence: One

Scope: Permanent

5.4.7 Node: ./OSGi/deploy/Inventory/<DPID>

The nodes under the Inventory parent node represent installed deployment packages. The deployment package ID, shown as <DPID> is the name of the Inventory child node.

AccessType: get

Format: node

Occurrence: ZeroOrMore

Scope: Dynamic

5.4.8 Node: ./OSGi/deploy/Inventory/<DPID>/Name

This node allows querying of the deployment package name as declared in the DeploymentPackage-Name manifest header.

AccessType: get

Format: chr

Occurrence: One

Scope: Dynamic

5.4.9 Node: ./OSGi/deploy/Inventory/<DPID>/Version

This node allows querying of the deployment package version as declared in the DeploymentPackage-Version manifest header.

AccessType: get

Format: chr

Occurrence: One

Scope: Dynamic

5.4.10 Node: ./OSGi/deploy/Inventory/<DPID>/ProcessorBundle

This node exists only if the deployment package declared a resource processor bundle within the package. If present, the value of this management object is the name of the resource processor bundle. Details of the bundle can be obtained by accessing the bundle node, which is the child node of 5.4.12.

AccessType: get

Format: chr

All Page Within This Box

Occurrence: ZeroOrOne

Scope: Dynamic

5.4.11 Node: ./OSGi/deploy/Inventory/<DPID>/Processors

This node exists only if the deployment package declared resource processors with the DeploymentPackage-Processing manifest header. This value of the management node is the value of the DeploymentPackage-Processing header.

AccessType: get

Format: chr

Occurrence: ZeroOrOne

Scope: Dynamic

5.4.12 Node: ./OSGi/deploy/Inventory/<DPID>/Bundles

This node is the parent node of information nodes of bundles belonging to the deployment package having the DPID package ID.

AccessType: get

Format: node

Occurrence: One

Scope: Dynamic

5.4.13 Node: ./OSGi/deploy/Inventory/<DPID>/Bundles/<BUNDLEID>

BUNDLEID represents the ID of the bundles belonging to the deployment package. The <BUNDLEID> is equal to the ID of the bundle.

AccessType: get

Format: node

Occurrence: ZeroOrMore

Scope: Dynamic

Each <BUNDLEID> management object has a number of leaf nodes allowing access to the bundle properties.

All Page Within This Box

Attribute Name	Format	Description
symbolicName	chr	Symbolic name of the bundle
location	chr	Location identifier of the bundle
version	chr	Version of the bundle
state	int	Lifecycle state of the bundle INSTALLED is represented by a value of 0 RESOLVED is represented by a value of 1 STARTING is represented by a value of 2 STOPPING is represented by a value of 3 ACTIVE is represented by a value of 4
importPackage	chr	Comma separated list of the pairs of imported package names and specification versions. The package name and the version are separated by a semicolon.
exportPackage	chr	Comma separated list of the pairs of exported package names and specification versions. The package name and the version are separated by a semicolon.
nativeCode	chr	The value of the Bundle-NativeCode manifest header
applicationDescriptor	chr	The contents of the application descriptor file

All Page Within This Box

5.5 org.osgi.service.deploymentadmin
Interface DeploymentAdmin

public interface **DeploymentAdmin**

This is the interface of Deployment Admin service. The service provides functionality to manage deployment packages. The deployment admin functionality is exposed as a standard OSGi service with no mandatory service parameters.

Each operation requires DeploymentAdminPermission **and in addition to this the appropriate org.osgi.framework.AdminPermission.**

Method Summary

boolean	**cancel**() This method cancels the currently active deployment session.
DeploymentPackage	**getDeploymentPackage**(long id) Get the deployment package instance based on the id of the package.
DeploymentPackage	**installDeploymentPackage**(java.io.InputStream in) Installs a deployment package from an input stream.
DeploymentPackage[]	**listDeploymentPackages**() Lists the deployment packages currently installed on the platform.

Method Detail

5.5.1 installDeploymentPackage

public DeploymentPackage **installDeploymentPackage**(java.io.InputStream in)
throws DeploymentException

Installs a deployment package from an input stream. If a version of that deployment package is already installed and the versions are different, the installed version is updated with this new version even if it is older. If the two versions are the same, then this method simply returns without any action. DeploymentAdminPermission("<filter>", "install") is needed for this operation.

Parameters:
in - The input stream which where the deployment package can be read
Returns:
A DeploymentPackage object representing the newly installed/updated deployment package
Throws:
DeploymentException - if the installation was not successful
See Also:
DeploymentAdminPermission

5.5.2 listDeploymentPackages

public DeploymentPackage[] **listDeploymentPackages**()

All Page Within This Box

Lists the deployment packages currently installed on the platform. DeploymentAdminPermission("<filter>", "list") is needed for this operation.

Returns:

Array of DeploymentPackage objects representing all the installed deployment packages.

See Also:

DeploymentAdminPermission

5.5.3 getDeploymentPackage

public DeploymentPackage **getDeploymentPackage**(long id)

Get the deployment package instance based on the id of the package. DeploymentAdminPermission("<filter>", "list") is needed for this operation.

Parameters:

id - the id of the deployment package to be retrieved

Returns:

The DeploymentPackage for the request id. If there is no deployment package with that id, null is returned.

See Also:

DeploymentAdminPermission

5.5.4 cancel

public boolean **cancel**()

This method cancels the currently active deployment session. This method addresses the need to cancel the processing of excessively long running, or resource consuming install, updates or uninstalls. DeploymentAdminPermission("<filter>", "cancel") is needed for this operation.

Returns:

true if there was an active session and it was successfully cancelled.

See Also:

DeploymentAdminPermission

5.6 org.osgi.service.deploymentadmin

Class DeploymentAdminPermission

java.lang.Object
  └ java.security.Permission
    └ **org.osgi.service.deploymentadmin.DeploymentAdminPermission**

All Implemented Interfaces:

java.security.Guard, java.io.Serializable

---

public class **DeploymentAdminPermission**
extends java.security.Permission

DeploymentAdminPermission controls access to MEG management framework functions. This permission controls only Deployment Admin-specific functions; framework-specific access is controlled by usual OSGi permissions (AdminPermission, etc.) **In addition to DeploymentAdminPermission, the caller of Deployment Admin must hold the appropriate AdminPermissions.** For example, installing a deployment package requires DeploymentAdminPermission to access the installDeploymentPackage method and AdminPermission to access the framework's install/update/uninstall methods.

The permission uses a <filter> string formatted similarly to the filter in RFC 73. The DeploymentAdminPermission filter does not use the id and location filters. The "signer" filter is

All Page Within This Box

matched against the signer chain of the deployment package, and the "name" filter is matched against the DeploymentPackage-Name header.

DeploymentAdminPermission("<filter>", "list")
A holder of this permission can access the inventory information of the deployment packages selected by the <filter> string. The filter selects the deployment packages on which the holder of the permission can acquire detailed inventory information. See DeploymentAdmin.getDeploymentPackage(long) and DeploymentAdmin.listDeploymentPackages().
DeploymentAdminPermission("<filter>", "install")
A holder of this permission can install/upgrade deployment packages if the deployment package satisfies the <filter> string. See DeploymentAdmin.installDeploymentPackage(java.io.InputStream).
DeploymentAdminPermission("<filter>", "uninstall")
A holder of this permission can uninstall deployment packages if the deployment package satisfies the <filter> string. See DeploymentPackage.uninstall().
DeploymentAdminPermission("<filter>", "uninstallForceful")
A holder of this permission can forecfully uninstall deployment packages if the deployment package satisfies the string. See DeploymentPackage.uninstallForceful().
DeploymentAdminPermission("<filter>", "cancel")
A holder of this permission can cancel an active deployment action. This action being cancelled could correspond to the install, update or uninstall of a deployment package satisfies the string. See DeploymentAdmin.cancel()

Wildcards can be used both in the name and the signer (see RFC-95) filers.

See Also:
Serialized Form

Field Summary

static java.lang.String	**ACTION_CANCEL** Constant String to the "cancel" action.
static java.lang.String	**ACTION_INSTALL** Constant String to the "install" action.
static java.lang.String	**ACTION_LIST** Constant String to the "list" action.
static java.lang.String	**ACTION_UNINSTALL** Constant String to the "uninstall" action.
static java.lang.String	**ACTION_UNINSTALL_FORCEFUL** Constant String to the "uninstallForceful" action.

Constructor Summary

DeploymentAdminPermission(java.lang.String name, java.lang.String actions) Creates a new DeploymentAdminPermission for the given name (containing the name of the target deployment package) and action.

Method Summary

boolean	**equals**(java.lang.Object obj)

All Page Within This Box

	Checks two DeploymentAdminPermission objects for equality.
java.lang.String	**getActions**() Returns the String representation of the action list.
int	**hashCode**() Returns hash code for this permission object.
boolean	**implies**(java.security.Permission permission) Checks if this DeploymentAdminPermission would imply the parameter permission.
java.security.PermissionCollection	**newPermissionCollection**() Returns a new PermissionCollection object for storing DeploymentAdminPermission objects.
java.lang.String	**toString**() Returns a string representation of the object.

Methods inherited from class java.security.Permission
checkGuard, getName

Methods inherited from class java.lang.Object
clone, finalize, getClass, notify, notifyAll, wait, wait, wait

Field Detail

5.6.1 ACTION_INSTALL

public static final java.lang.String **ACTION_INSTALL**

Constant String to the "install" action.

See Also:

Constant Field Values

5.6.2 ACTION_LIST

public static final java.lang.String **ACTION_LIST**

Constant String to the "list" action.

See Also:

Constant Field Values

5.6.3 ACTION_UNINSTALL

public static final java.lang.String **ACTION_UNINSTALL**

Constant String to the "uninstall" action.

See Also:

Constant Field Values

All Page Within This Box

5.6.4 ACTION_UNINSTALL_FORCEFUL

public static final java.lang.String **ACTION_UNINSTALL_FORCEFUL**

Constant String to the "uninstallForceful" action.

See Also:

Constant Field Values

5.6.5 ACTION_CANCEL

public static final java.lang.String **ACTION_CANCEL**

Constant String to the "cancel" action.

See Also:

Constant Field Values

Constructor Detail

5.6.6 DeploymentAdminPermission

public **DeploymentAdminPermission**(java.lang.String name,
java.lang.String actions)

Creates a new DeploymentAdminPermission for the given name (containing the name of the target deployment package) and action.

Throws:

java.lang.IllegalArgumentException - if the arguments are incorrect

Method Detail

5.6.7 equals

public boolean **equals**(java.lang.Object obj)

Checks two DeploymentAdminPermission objects for equality. Two permission objects are equal if:

- their "signer" and "name" parts of the name (target deployment package) are equal and

- their actions are the same.

Parameters:

obj - The reference object with which to compare.

Returns:

true if the two objects are equal.

See Also:

Object.equals(java.lang.Object)

5.6.8 hashCode

public int **hashCode**()

Returns hash code for this permission object.

Returns:

Hash code for this permission object.

See Also:

Object.hashCode()

5.6.9 getActions

public java.lang.String **getActions**()

Returns the String representation of the action list.

Returns:

All Page Within This Box

Action list of this permission instance. This is a comma-separated list that reflects the action parameter of the constructor.

See Also:

Permission.getActions()

5.6.10 implies

public boolean **implies**(java.security.Permission permission)

Checks if this DeploymentAdminPermission would imply the parameter permission.

Parameters:

permission - Permission to check.

Returns:

true if this DeploymentAdminPermission object implies the specified permission.

See Also:

Permission.implies(java.security.Permission)

5.6.11 newPermissionCollection

public java.security.PermissionCollection **newPermissionCollection**()

Returns a new PermissionCollection object for storing DeploymentAdminPermission objects.

Returns:

The new PermissionCollection.

See Also:

Permission.newPermissionCollection()

5.6.12 toString

public java.lang.String **toString**()

Returns a string representation of the object.

Returns:

string representation of the object

5.7 org.osgi.service.deploymentadmin

Class DeploymentException

java.lang.Object
  └ java.lang.Throwable
    └ java.lang.Exception
      └ **org.osgi.service.deploymentadmin.DeploymentException**

All Implemented Interfaces:

java.io.Serializable

---

public class **DeploymentException**
extends java.lang.Exception

Checked exception received when something fails during any deployment processes. Beside the exception message, a DeploymentException always contains an error code (one of the constants specified in this class), and may optionally contain the textual description of the error condition and a nested cause exception.

See Also:

Serialized Form

All Page Within This Box

Field Summary

static int	**CODE_BAD_HEADER** Syntax error in any manifest header.
static int	**CODE_BUNDLE_NAME_ERROR** Bundle symbolic name is not the same as defined by the deployment package manifes.
static int	**CODE_BUNDLE_SHARING_VIOLATION** Bundle with the same symbolic name alerady exists.
static int	**CODE_FOREIGN_CUSTOMIZER** Matched resource processor service is a customizer from another deployment package.
static int	**CODE_MISSING_BUNDLE** A bundle in the deployment package is marked as MissingResource-Bundle but there is no such bundle in the target deployment package.
static int	**CODE_MISSING_FIXPACK_TARGET** Fix pack version range doesn't fit to the version of the target deployment package or the target deployment package of the fix pack doesn't exist.
static int	**CODE_MISSING_HEADER** Missing mandatory manifest header.
static int	**CODE_MISSING_RESOURCE** A resource in the deployment package is marked as MissingResource-Resource but there is no such resource in the target deployment package.
static int	**CODE_NO_SUCH_RESOURCE** The dropped(String resource) method was called on the matched resource processor but the resource processor doesn't manage this resource.
static int	**CODE_ORDER_ERROR** The manifest is not the first file in the stream or bundles don't precede resource files.
static int	**CODE_OTHER_ERROR** Other error condition.
static int	**CODE_PREPARE** Resource processors are allowed to raise such exceptions that indicates that the processor is not able to commit the operations it made since the last call of begin method.
static int	**CODE_RESOURCE_SHARING_VIOLATION** A side effect of any resource already exists.
static int	**CODE_SIGNING_ERROR** Bad deployment package signing.

Constructor Summary

DeploymentException(int code) Create an instance of the exception.
DeploymentException(int code, java.lang.String message) Create an instance of the exception.
DeploymentException(int code, java.lang.String message, java.lang.Throwable cause)

All Page Within This Box

Create an instance of the exception.

Method Summary

java.lang.Throwable	**getCause**()
int	**getCode**()
java.lang.String	**getMessage**()

Methods inherited from class java.lang.Throwable
fillInStackTrace, getLocalizedMessage, getStackTrace, initCause, printStackTrace, printStackTrace, printStackTrace, setStackTrace, toString

Methods inherited from class java.lang.Object
clone, equals, finalize, getClass, hashCode, notify, notifyAll, wait, wait, wait

Field Detail

5.7.1 CODE_ORDER_ERROR

public static final int **CODE_ORDER_ERROR**

The manifest is not the first file in the stream or bundles don't precede resource files.

See Also:

Constant Field Values

5.7.2 CODE_MISSING_HEADER

public static final int **CODE_MISSING_HEADER**

Missing mandatory manifest header.

See Also:

Constant Field Values

5.7.3 CODE_BAD_HEADER

public static final int **CODE_BAD_HEADER**

Syntax error in any manifest header.

See Also:

Constant Field Values

5.7.4 CODE_MISSING_FIXPACK_TARGET

public static final int **CODE_MISSING_FIXPACK_TARGET**

All Page Within This Box

Fix pack version range doesn't fit to the version of the target deployment package or the target deployment package of the fix pack doesn't exist.

See Also:

Constant Field Values

5.7.5 CODE_MISSING_BUNDLE

public static final int **CODE_MISSING_BUNDLE**

A bundle in the deployment package is marked as MissingResource-Bundle but there is no such bundle in the target deployment package.

See Also:

Constant Field Values

5.7.6 CODE_MISSING_RESOURCE

public static final int **CODE_MISSING_RESOURCE**

A resource in the deployment package is marked as MissingResource-Resource but there is no such resource in the target deployment package.

See Also:

Constant Field Values

5.7.7 CODE_SIGNING_ERROR

public static final int **CODE_SIGNING_ERROR**

Bad deployment package signing.

See Also:

Constant Field Values

5.7.8 CODE_BUNDLE_NAME_ERROR

public static final int **CODE_BUNDLE_NAME_ERROR**

Bundle symbolic name is not the same as defined by the deployment package manifes.

See Also:

Constant Field Values

5.7.9 CODE_FOREIGN_CUSTOMIZER

public static final int **CODE_FOREIGN_CUSTOMIZER**

Matched resource processor service is a customizer from another deployment package.

See Also:

Constant Field Values

5.7.10 CODE_NO_SUCH_RESOURCE

public static final int **CODE_NO_SUCH_RESOURCE**

The dropped(String resource) method was called on the matched resource processor but the resource processor doesn't manage this resource.

See Also:

Constant Field Values

5.7.11 CODE_BUNDLE_SHARING_VIOLATION

public static final int **CODE_BUNDLE_SHARING_VIOLATION**

Bundle with the same symbolic name alerady exists.

See Also:

Constant Field Values

5.7.12 CODE_RESOURCE_SHARING_VIOLATION

public static final int **CODE_RESOURCE_SHARING_VIOLATION**

A side effect of any resource already exists.

See Also:

Constant Field Values

5.7.13 CODE_PREPARE

public static final int **CODE_PREPARE**

Resource processors are allowed to raise such exceptions that indicates that the processor is not able to commit the operations it made since the last call of begin method.

See Also:

Constant Field Values

5.7.14 CODE_OTHER_ERROR

public static final int **CODE_OTHER_ERROR**

Other error condition.

See Also:

Constant Field Values

Constructor Detail

5.7.15 DeploymentException

public **DeploymentException**(int code,
java.lang.String message,
java.lang.Throwable cause)

Create an instance of the exception.

Parameters:

code - The error code of the failure
message - Message associated with the exception

5.7.16 DeploymentException

public **DeploymentException**(int code,
java.lang.String message)

Create an instance of the exception. Cause exception is implicitly set to null.

Parameters:

code - The error code of the failure
message - Message associated with the exception

5.7.17 DeploymentException

public **DeploymentException**(int code)

Create an instance of the exception. Cause exception and message are implicitly set to null.

Parameters:

code - The error code of the failure

Method Detail

All Page Within This Box

5.7.18 getCause
public java.lang.Throwable **getCause**()

Returns:
Returns the cause.

5.7.19 getCode
public int **getCode**()

Returns:
Returns the code.

5.7.20 getMessage
public java.lang.String **getMessage**()

Returns:
Returns the message.

5.8 org.osgi.service.deploymentadmin
Interface DeploymentPackage

---

public interface **DeploymentPackage**

The DeploymentPackage object represents a deployment package (already installed or being currently processed).

Method Summary

boolean	**equals**(java.lang.Object other) Indicates whether some other object is "equal to" this one.
org.osgi.framework.Bundle	**getBundle**(java.lang.String bundleSymName) Returns the bundle instance that corresponds to the bundle's name/version pair.
java.lang.String[][]	**getBundleSymNameVersionPairs**() Returns an 2D array of strings representing the bundles and their version that are specified in the manifest of this deployment package
java.lang.String	**getHeader**(java.lang.String name) Returns the requested deployment package manifest header from the main section.
long	**getId**() Returns the identifier of the deployment package.
java.lang.String	**getName**() Returns the name of the deployment package.
java.lang.String	**getResourceHeader**(java.lang.String path, java.lang.String header) Returns the requested deployment package manifest header from the name section determined by the path parameter.
org.osgi.framework.ServiceReference	**getResourceProcessor**(java.lang.String resource) At the time of deployment, resource processor service instances are

All Page Within This Box

	located to processor the resources contained in a deployment package.
java.lang.String[]	**getResources**() Returns an array of strings representing the resources that are specified in the manifest of this deployment package
org.osgi.framework.Version	**getVersion**() Returns the version of the deployment package.
int	**hashCode**() Returns a hash code value for the object.
void	**uninstall**() Uninstalls the deployment package.
boolean	**uninstallForceful**() This method is called to completely uninstall a deployment package, which couldn't be uninstalled using traditional means due to exceptions.

Method Detail

5.8.1 getId

public long **getId**()

Returns the identifier of the deployment package. Every installed deployment package has its own unique identifier.

Returns:

The ID of the resource package.

5.8.2 getName

public java.lang.String **getName**()

Returns the name of the deployment package.

Returns:

The name of the deployment package.

5.8.3 getVersion

public org.osgi.framework.Version **getVersion**()

Returns the version of the deployment package.

Returns:

version of the deployment package

5.8.4 getBundleSymNameVersionPairs

public java.lang.String[][] **getBundleSymNameVersionPairs**()

Returns an 2D array of strings representing the bundles and their version that are specified in the manifest of this deployment package

Returns:

The 2d string array corresponding to bundle symbolic name and version pairs

---

5.8.5 getBundle

public org.osgi.framework.Bundle **getBundle**(java.lang.String bundleSymName)

All Page Within This Box

Returns the bundle instance that corresponds to the bundle's name/version pair. This method will return null for request for bundle/version pairs that are not part of this deployment package. As this instance is transient, this method may return null if the bundle/version pair is part of this deployment package, but is not currently defined to the framework.

Returns:

The deployment package instance for a given bundle name/version pair.

5.8.6 getResources

public java.lang.String[] **getResources**()

Returns an array of strings representing the resources that are specified in the manifest of this deployment package

Returns:

The string array corresponding to resources

5.8.7 getResourceProcessor

public org.osgi.framework.ServiceReference **getResourceProcessor**(java.lang.String resource)

At the time of deployment, resource processor service instances are located to processor the resources contained in a deployment package. This call returns a service reference to the corresponding service instance. If this call is made during deployment, prior to the locating of the service to process a given resource, null will be returned. Services can be updated after a deployment packahge has been deployed. In this event, this call will return a reference to the updated service, not to the instance that was used at deployment time.

Returns:

resource procesor for the resource

5.8.8 getHeader

public java.lang.String **getHeader**(java.lang.String name)

Returns the requested deployment package manifest header from the main section. Header names are case insensitive.

Parameters:

name - the requested header

Returns:

the value of the header

5.8.9 getResourceHeader

public java.lang.String **getResourceHeader**(java.lang.String path,
java.lang.String header)

Returns the requested deployment package manifest header from the name section determined by the path parameter. Header names are case insensitive.

Returns:

the value of the header

5.8.10 uninstall

public void **uninstall**()
throws DeploymentException

Uninstalls the deployment package. After uninstallation, the deployment package object becomes stale. This can be checked by using DeploymentPackage:getId(), which will return a -1 when stale.

Throws:

`DeploymentException` - if the deployment package could not be successfully uninstalled.

All Page Within This Box

5.8.11 uninstallForceful

public boolean **uninstallForceful**()

This method is called to completely uninstall a deployment package, which couldn't be uninstalled using traditional means due to exceptions.

Returns:

true if the operation was successful

5.8.12 hashCode

public int **hashCode**()

Returns a hash code value for the object.

Returns:

a hash code value for this object

5.8.13 equals

public boolean **equals**(java.lang.Object other)

Indicates whether some other object is "equal to" this one. Two deployment packages are equal if they have the sam name and version.

Parameters:

other - the reference object with which to compare.

Returns:

true if this object is the same as the obj argument; false otherwise.

5.9 org.osgi.service.deploymentadmin
Interface DeploymentSession

---

public interface **DeploymentSession**

The Session interface represents a currenty running Deployment session and provides access to information/objects involved in the processing of the currently being deployed DP.

Field Summary

static int	**INSTALL** The action value INSTALL indicates this session is associated with the installation of a deployment package.
static int	**UNINSTALL** The action value UNINSTALL indicates this session is associated with the uninstalling of a deployment package.
static int	**UPDATE** The action value UPDATE indicates this session is associated with the update of a deployment package.

Method Summary

java.io.File	**getDataFile**(org.osgi.framework.Bundle bundle) Returns the private data area descriptor area of the specified bundle.

All Page Within This Box

int	**getDeploymentAction**() Returns whether this session is doing an install, an update, or an uninstall.
DeploymentPackage	**getSourceDeploymentPackage**() If the deployment ation is an install or an update, this call returns the DeploymentPackage instance that corresponds to the DP being streamed in for this session.
DeploymentPackage	**getTargetDeploymentPackage**() If the deployment ation is an update or an uninstall, this call returns the DeploymentPackage instance for the installed DP.

Field Detail

5.9.1 INSTALL

public static final int **INSTALL**

The action value INSTALL indicates this session is associated with the installation of a deployment package.

See Also:

Constant Field Values

5.9.2 UPDATE

public static final int **UPDATE**

The action value UPDATE indicates this session is associated with the update of a deployment package.

See Also:

Constant Field Values

5.9.3 UNINSTALL

public static final int **UNINSTALL**

The action value UNINSTALL indicates this session is associated with the uninstalling of a deployment package.

See Also:

Constant Field Values

Method Detail

5.9.4 getDeploymentAction

public int **getDeploymentAction**()

Returns whether this session is doing an install, an update, or an uninstall. While this can be determined by inspecting the source and target DP, providing this action saves RPs from going through the trouble of doing so.

5.9.5 getTargetDeploymentPackage

public DeploymentPackage **getTargetDeploymentPackage**()

If the deployment ation is an update or an uninstall, this call returns the DeploymentPackage instance for the installed DP. If the deployment action is an install, version 0 of the DP is created that contains no bundles and no resources.

All Page Within This Box

5.9.6 getSourceDeploymentPackage

public DeploymentPackage **getSourceDeploymentPackage**()

If the deployment ation is an install or an update, this call returns the DeploymentPackage instance that corresponds to the DP being streamed in for this session. Since the session would not be created until after the manifest has been read in, all the necessary information will be available to create this object. If the deployment action is an uninstall, this call returns the empty DP.

5.9.7 getDataFile

public java.io.File **getDataFile**(org.osgi.framework.Bundle bundle)

Returns the private data area descriptor area of the specified bundle. The bundle's name/version must be part of either the source or target deployment packages.

5.10 org.osgi.service.deploymentadmin
Interface ResourceProcessor

---

public interface **ResourceProcessor**

ResourceProcessor interface is implemented by processors handling resource files in deployment packages. The ResourceProcessor interfaces are exported as OSGi services. Bundles exporting the service may arrive in the deployment package (customizers) or may be preregistered. Resource processors has to define the service.pid standard OSGi service property which should be a unique string.

Method Summary

void	**begin**(DeploymentSession session) Called when the Deployment Admin starts a new operation on the given deployment package, and the resource processor is associated a resource within the package.
void	**cancel**() Processing of a resource passed to the resource processor may take long.
void	**commit**() Called when the processing of the current deployment package is finished.
void	**dropAllResources**() This method is called during an "uninstall" deployment session.
void	**dropped**(java.lang.String name) Called when a resource, associated with a particular resource processor, had belonged to an earlier version of a deployment package but is not present in the current version of the deployment package.
void	**prepare**() This method is called on the Resource Processor immediately before calling the commit method.
void	**process**(java.lang.String name, java.io.InputStream stream) Called when a resource is encountered in the deployment package for which this resource processor has been selected to handle the processing of that resource.
void	**rollback**() Called when the processing of the current deployment package is finished.

All Page Within This Box

Method Detail

5.10.1 begin

public void **begin**(DeploymentSession session)

Called when the Deployment Admin starts a new operation on the given deployment package, and the resource processor is associated a resource within the package. Only one deployment package can be processed at a time. It is important to note that the deployment action (i.e. install, update, uninstall) that is part of the deployment session instance passed to the begin method corresponds to what is happening to the resources being processed by this RP. For example, an "update" of a DP, can result in the "install" of a new resource.

Parameters:

session - object that represents the current session to the resource processor

---

5.10.2 process

public void **process**(java.lang.String name,
java.io.InputStream stream)
throws DeploymentException

Called when a resource is encountered in the deployment package for which this resource processor has been selected to handle the processing of that resource.

Parameters:

name - The name of the resource relative to the deployment package root directory.

stream - The stream for the resource.

Throws:

`DeploymentException` - if the resource cannot be processed.

5.10.3 dropped

public void **dropped**(java.lang.String name)
throws DeploymentException

Called when a resource, associated with a particular resource processor, had belonged to an earlier version of a deployment package but is not present in the current version of the deployment package. This provides an opportunity for the processor to cleanup any memory and persistent data being maintained for the particular resource. This method will only be called during "update" deployment sessions.

Parameters:

name - Name of the resource being dropped from the deployment package.

Throws:

`DeploymentException` - if the resource is not allowed to be dropped.

5.10.4 dropAllResources

public void **dropAllResources**()
throws DeploymentException

This method is called during an "uninstall" deployment session. This method will be called on all RPs that are associated with resources in the DP being uninstalled. This provides an opportunity for the processor to cleanup any memory and persistent data being maintained for the deployment package.

Throws:

`DeploymentException` - if all resources could not be dropped.

5.10.5 prepare

public void **prepare**()

All Page Within This Box

throws DeploymentException

This method is called on the Resource Processor immediately before calling the commit method. The Resource Processor has to check whether it is able to commit the operations since the last begin method call. If it determines that it is not able to commit the changes, it has to raise a DeploymentException.

Throws:

`DeploymentException` - if the resource processor is able to determine it is not able to commit.

5.10.6 commit

public void **commit**()

Called when the processing of the current deployment package is finished. This method is called if the processing of the current deployment package was successful, and the changes must be made permanent.

5.10.7 rollback

public void **rollback**()

Called when the processing of the current deployment package is finished. This method is called if the processing of the current deployment package was unsuccessful, and the changes made during the processing of the deployment package should be removed.

5.10.8 cancel

public void **cancel**()

Processing of a resource passed to the resource processor may take long. The cancel() method notifies the resource processor that it should interrupt the processing of the current resource. This method is called by the DeploymentAdmin implementation after the DeploymentAdmin.cancel() method is called.

6 Considered Alternatives

See CVS for previous versions of RFC 88.

7 Security Considerations

Section 5.2.7 addresses the security concerns associated with resource processors.

All Page Within This Box

8 Document Support

8.1 References

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

[2]. Java Language Specification, 2nd Edition, Chapter 7 Packages. http://java.sun.com/docs/books/jls/second_edition/html/packages.doc.html.

[3]. J2SE 1.3 JAR File Specification, http://java.sun.com/j2se/1.3/docs/guide/jar/jar.html.

[4]. RFP 53 MEG Deployment, http://membercvs.osgi.org/rfps/rfps-0053_MEG_WS_Deployment.doc. Version 0.9, April 23, 2004.

[5]. RFC 94 MEG Deployment Configuration, http://membercvs.osgi.org/rfcs/rfc0094/rfc-0094-MEG Deployment Configuration.doc.

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

8.2 Acronyms and Abbreviations

DM – Device Management

DMT – Device Management Tree

DP – Deployment Package

DPID – Deployment Package Identifier

OMA – Open Mobile Alliance

OTA – Over-the-Air

8.3 End of Document

All Page Within This Box