



## **RFC 91 MEG Application Model**

Draft

63 Pages

### **Abstract**

The purpose of this document is to explain application model design, API and samples that address the requirements specified in RFP-052.

Copyright © OSGi Alliance 2005.

This contribution is made to the OSGi Alliance as MEMBER LICENSED MATERIALS pursuant to the terms of the OSGi Member Agreement and specifically the license rights and warranty disclaimers as set forth in Sections 3.2 and 12.1, respectively.

All company, brand and product names contained within this document may be trademarks that are the sole property of the respective owners.

The above notice must be included on all copies of this document that are made.

---

# 0 Document Information

---

## 0.1 Table of Contents

<b>0 Document Information .....</b>	<b>2</b>
0.1 Table of Contents .....	2
0.2 Terminology and Document Conventions .....	4
0.3 Revision History .....	4
<b>1 Introduction .....</b>	<b>6</b>
<b>2 Application Domain .....</b>	<b>7</b>
2.1 Terminology .....	7
<b>3 Problem Description .....</b>	<b>8</b>
<b>4 Requirements .....</b>	<b>9</b>
<b>5 Technical Solution .....</b>	<b>9</b>
5.1 Application Model Architecture .....	9
5.2 Application Management Framework .....	10
5.2.1 Application Descriptor .....	10
5.2.2 Application Handle .....	12
5.2.3 Application Management Framework Functions .....	13
5.2.4 Scheduled Application .....	15
5.2.5 Permissions for Handling Applications .....	16
5.3 The Meglet Model .....	17
5.3.1 The Meglet Base Class .....	17
5.3.2 Lifecycle States of a Meglet .....	17
5.3.3 <i>Developing a Meglet</i> .....	19
5.4 Content Handling .....	26
5.4.1 Content Handler API Implementation on OSGi .....	29
5.5 Internal Interactions .....	29
5.5.1 General Interactions for the Application Management Framework .....	29
5.5.2 Interactions for the Meglets .....	32
5.6 Timer Event .....	37
5.6.1 Timer Event Examples .....	38
5.6.2 Sending Timer Events .....	38
<b>6 API Specification .....</b>	<b>39</b>
6.1 org.osgi.service.application Class ApplicationDescriptor .....	39
6.1.1 APPLICATION_NAME .....	40

6.1.2 APPLICATION_ICON .....	41
6.1.3 APPLICATION_PID .....	41
6.1.4 APPLICATION_VERSION .....	41
6.1.5 APPLICATION_VENDOR .....	41
6.1.6 APPLICATION_SINGLETON .....	41
6.1.7 APPLICATION_AUTOSTART .....	41
6.1.8 APPLICATION_VISIBLE .....	41
6.1.9 APPLICATION_LAUNCHABLE .....	41
6.1.10 APPLICATION_LOCKED .....	41
6.1.11 ApplicationDescriptor .....	41
6.1.12 getPID .....	41
6.1.13 getProperties .....	42
6.1.14 launch .....	42
6.1.15 launchSpecific .....	43
6.1.16 schedule .....	43
6.1.17 lock .....	43
6.1.18 unlock .....	44
6.1.19 isLocked .....	44
6.1.20 getBundleContext .....	44
6.2 org.osgi.service.application Class ApplicationHandle .....	44
6.2.1 STOPPING .....	45
6.2.2 RUNNING .....	45
6.2.3 ApplicationHandle .....	46
6.2.4 getState .....	46
6.2.5 getInstanceID .....	46
6.2.6 getApplicationDescriptor .....	46
6.2.7 destroy .....	46
6.2.8 destroySpecific .....	47
6.3 org.osgi.service.application Interface ScheduledApplication .....	47
6.3.1 getTopic .....	48
6.3.2 getEventFilter .....	48
6.3.3 isRecurring .....	48
6.3.4 getApplicationDescriptor .....	48
6.3.5 getArguments .....	48
6.3.6 remove .....	48
6.4 org.osgi.service.application Class SingletonException .....	49
6.4.1 SingletonException .....	50
6.4.2 SingletonException .....	50
6.5 org.osgi.service.application Class ApplicationAdminPermission .....	50
6.5.1 LAUNCH .....	51
6.5.2 SCHEDULE .....	51
6.5.3 MANIPULATE .....	51
6.5.4 LOCK .....	52
6.5.5 ApplicationAdminPermission .....	52
6.5.6 ApplicationAdminPermission .....	52
6.6 org.osgi.service.application.meglet Class MegletDescriptor .....	52
6.6.1 MegletDescriptor .....	53
6.6.2 getPID .....	53
6.6.3 launchSpecific .....	54
6.6.4 getBundleContext .....	54
6.7 org.osgi.service.application.meglet Class MegletHandle .....	54
6.7.1 SUSPENDED .....	56
6.7.2 MegletHandle .....	56

6.7.3	getState.....	56
6.7.4	getInstanceId.....	56
6.7.5	getApplicationDescriptor.....	56
6.7.6	destroySpecific.....	56
6.7.7	suspend.....	57
6.7.8	resume.....	57
6.8	org.osgi.meglet Class Meglet.....	57
6.8.1	Meglet.....	59
6.8.2	getComponentContext.....	59
6.8.3	start.....	59
6.8.4	stop.....	59
6.8.5	handleEvent.....	59
6.8.6	requestSuspend.....	59
6.8.7	requestStop.....	60
6.8.8	getEventAdmin.....	60
6.8.9	getLogService.....	60
6.8.10	registerForEvents.....	60
6.8.11	unregisterForEvents.....	60
6.8.12	getProperties.....	60
6.8.13	activate.....	60
6.8.14	deactivate.....	61
<b>7</b>	<b>Considered Alternatives.....</b>	<b>61</b>
<b>8</b>	<b>Security Considerations.....</b>	<b>62</b>
<b>9</b>	<b>Document Support.....</b>	<b>62</b>
9.1	References.....	62
9.2	Author's Address.....	63
9.3	Acronyms and Abbreviations.....	63
9.4	End of Document.....	63

---

## 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].

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	Jun 23, 2004	MEG application model conceptual architecture described. Venkat Amirisetty, Motorola, Inc. <a href="mailto:vamirisetty@motorola.com">vamirisetty@motorola.com</a>
Second Draft	Jun 30, 2004	Merged concepts from Nokia proposal. Venkat Amirisetty, Motorola, Inc <a href="mailto:vamirisetty@motorola.com">vamirisetty@motorola.com</a>
Third Draft	July 12, 2004	API specification initial cut added, some comments from Peter Kriens addressed Venkat Amirisetty, Motorola, Inc <a href="mailto:vamirisetty@motorola.com">vamirisetty@motorola.com</a>
Fourth Draft	August 12, 2004	Description of application related stuff, container and application manager based on the API Robert Fajta, Nokia, <a href="mailto:robert.fajta@nokia.com">robert.fajta@nokia.com</a>
Fifth Draft	August 18, 2004	API specification revised. Minor edits made throughout. Venkat Amirisetty, Motorola, Inc <a href="mailto:vamirisetty@motorola.com">vamirisetty@motorola.com</a>
Sixth Draft	September 2, 2004	DMT definition added Robert Fajta, Nokia, <a href="mailto:robert.fajta@nokia.com">robert.fajta@nokia.com</a>
Seventh Draft	September 20, 2004	Model description changed to reflect the concept of generic and convenience layers in Technical Solution chapter. Event section removed and reference to rfc-97 added. DMT definition refined according to the conf call comments. Robert Fajta, Nokia, <a href="mailto:robert.fajta@nokia.com">robert.fajta@nokia.com</a>
8th Draft	September 22, 2004	DMT refined. Robert Fajta, Nokia, <a href="mailto:robert.fajta@nokia.com">robert.fajta@nokia.com</a>
9th Draft	October 1, 2004	Editorial changes in the model description. DMT definition moved to the rfc-87. Robert Fajta, Nokia, <a href="mailto:robert.fajta@nokia.com">robert.fajta@nokia.com</a>
10th Draft – version 0.5	November 15, 2004	Document cleanup. Synchronizing the document contents with other MEG RFCs. Deep edits Venkat Amirisetty, Motorola, <a href="mailto:vamirisetty@motorola.com">vamirisetty@motorola.com</a>

11 <sup>th</sup> Draft – version 0.6	November 25, 2004	Incorporated feedback on version 0.5 from Peter Kriens and Robert Fajta. Added first cut of content handler development model.  Venkat Amirisetty, Motorola, <a href="mailto:vamirisetty@motorola.com">vamirisetty@motorola.com</a>
version 0.7	December 17, 2004	Accommodated comments and decisions from the Boca Raton meeting.  Robert Fajta, Nokia, <a href="mailto:robert.fajta@nokia.com">robert.fajta@nokia.com</a>
version 7.1	January 7, 2005	Minor, mostly editorial corrections.  Robert Fajta, Nokia, <a href="mailto:robert.fajta@nokia.com">robert.fajta@nokia.com</a>
Version 7.2	January 31, 2005	Comments and decisions from the Norwalk, CA meeting  Rick DeNatale, IBM <a href="mailto:denatale@us.ibm.com">denatale@us.ibm.com</a>
Version 0.8	February 11, 2005,	Added suspending to Meglets. Changed scheduling to event based. Added content handling related chapter. Described lifecycle states of Meglets. Updated the considered alternatives chapter.  Robert Fajta, Nokia, <a href="mailto:robert.fajta@nokia.com">robert.fajta@nokia.com</a>
Version 0.81	February 15, 2005	Document review and minor comments  Venkat Amirisetty, Motorola, <a href="mailto:vamirisetty@motorola.com">vamirisetty@motorola.com</a>
Version 0.82	February 21, 2005	Timer event examples corrected. Comments accommodated. Diagrams added. API documentation added. Minor editorial changes.  Robert Fajta, Nokia, <a href="mailto:robert.fajta@nokia.com">robert.fajta@nokia.com</a>
Version 0.83	February 21, 2005	Change formatting in 5.5.  Robert Fajta, Nokia, <a href="mailto:robert.fajta@nokia.com">robert.fajta@nokia.com</a>
Version 0.84	April 1, 2005	Text modified to reflect most of Munich agreements. Content handling chapter updated. The API chapter has not been updated, it is out of sync.  Robert Fajta, Nokia, <a href="mailto:robert.fajta@nokia.com">robert.fajta@nokia.com</a>

---

# 1 Introduction

---

The OSGi service platform is a generic, service centric execution environment. It specifies a framework and a core set of service interfaces that enable delivery of multiple value added service implementations, potentially from different vendors.

The Application model WS is part of the overall Mobile Expert Group (MEG) effort, tasked to define the architecture, API, samples that specify an execution environment for new type of OSGi runtime artifacts, called Applications. Applications are usually UI centric, designed for user interaction, but this RFC does not address the UI aspects.

A Meglet is an application packaged in a bundle that is delivered as part of a deployment package. Meglets have a well-defined lifecycle; they can be started, suspended, resumed and stopped. Meglets can, but are not required to use services. Meglet bundle can also contain packages or services that Meglet developer may want to expose to other Meglets or services.

This RFC addresses the application requirements specified in RFP 52. The event management infrastructure, originally designed to address MEG requirements in this RFC, has been adopted by CPEG and moved into a separate RFC, namely RFC-0097-Generic Event Mechanism [3]. This RFC also depends on RFC-0080-Declarative Services [2]. It is important to go through these RFCs to understand the application model design.

---

## 2 Application Domain

---

In the mobile phone domain Java is very attractive and emerging technology. A well-defined Java application model for such devices is MIDP, a very important but limited platform, specifically designed to make Java available for small application development.

The Mobile Information Device Profile (MIDP) applications have gained significant popularity in the mobile world. Support for Java/MIDP has become a standard feature of modern mobile phones, and downloadable games and applications, provided by service operators, are generating more and more revenue.

Originally OSGi was designed as a very generic and horizontal service framework, with its design focused on flexibility of service deployment and consumption. Aiming at a wide variety of application domains, it could not (and should not) have addressed in detail the question of how to make application development more straightforward and convenient in the OSGi environment. This document introduces a new artifact – Meglet – taking full advantage of OSGi framework's flexibility, while providing a simple development model.

---

### 2.1 Terminology

Terminology is defined in RFP-62. Key concepts defined in this document are:

- **Application:** A software component, which has well-defined entry and exit criteria. Applications can be started and stopped, and usually are designed for user interaction. Applications may be of various types which are defined by application type specifications. Most of these application type specifications are external to the OSGi specification. A particular application is implemented by a class referred to in this specification as the application implementation. The application implementation is not directly visible outside

of the implementation of its run-time environment. Applications and application instances are visible through their application descriptors and application handles.

- **Application Descriptor:** A service that represents an installed application and provides application launching, scheduling and locking features. An application descriptor is registered for each application as long as the application is installed. The application descriptor keeps information about the represented application. The application descriptor services can be obtained from the OSGi service registry. Service properties can be used to select application descriptors by a certain point of view. Application descriptor is an abstract class, and it implements the rules of the framework in a non-overridable fashion.
- **Application Handle:** A service that represents an instance of an application, available in the OSGi service registry as long as the application instance exists. The application handle abstract class implements methods to enforce generic rules for manipulating the represented application. Thus performing security checks on life cycle operations as well as doing resource management actions in a non-overridable fashion.
- **Application Container:** is an entity that consists of a set of objects, which implements a runtime environment for one or more application types within an MEG framework. It must provide a specialized application descriptor and application handle that corresponds to the supported application type. The design of a particular application container is constrained by specifications, which are usually outside of the scope of OSGi and MEG. For example an application container, which implements MIDlets must follow the appropriate JSR specifications for MIDP. Application containers typically provide isolation, security, scope/visibility for the nested applications, and other features required by the specifications they implement but the details of how this is done is irrelevant to the OSGi/MEG specification, and will not be addressed here. The mechanisms by which an application container installs one of its applications are also outside of the scope of this specification. From the point of view of the OSGi/MEG specification, the primary responsibility of a container developer is to provide one or more specialized application descriptor and application handle extensions to support the applications of the specific type.
- **Meglet:** a Meglet is a new application type, which is defined in this specification. Its details are defined by the *Meglet model*. Multiple Meglets can be deployed in a bundle of a deployment package. The MEG framework implementation provides an abstract class that simplifies the programming task of developing this kind of application for OSGi/MEG based devices, this is called as *Meglet base class*. The framework also provides application descriptor and application handle specific to the Meglet model.

---

## 3 Problem Description

---

Applications take center stage in the mobile device environment. They expose critical device functions (access stack, core functions like file system, database, accessories etc.) to the user while providing a rich user interface experience. OSGi provides flexible and efficient code sharing mechanism between bundles, while this is not supported in MIDP environment between MIDlets from different MIDlet Suites. Bundles can also provide services to other bundles dynamically. This feature of OSGi helps creating large-scale applications, but dynamics makes developing bundles more complex than MIDlets. To hide this complexity this specification introduces an application management framework and builds a Meglet model onto it, which helps simplifying application development in the OSGi framework.



---

## 4 Requirements

---

Requirements can be found in RFP-052 [4].

---

## 5 Technical Solution

---

This document addresses the application model requirements specific to an OSGi-based application model for mobile devices by:

- Introducing a new role of Application Developer to OSGi target developers
- Defining an application management framework, which provides abstract classes and services for performing high-level application management.
- Defining a model for a new type of application called Meglet, which provides convenience for application developers who wish to exploit the OSGi environment. This model is built on the application management framework.
- Leveraging bundles and deployment packages for all modes of Meglet delivery to the device, both server assisted and offline.
- Leveraging the OSGi/MEG event mechanism for event subscription and consumption.
- Leveraging the OSGi/MEG declarative services mechanism for service dependency declaration and service access.

---

### 5.1 Application Model Architecture

The application model defines two main concepts:

- An Application Management Framework and
- A Meglet model.

The application management framework defines how to represent applications and their instances and how to administer them in an application type agnostic manner.

Different types of applications can be accommodated in the application management framework. The implementation of an application type typically follows an external specification such as the MIDP specification. The implementer of such a specification can allow such applications to participate in the Application Management Framework by registering a specialized application descriptor for each installed application of the supported type,

and a specialized application handle implementation for each instance of such an application. The application management framework provides generic manipulation of the applications lifecycle with control over resource and security policies via application descriptor and application handle base implementations.

The Meglet model is meant to be used by application developers for the OSGi platform. The model describes how to develop applications of Meglet type. It provides a Meglet base class to offer convenience for the developer with a set of convenience methods that simplify the dynamics of the OSGi. The convenience methods do not prohibit Meglet developer to use other APIs and features in the Application Management Framework and OSGi as necessary. To create a Meglet, the developer has to extend the `org.osgi.meglet.Meglet` base class, create an XML file to describe the Meglet, refer to this XML file from the bundle manifest file, pack the whole into a bundle, and finally put the bundle into a deployment package to get installed onto the device.

## 5.2 Application Management Framework

The application management framework has the following important artifacts available as services: application descriptor, application handle and scheduled application. They will be introduced in the subsequent chapters.

### 5.2.1 Application Descriptor

Applications appear to the user as executable entities, usually associated with an icon, name, etc. Each application in the OSGi environment is represented by an application descriptor service. When an application is installed then a new application descriptor will be created and registered into the OSGi service registry. When the application is being uninstalled then the corresponding application descriptor will be unregistered. An application descriptor provides information about the represented application and specialized application descriptors may offer additional methods to perform operations related to the application model of the represented application.

An application descriptor has methods and service properties to provide information about the represented application. Each application has its own `String` PID, which is a unique identifier within the OSGi environment. It is available as the `service.pid` service property of the application descriptor.

The `getProperties(String locale)` method [ref] of the application descriptor service provides access to localized and non-localized information about the application in a `java.util.Map`. If no locale specific value of an application property is available then the default one must be returned. It is possible to obtain raw (non-localized) data by specifying an empty `String` ("") as the locale. The following case sensitive key names are treated as standard for locale specific values in the `Map`. Additional elements may also be stored in the `Map` but these names must not be used for any other purposes.

Key name	Value type	Value Restriction	Mandatory	Description
<code>application.name</code>	<code>String</code>		yes	Contains the localized name of the application according to the locale string passed to the <code>getProperties(String locale)</code> method [ref].
<code>application.icon</code>	<code>URL</code>		yes	Specifies the path of the icon's image file that contains the localized image within the bundle's jar file according to the locale string passed to the <code>getProperties(String locale)</code> method [ref]. The framework may support any kind of image format but supporting the

PNG image format is mandatory.

Most of the service properties returned by the `getProperties(String locale)` method are locale independent, they are listed below. The names below are treated as standard service properties of the application descriptor service. OSGi environment should not allow modification of the values of these properties by applications. Additional service properties may also be stored but these names must not be used for any other purposes.

Key name	Value type	Value Restriction	Mandatory	Description
<code>service.pid</code>	String	unique on the device	yes	Unique identifier of the application. It is recommended to set a value generated from the vendor's reverse domain name, e.g. <code>com.acme.application.chess</code> . The <code>service.pid</code> service property is defined by the OSGi.
<code>application.version</code>	String		no	Specifies the version of the application. The default value is an empty string.
<code>service.vendor</code>	String		no	Specifies the vendor of the application. The default value is an empty string.
<code>application.singleton</code>	Boolean	true or false	no	Specifies whether only one instance of the application can run at the same time. The default value is <code>true</code> .
<code>application.autostart</code>	Boolean	true or false	no	Specifies whether the application has to be started automatically when it gets installed and at the startup time of the OSGi Environment. The default value is <code>false</code> .
<code>application.visible</code>	Boolean	true or false	yes	Specifies whether the application should be visible for the user. For example, some applications may provide features to other applications but nothing directly to the user. In this case the application should not be revealed to the user to start it individually. The default value is <code>true</code> .
<code>application.launchable</code>	Boolean	true or false	yes	Specifies whether the application is ready to be launched. If the value is <code>true</code> , it means that all the requirements of the application are fulfilled. The default value is <code>false</code> .

application.locked	Boolean	true or false	yes	Specifies whether the represented application is locked to prevent launching it. The default value is <code>false</code> .
--------------------	---------	---------------------	-----	--

The returned Map will contain the standard OSGi service properties as well (e.g. `service.id`, `service.vendor` etc.) and specialized application descriptors may offer further service properties. The returned Map contains a snapshot of the properties. It will not reflect further changes in the property values nor will the update of the Map change the corresponding service property.

## 5.2.2 Application Handle

An application handle is an OSGi service that represents an instance of an Application. The application handle is registered by the application descriptor after successfully launching a new application instance.

Each application instance has a unique identifier within the framework called Instance ID. This identifier can be queried by the `getInstanceID()` method of the application handle and it is available in the `service.pid` service property. Depending on the application model and the container implementation, it is possible that the lifecycle of an application instance spreads through multiple restarts of the framework. The Instance ID of such application instances must be preserved through the restarts. Once the application instance is destroyed, its Instance ID must not be reused (at least in reasonable timeframe).

An application handle can be used to query the application instance lifecycle state and manipulate the application instance. It is the responsibility of the application handle to maintain the application instance lifecycle state by interacting with the application's implementation object. This is possible because the application handle implementation is provided as part of the application container, which understands the details of the application type.

The application handle also includes a method to query the application descriptor, which was used to launch the application instance.

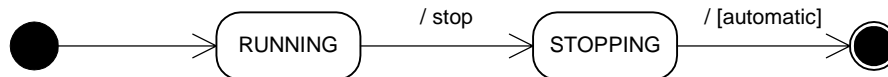
The application handle service maintains the following service properties. Specialized application handles may offer further service properties, but the key names specified in the table below must not be used for other purposes.

Key name	Value type	Value Restriction	Mandatory	Description
<code>service.pid</code>	String	Must be unique	Yes	Contains the unique instance of the service that can also be queried by the <code>getInstanceID()</code> method of the application handle.
<code>application.state</code>	String	Must be a valid state	Yes	Contains the current state of the application instance represented by this application handle. The value of the states must be qualified names, non-qualified names are reserved.
<code>application.descriptor</code>	String	Must refer to an existing	Yes	Specifies the path of the icon's image file that contains the localized image within the bundle's jar file according to

		descriptor's service.pid property		the locale string passed to the <code>getProperties(String locale)</code> method [ref]. The framework may support any kind of image format but supporting the PNG image format is mandatory.
--	--	---	--	---

### 5.2.2.1 Lifecycle States

The application handle service maintains the service property called `application.state` to reflect the current lifecycle state of the represented application instance. Its value is one of `RUNNING` and `STOPPING`. When the application handle is registered the value of this property is `RUNNING`. Upon calling the application handle's `destroy()` method it will be set to `STOPPING`. When the application instance has stopped the application handle will be unregistered. According to the above, the application lifecycle state transitions must be compliant with the following state diagram.



Specialized application handles may offer further application states. The value of the states must be qualified names, non-qualified names are reserved.

Lifecycle state transitions of an application instance can be observed as changes on its application handle service. These changes are advertised through OSGi ServiceEvents, described in [ref to 5.6.2.3. ServiceEvent of RFC 97], thus a client interested in lifecycle state changes can subscribe for these ServiceEvents.

When an application is started successfully the corresponding application handle service will be registered, and an `"org.osgi.framework.ServiceEvent.REGISTERED"` event will be fired. So clients interested in application starting have to subscribe to this event. The registered service can be found by searching for an application handle service by matching the value of the event's `service.id` property and the service's `service.id` property.

If an application instance is stopping, then its application handle's state property will be changed to `STOPPING` and an `"org.osgi.framework.ServiceEvent.MODIFIED"` event will be fired. An interested client can listen to this event. The related application handle can be found by searching for an application handle service by matching the value of the event's `service.id` property and the service's `service.id` property.

An interested client may want to know that an application completed its stopping process. In this case it can listen to `"org.osgi.framework.ServiceEvent.UNREGISTERED"` events.

A particular application model may have additional states that are represented on the `application.state` property. State transitions to or from those states can also be observed by listening to `"org.osgi.framework.ServiceEvent.MODIFIED"` events.

### 5.2.3 Application Management Framework Functions

The application management framework must:

- Provide methods for locking, managing application's lifecycle and scheduling of applications. The framework uses application descriptors, and application handles to provide these features.

- Implement execution policies in a way that they cannot be circumvented.

### 5.2.3.1 Application Locking

Applications represented by the application descriptors can be locked. If an application is locked then no new instance of the represented application can be started until it is unlocked. The locking state of the application has no effect on the already launched instance(s). The Application Descriptor provides methods `lock()` [ref] and `unlock()` [ref] to set and unset the locking state. Locking and unlocking an application represented by an application descriptor requires the proper permission as described in [ref]. The methods to lock and unlock the locked status of an application are implemented as final methods of the abstract application descriptor class to ensure that an application container implementation will not be able to circumvent this security policy.

### 5.2.3.2 Launching an Application

The Application Admin provides `launch(ApplicationDescriptor appDesc, Map args)` method [ref] to launch an application and create an application handle which is registered as a service and returned to the caller. The method in the application descriptor abstract class, which is used to launch the application is declared final to prevent application containers from circumventing the framework policies. This implementation must check if the caller can launch the application represented by the application descriptor. If all the conditions are met then it has to create a new application instance and start it according to the application container's specification. Once it is successfully started the new application handle is registered to represent the newly launched instance and returned. The state exposed on the application handle representing an application instance gets into `RUNNING` state right after the application instance successfully created and started. In this `RUNNING` state, the application instance can operate normally and it can use the functionality of the runtime environment. The detailed steps can be found in [ref]. The application handle is described in [ref].

The `Map` argument of the `launch` method contains startup arguments for the application. The keys used in the `Map` can be standard or application specific. MEG defines the `"org.osgi.triggeringevent"` key to be used to pass the triggering event to a scheduled application (see [ref]), however in the future it is possible that other well-known keys will be defined. To avoid unwanted clashes of keys, the following rules should be applied:

- The keys starting with the dash ("-") character are application specific, no well-known meaning should be associated with them.
- Well-known keys should follow the reverse domain name based naming. In particular, the keys standardized in OSGi should start with `"org.osgi."`.

### 5.2.3.3 Stopping an Application Instance

An application instance can be stopped by calling the `destroy()` method [ref] of the corresponding application handle service with the proper (`ApplicationAdminPermission "manipulate"`) permission [ref]. At this point the state exposed by the application handle of the application instance turns to `STOPPING`. The application instance may save its persistent data before termination and it must release all the resources and must stop its threads. The application instance's artifacts should not be reused any more. The framework should assure (even forcefully) that all allocated resources are cleaned up and the corresponding threads are terminated.

If the application instance completes its stopping process then its application handle will be unregistered.

The application instances supported by a container must be stopped when the container is stopped.

#### 5.2.3.4 Scheduling an Application

The intention of this feature is to launch a new application instance when the specified event of any kind occurs. Launching a new instance may fail in some circumstances as described below. The application may schedule itself, as well as, it can be scheduled by an external application.

The application descriptor enables scheduling an application at the occurrence of the specified event with the `schedule(Map args, String topic, String eventFilter, boolean recurring)` method (described in [ref]) that returns the scheduled application service. The first parameter defines the startup arguments for the scheduled application. The `topic` parameter is a filter for the topic of the specified event. The next parameter is an LDAP filter for the event. If the last `recurring` parameter is false then the application will be launched only once, when the event firstly occurs. If the parameter is true then launching will take place for every event occurrence; i.e. it is a recurring schedule. An example is `schedule(messengerStartupArgs, "com.acme.MessageReceived", "(sender=Joe)", true)` schedules the messenger application to be launched when a message is received from Joe.

Applications can assume that application management framework makes reasonable effort to launch scheduled applications in a timely manner. However, launching is not guaranteed, implementations may drop events if it is necessary in order to preserve the stability and integrity of the device.

If an event would launch multiple applications then the order of launching is not defined, it is implementation specific.

If scheduling event occurs and the target application is not installed the scheduling must not be removed automatically, even if the event is non-recurring. If a schedule is non-recurring and the specified event has not occurred yet, or the scheduling is recurring then event subscription should not be removed automatically even if the application is uninstalled. Once it is installed again with the same unique identifier the system will be able to launch the reinstalled application based on the previous scheduling. If the application is not reinstalled for a while then e.g. the user may permit the system to cancel the scheduling.

Launching a scheduled application is constrained by the same rules as in case of usual application launch. Thus, attempting to launch a locked scheduled application on the specified event will fail until the application gets unlocked. A scheduled application must not be launched on the specified event if it already has an instance and the application is singleton, so scheduling attempt will fail and the corresponding non-recurring schedule must be removed. If the scheduling is non-recurring and launching a new instance fails then when the specified event occurs again launching the application must be attempted again. A non-recurring scheduling on a timer event (timer event is described in this document in 5.6) should be removed automatically after the first successful launching of the specified application, even if the event filter defines more than one point of time.

The triggering event will be delivered to the starting application instance as an additional item identified by the `"org.osgi.triggeringevent"` key among its startup parameters. This key must not be used for other purposes in the startup parameters. To ensure that no events are leaked to applications without the appropriate permission, the event is delivered in a `GuardedObject`, where the guarding permission is the `TopicPermission` for the topic to which the event was posted.

Scheduling and unscheduling an application or retrieving information about scheduled applications requires the `(ApplicationAdminPermission "schedule" <target>)` on the target application to be scheduled. If the target is the unique identifier of the scheduling application itself then it can schedule itself. In addition, the scheduling entity must have `TopicPermission` for the specified topic.

#### 5.2.4 Scheduled Application

The `ScheduledApplication`, returned by the `schedule(Map args, String topic, String eventFilter, boolean recurring)` method, is a service and represents a scheduled application. The



scheduled applications can be obtained from the service registry. The `ScheduledApplication` provides methods to retrieve information about the represented scheduled application, its startup parameters, the event topic, the event filter and the recurring mode of the scheduling. It provides the `remove()` method to discard the scheduling.

### 5.2.5 Permissions for Handling Applications

The application management framework uses `ApplicationAdminPermission` to control access to application manipulation APIs. `ApplicationAdminPermission` is a subclass of `BasicPermission`. This permission is granted to an application (maybe through its bundle) and allows the application to do certain actions on other applications or application instances. The `ApplicationAdminPermission` permission has two parameters. The first parameter is optional, and describes the target on which the action is performed. If this parameter is not specified then all the possible entities will be included. The second one describes the action and is mandatory.

The following table lists the actions that can be controlled with `ApplicationAdminPermission` and when to check if the action is granted.

Action	Meaning	Check in
launch	The application is allowed to launch another application. The target for this permission is the PID of an application descriptor, or nothing if it is allowed to launch all the other applications.	<code>ApplicationDescriptor.launch()</code>
schedule	The application is allowed to schedule another application. The target for this permission is the PID of an application descriptor, or nothing if it is allowed to schedule all the other applications. It implies the launch permission for the same target.	<code>ApplicationDescriptor.schedule()</code>
manipulate	The application is allowed to manipulate the lifecycle state of an application instance. The target for this permission is the PID of an application descriptor of which instances are manipulated, or nothing if all the other applications' instances are allowed to be manipulated. This permission is not necessary for manipulate itself. It implies the launch permission for the same target.	<code>ApplicationHandle.destroy()</code>
lock	The application is allowed to set/unset the locking state of other applications. The target for	<code>ApplicationDescriptor.lock()</code> <code>ApplicationDescriptor.unlock()</code>



this permission is the PID of an application descriptor, or nothing if it is allowed to lock/unlock all the other applications. This permission is not necessary to set/unset its own application's locking state.

The caller must be able to retrieve the application descriptor and application handle service objects from the service registry to access their methods.

## 5.3 The Meglet Model

The Meglet model defines a new application type called Meglet. The model implementation will provide an application container for Meglets including specialized application descriptor and application handle implementations called Meglet application descriptor and Meglet application handle.

Meglets are packaged in bundles and bundles are deployed as parts of deployment packages. A bundle, carrying one or more Meglets, must include metadata in the form of an XML file (called meglet descriptor XML file), in addition to the expected class files and bundle manifest file. Meglets can use services and exported Java packages. A bundle carrying Meglets may offer services and export packages, as well. A Meglet must have a unique identifier, provided by the developer.

### 5.3.1 The Meglet Base Class

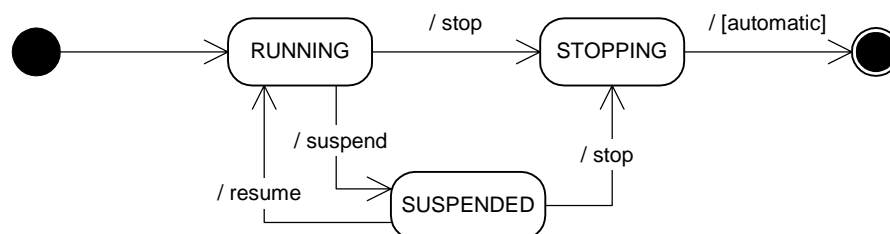
A Meglet must extend the abstract `org.osgi.meglet.Meglet` base class provided by the implementation of the Meglet model.

The rationale is to provide a simple, easy to use development model to the developers, so the Meglet base class provides convenience functions. The developer has to override only a few methods and provide methods to implement the application specific functions of the Meglet. In addition the runtime infrastructure of Meglets exploits the declarative services runtime (called Service Component Runtime – SCR) [ref] to simplify the Meglet programmer's full access to dynamically available OSGi services.

The Meglet application type also provides a lifecycle model which allows Meglets to be suspended and resumed, during which time the used resources can be reduced.

### 5.3.2 Lifecycle States of a Meglet

The Meglet model supports the **RUNNING** and **STOPPING** states, defined by the application management framework, and defines an additional state called **SUSPENDED**. The valid state transitions of a Meglet are shown on the figure below:



An instance of a Meglet can be started, suspended, resumed and stopped. For this purpose the Meglet base class contains the `start(Map args, InputStream stateStorage)` and `stop(OutputStream stateStorage)` methods.

The expected behavior is that when a Meglet instance is being suspended it saves its inner state, cleans up its resources and stops. Although The Meglet instance itself is destroyed, its Meglet application handle remains registered to represent the suspended Meglet instance. During the resume operation a new Meglet instance will be created and started and during the startup it can restore the state that was saved on suspend. The detailed behavior is specified below.

### 5.3.2.1 Starting a Meglet Instance

When a brand new instance of a Meglet is started then the `launch(Map args)` method of the Meglet application descriptor that represents the Meglet will call its `launchSpecific(Map args)` method. Then it will call the `start(Map args, InputStream stateStorage)` method of the newly created Meglet instance where the `stateStorage` parameter must be `null`. If a Meglet instance is started successfully, its self-initialization performed successfully then its Meglet application handle will be registered and will expose that it is in `RUNNING` state. In this state the Meglet instance can operate normally. `RUNNING` state can be achieved when the Meglet instance started successfully or returning from `SUSPENDED` state.

If launching a Meglet fails then all of the resources that may have occupied by the starting instance should be reclaimed. No Meglet application handle will be registered.

### 5.3.2.2 Suspending a Meglet Instance

To suspend a Meglet the Meglet application handle's `suspend()` method is called which will call the `destroySpecific()` method. It will call the Meglet instance's `stop(OutputStream stateStorage)` method. The Meglet instance can ask for being suspended, too, by calling the `requestSuspend()` method inherited from the Meglet base class which should call the `suspend()` method of the application handle asynchronously. The output stream passed in the parameter must not be `null`. The Meglet instance can use the output stream to save its inner state for restoring purpose at some later time. The Meglet instance is not mandated to close the output stream. The information stored in the output stream must be preserved persistently across framework restarts for restoring the Meglet instance at some later point. After stopping the Meglet instance, represented by its Meglet application handle, it will not exist any more. The Meglet application handle remains registered to indicate that the represented Meglet instance has been suspended (though the Meglet instance itself has been destroyed). The suspended Meglet instance must not have any resources, except the inner state information saved to the output stream. If the Meglet instance is destroyed when it is in the `SUSPENDED` state then the information about its inner state can be discarded. A Meglet instance can get to `SUSPENDED` state from the `RUNNING` state.

If the suspending operation fails then the Meglet instance may be terminated forcefully, maybe even immediately. In this case its state exposed on its Meglet application handle must be set to `STOPPING` and the Meglet instance must be terminated. Then the Meglet application handle must be unregistered.

### 5.3.2.3 Resuming a Meglet Instance

When a suspended Meglet instance is resumed then its application handle's `resume()` method is called which calls the `launchSpecific(Map args)` method. It will create a new Meglet instance and call its `start(Map args, InputStream stateStorage)` method. The `args` parameter must `null`. The input stream parameter must not be `null` and must contain exactly the same data to restore the inner state in the new instance that was saved by the corresponding Meglet instance when it was suspended. The input stream may be empty if the suspending Meglet instance saved no inner state data. The Meglet is not mandated to close the input stream. If

the new instance started successfully and restored its inner state the state exposed on the application handle must be set to `RUNNING`.

If the resuming operation fails then the Meglet instance may be stopped or kept in the `SUSPENDED` state.

#### 5.3.2.4 Stopping a Meglet Instance

When a Meglet instance is stopped then its Meglet application handle's `destroy()` method is called which calls the `destroySpecific()` method. The Meglet instance can ask for being stopped by calling the `requestStop()` method inherited from the Meglet base class which should call the `destroy()` method of the Meglet handle asynchronously. Then the state exposed on its application handle must be turned to `STOPPING`.

`STOPPING` state can be reached either from `RUNNING` or `SUSPENDED` state. If the `STOPPING` state is reached from `RUNNING` state then the `destroy()` method will call through `destroySpecific()` the Meglet instance's `stop(OutputStream stateStorage)` method. Its output stream parameter must be `null`, so the Meglet instance cannot save its inner state as in case of suspending. If the application instance finished its stopping process then its application handle gets unregistered. If the `STOPPING` state is reached from `SUSPENDED` state then the Meglet instance's `stop(OutputStream stateStorage)` method should not be called. The Meglet application handle gets unregistered.

If stopping a Meglet instance fails then it may be terminated forcefully and even immediately. After a forceful termination the corresponding Meglet application handle must be unregistered.

A stopped Meglet instance must not assume that any of its left pieces (e.g. unterminated threads) can run any more or still occupied resources will remain occupied.

#### 5.3.2.5 Special Rules

If a Meglet is locked and it has a suspended instance then creating a new instance for resuming must be allowed. If the Meglet is a singleton application and an instance of it has been started and is now in `SUSPENDED` state then creating a new instance for resuming must be allowed.

### 5.3.3 Developing a Meglet

An application developer can implement his/her own class that is a subclass of the Meglet base class. He/she must implement the `start(Map args, InputStream stateStorage)` [ref] and `stop(OutputStream stateStorage)` [ref] methods to customize the application startup and shutdown. The `start(Map args, InputStream stateStorage)` method is called with `null stateStorage` parameter when a brand new instance of the Meglet is started or a non-`null stateStorage` parameter when a Meglet is being resumed. The method accepts a parameter for passing startup arguments as key-value pairs. The developer can also override the `handleEvent(Event event)` method to deal with events to which it has subscribed.

#### 5.3.3.1 Using services

Meglets can use services provided by bundles. The recommended way to access a service is the following.

The developer has to declare in the Meglet descriptor XML file in the recommended way that he/she wants to use a service, as described in [ref]. Additionally the developer has to add two methods to his/her code to receive and release the service, they are known as binder and unbinder methods.

The binder method is called prior to activation, as described at [ref], i.e. prior to starting the application. In the binder method the service received as method parameter can be stored in a member field. The unbinder method

is called after deactivation, i.e. after stopping the application. In the unbind method the stored reference(s) of the service must be set to null.

### 5.3.3.2 Dealing with Events

The event mechanism is based on the Generic Event Mechanism defined in [ref]. Meglets can produce and consume events if the appropriate rights are granted.

Meglets can produce Meglet specific events. The framework ensures that all the listeners interested in the event get notified. According to the Generic Event Mechanism events can be sent synchronously or asynchronously if the specific right granted to the sender.

Consuming events does not require access to the `EventAdmin` service. To consume an event the Meglet has to subscribe to the interested event with a specific topic and react to the incoming events. The Meglet base class offers a default implementation of the `handleEvent(Event event)` method defined in [ref] which can be overridden in the subclasses to handle the events. Subscription can be done dynamically or statically.

Dynamic event registration means that the Meglet subscribes to an event during its execution. For this the Meglet has to perform the `registerForEvent(String topic, String eventFilter)` call during its execution. From this point the Meglet will receive all the events with the specified topic. Unsubscription happens with the `unregisterForEvent(String topic, String eventFilter)` method and the parameter's value must be equal to the parameter's value that was passed during subscription. Dynamic event registration must be discarded if the Meglet instance is stopped or suspended.

Static event registration is also possible. In this case the Meglet developer has to specify in the Meglet descriptor XML file that the Meglet is interested in some event [ref]. It is possible to specify the action to be performed for a specific event:

- Handle: the instance(s) of the application (if any) will receive the event.
- Start: a brand new instance of the application will be launched and then the event will be delivered to that new instance. If the application is a singleton and already has an instance then that instance will receive the event for handling.
- Stop: all the instances of the application will be stopped.

If some failure occurs then a log record may be created about it by the framework. For static event registration it is not necessary to have any instance of the application.

### 5.3.3.3 Meglet Descriptor File

The developer has to describe the Meglets of a bundle in the Meglet descriptor XML file, called `meglets.xml` file. Based on the information provided about the Meglet(s) of a bundle to the framework the application descriptor(s) will be registered when the Meglets are getting installed.

The developer has to insert a section into every `<application>` section to define a component factory service for each Meglet to hold its dependencies on services, as it is described in [ref to rfc 80]. Once, these dependencies are fulfilled then the component factory service assigned to a Meglet becomes available in the OSGi service registry automatically. Thus it helps tracking the fulfillment of the dependencies of the corresponding Meglet.

The manifest header `Service-Component` must be specified in the bundle manifest file and the Meglet descriptor XML file must be inserted into the comma-separated list of XML resources to be handled by the SCR.

The properties of a Meglet described in the Meglet descriptor file are stored in the service properties of the corresponding application descriptor. A service property with the same name is registered on the Meglet application descriptor with the value provided here.

#### 5.3.3.4 XML Schema for the Meglet Descriptor File

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:app="http://www.osgi.org/xmlns/app/v1.0.0"
  xmlns:xs="http://www.w3.org/2001/XMLSchema"
  xmlns:scr="http://www.osgi.org/xmlns/scr/v1.0.0"
  targetNamespace="http://www.osgi.org/xmlns/app/v1.0.0"
  elementFormDefault="qualified"
  attributeFormDefault="unqualified" version="1.0.0">
  <xs:import namespace="http://www.osgi.org/xmlns/scr/v1.0.0"
    schemaLocation="file://C:/schemas/DeclServ.xsd"/>
  <xs:element name="descriptor" type="app:descriptorType">
    <xs:annotation>
      <xs:documentation>descriptor element encloses the application
        descriptors provided in a document</xs:documentation>
    </xs:annotation>
  </xs:element>

  <xs:complexType name="applicationType">
    <xs:annotation>
      <xs:documentation>describes an application</xs:documentation>
    </xs:annotation>
    <xs:sequence>
      <xs:element name="locale" type="app:localeType" maxOccurs="unbounded"/>
      <xs:element name="version" type="app:versionType"/>
      <xs:element name="vendor" type="app:vendorType" minOccurs="0"/>
      <xs:element name="singleton" type="app:singletonType" minOccurs="0"/>
      <xs:element name="autostart" type="app:autostartType" minOccurs="0"/>
      <xs:element name="visibility" type="app:visibilityType" minOccurs="0"/>
      <xs:element name="subscribe" type="app:subscribeType" minOccurs="0"/>
      <xs:element ref="component"/>
    </xs:sequence>
    <xs:attribute name="pid" type="xs:string"/>
  </xs:complexType>

  <xs:complexType name="localeType">
    <xs:annotation>
      <xs:documentation>container for locale specific data e.g. name or icon
        of the application. One element describes one locale.
      </xs:documentation>
    </xs:annotation>
    <xs:sequence>
      <xs:element name="name" type="app:nameType"/>
      <xs:element name="icon" type="app:iconType"/>
    </xs:sequence>
    <xs:attribute name="name" type="xs:string"/>
    <xs:attribute name="default" type="xs:boolean" use="optional"
      default="false"/>
  </xs:complexType>
```

```
<xs:complexType name="versionType">
  <xs:attribute name="value" type="xs:string"/>
</xs:complexType>

<xs:complexType name="vendorType">
  <xs:attribute name="value" type="xs:string"/>
</xs:complexType>

<xs:complexType name="singletonType">
  <xs:attribute name="value" type="xs:boolean" default="true"/>
</xs:complexType>

<xs:complexType name="autostartType">
  <xs:attribute name="value" type="xs:boolean" default="false"/>
</xs:complexType>

<xs:complexType name="visibilityType">
  <xs:attribute name="value" type="xs:boolean" default="true"/>
</xs:complexType>

<xs:simpleType name="actionType">
  <xs:restriction base="xs:string">
    <xs:enumeration value="start"/>
    <xs:enumeration value="stop"/>
    <xs:enumeration value="handle"/>
  </xs:restriction>
</xs:simpleType>

<xs:complexType name="subscribeType">
  <xs:attribute name="event" type="xs:string"/>
  <xs:attribute name="action" type="app:actionType"/>
</xs:complexType>

<xs:complexType name="nameType">
  <xs:attribute name="value" type="xs:string"/>
</xs:complexType>

<xs:complexType name="iconType">
  <xs:attribute name="value" type="xs:string"/>
</xs:complexType>

<xs:complexType name="descriptorType">
  <xs:sequence>
    <xs:element name="application" type="app:applicationType"
      maxOccurs="unbounded"/>
  </xs:sequence>
</xs:complexType>
</xs:schema>
```

The following elements can be found in the schema:

- `<application>`: defines an application. Mandatory for each Meglet in the bundle which is to be installed by the meglet container when the bundle is installed.

- `<locale>`: The locale string for the application name and icon to which they will belong. One of the locales can be marked with `default="true"` attribute to indicate that it is the default. If no matching locale were specified on the device then the items of the default can be used. If no default is specified then items found in the first locale section will be used. The locale section must occur at least once.
  - `<name>`: The name of the application to be shown to the user.
  - `<icon>`: Path to the icon's image resource file.
- `<version>`: The version of the application. Optional.
- `<vendor>`: The name of the vendor of the application. Optional.
- `<singleton>`: Defines if the application is a singleton. Optional, by default it is `true`.
- `<autostart>`: Defines if the application has to be started automatically when it gets installed or when the framework started up. Optional, by default it is `false`.
- `<visibility>`: Some applications may provide facilities to other applications and not to the user. This element defines if the application should be revealed to the user in order to launch. Optional, by default it is `true`.
- `<subscribe>`: Application can subscribe statically to events using the `event` attribute to describe the event's topic. The `action` attribute's value is one of `"start"`, `"stop"` and `"handle"`. It specifies what will happen if the event occurs as described in [ref]. This element is optional
- `<xs:element ref="component"/>` is a placeholder for a component element. It describes the dependencies of the application on services. The XML schema of this component description is defined in [ref to rfc 80]. It is recommended to use the following skeleton to describe the component:

```

1  <scr:component name="<<fully-qualified-application-class-name>>"
2    factory="<<application-pid>>"
3    autoenable=true
4    xmlns:scr="http://www.osgi.org/xmlns/scr/v1.0.0">
5    <implementation class="<<fully-qualified-application-class-name>>" />
6    <reference name="log_service"
7      interface="org.osgi.service.log.LogService"
8    />
9    <reference name="event_service"
10     interface="org.osgi.service.event.EventChannel"
11   />
12   <reference name="<<any-name>>"
13     interface="<<fully-qualified-service-interface-name>>"
14     bind="<<binder-method-name>>"
15     unbind="<<unbinder-method-name>>"
16   />
17 </scr:component>

```

The developer has to change the items marked with `<< >>` signs in this description according to the followings:



- `<<fully-qualified-application-class-name>>` is the fully qualified name of the class of developer's application, which is a subclass of Meglet [ref]. For example `com.wombat.applications.MessagingApplication`.
- `<<application-pid>>` is the unique identifier of the Meglet, specified by the developer [ref to appPID].
- `<<any-name>>` is the logical name of the required service, as this application knows. It can be any name, but it must be unique for this application.
- `<<fully-qualified-service-interface-name>>` is the fully qualified name of the interface provided by the required service, for example `org.osgi.service.LogReaderService`.
- `<<binder-method-name>>` the name of the method implemented in the application's class by the developer, which has the signature: `protected void <<binder-method-name>>(<<fully-qualified-service-interface-name>>)`. This method will be called immediately prior to calling `activate(ComponentContext)`, its intended functionality is to store the received service in a field, thus making it available for the entire life of the application.
- `<<unbinder-method-name>>` the name of the method implemented in the application's class by the developer, which has the signature: `protected void <<unbinder-method-name>>(<<fully-qualified-service-interface-name>>)`. This method will be called immediately after calling `deactivate(ComponentContext)`, its intended functionality is to set to `null` all the references to the service, which was stored by the binder method. In most of the cases it means that the member field, which stored the service is set to `null`.

The `<reference>` section at line 12 may appear zero or more times, depending on the number of the required services.

Though the developer can modify the proposed component description according to the RFC-80, it is recommended to follow this proposal. Modifying it may change the intended behavior of accessing services, which may have a deep impact on the behavior of the developer's Meglet, and may require the developer to deeply modify his/her code.

### 5.3.3.5 Example for Meglet Descriptor File

The following example shows the content of a possible `meglets.xml` file where two applications are described. The first one depends on the `LogReaderService`, the second one has no service dependencies.

```
<?xml version="1.0" encoding="UTF-8"?>
<descriptor xmlns="http://www.osgi.org/xmlns/app/v1.0.0"
xmlns:scr="http://www.osgi.org/xmlns/scr/v1.0.0">
  <application pid="com.acme.game.backgammon">
    <locale name="en">
      <name value="Backgammon"/>
      <icon value="/Backgammon.gif"/>
    </locale>
    <version value="1.0.0"/>
    <vendor value="Acme Inc."/>
    <singleton value="true"/>
    <autostart value="false"/>
    <visibility value="true"/>
    <scr:component xmlns="" factory="com.acme.game.backgammon"
```



```

        autoenable="true" name="com.acme.game.backgammonMeglet" >
        <implementation class="com.acme.game.backgammon.BackgammonMeglet"/>
        <reference name="log_service"
            interface="org.osgi.service.log.LogService"/>
        <reference name="event_service"
            interface="org.osgi.service.event.EventChannel"/>
        <reference name="logreader_service"
            interface="org.osgi.service.log.LogReaderService"
            bind="bindLogReader" unbind="unbindLogReader"/>
    </scr:component>
</application>
<application pid="com.acme.game.chess">
    <locale name="en" default="true">
        <name value="Chess"/>
        <icon value="/ChessEn.gif"/>
    </locale>
    <locale name="de">
        <name value="Schach"/>
        <icon value="/ChessDe.gif"/>
    </locale>
    <version value="1.3.1"/>
    <vendor value="Acme Inc."/>
    <singleton value="true"/>
    <autostart value="false"/>
    <visibility value="true"/>
    <subscribe action="start"
        event="com.acme.game.online.ChallengeReceivedEvent"/>
    <scr:component xmlns="" factory="com.acme.game.chess" autoenable="true"
        name="com.acme.game.chess.ChessMeglet">
        <implementation class="com.acme.game.chess.ChessMeglet"/>
        <reference name="event"
            interface="org.osgi.service.event.EventChannel"/>
    </scr:component>
</application>
</descriptor>

```

### 5.3.3.6 Bundle Containing Meglet

If a bundle contains one or more Meglets then it should not have a BundleActivator class. In this case its declarative services and the component factory service will be registered into the OSGi service registry without loading the bundle. Loading the bundle will be delayed until the first request for one of its services. If a bundle activator is added then the bundle will consume resources even if its services are not used at all.

If a bundle contains one or more Meglets then it must contain the Meglet descriptor XML file called `meglets.xml`. This file contains the metadata of the Meglet(s).

The manifest header `Service-Component` must be specified in the bundle manifest file (`/META-INF/MANIFEST.MF` in the bundle's jar file), the value of this header is a comma-separated list of component descriptor XML files within the bundle to be handled by the SCR. The `meglets.xml` must be inserted into this list.

The bundle must also contain the compiled code and resources for the Meglet(s).

Beyond the above the bundle can contain any usual element, e.g. further resources.

### 5.3.3.7 Availability of Meglets

When a bundle, that contains one or more Meglets described in a `meglets.xml` file, is installed then a Meglet application descriptor service [ref] will be registered into the OSGi service registry for each Meglet. The application descriptor service represents the installed Meglet for the rest of the system, and keeps the properties of the Meglet as its service properties. Most of these properties are extracted from the corresponding part of the `meglets.xml` file. If someone is interested in the appearance of Meglets then he can listen to the appearance of the Meglet application descriptor. At this point the application exists in the system but it is not launchable, because fulfilling its dependencies may not be possible. Thus the application descriptor's `application.launchable` service property is set to `false`.

Initially, the `application.visible` property of the Meglet application descriptor is set to `false`. When the bundle of the meglet is started then it can be set to `true` for those applications for which the same attribute in the `meglets.xml` is `true` (or unspecified). When bundle is stopped, this attribute is set to `false`. As the next step the bundle containing the Meglet is activated. When all the dependencies of a Meglet are fulfilled SCR will register a component factory service to the OSGi service registry. The corresponding Meglet application descriptor's `application.launchable` property will be set to `true`. The Meglet is now available for launching as long as it is not locked (for locking see [ref]). When a Meglet is launched a new instance of it is created by the corresponding component factory service's `newInstance()` method. Then the Meglet instance's `start(Map args, InputStream stateStorage)` method is called so the Meglet can initialize itself.

If any of the services required by the Meglet disappears then the Meglet's component factory service will be unregistered and the corresponding application descriptor's `application.launchable` service property will be set to `false`. Now the application cannot be launched. The existing Meglet instances may be deactivated as described in RFC 80 Declarative Services document [ref], then the deactivated Meglet instance must be suspended.

When the requirements are fulfilled again then the component factory service will be registered again and the corresponding application descriptor's `application.launchable` property will be set to `true` again, thus the Meglet can be launched.

The value of component factory service's `component.name` and `component.factory` service properties will be set to the value coming from the `meglets.xml`. Thus finding the Meglet application descriptor belonging to the component factory can be done by searching for Meglet application descriptor service of which `service.pid` service property has the same value as the component factory's `component.factory` service property.

---

## 5.4 Content Handler API Implementation

The following subsections define how the optional package defined in JSR 211 Content Handler API [ref] must be implemented if it is implemented on top of OSGi.

### 5.4.1 Introduction

Content Handler API (CHAPI) [ref] is an optional package for the J2ME platform. It allows applications to register themselves as content handlers and to invoke other content handlers. A content handler is any application that has registered itself to be invoked through the API. The registered applications typically handle some content, but this is not required. An application can register itself just to be invocable by other applications without needing to handle any content. Invocations can be based on content URL, content type or content handler ID. Arguments may also be passed to a content handler and the content handler can return results and a status.

CHAPI specification consists of a generic part and a platform specific part. The generic part of the specification applies to all CHAPI implementations, whereas the platform specific part needs to be specified for each Java platform that has its own definitions for application packaging and lifecycle. CHAPI defines the platform specific

part of the specification for Mobile Information Device Profile (MIDP), but leaves it undefined for other platforms. Consequently, a specification for the platform specific parts of CHAPI on OSGi platform is needed.

The following subsections briefly define the platform specific parts of a CHAPI implementation on the OSGi platform. The reader is assumed to be familiar with the Content Handler API specification. The specification here only lists the additional requirements and clarifications that are needed to implement the API in an interoperable way on OSGi.

### 5.4.2 Content Handlers

Strictly speaking, the Content Handler API allows only applications to act as content handlers. This is apparent from the terminology that is used throughout the document. Content handlers are always assumed to be applications. A clear example of this assumption is the description for method `Registry.register()` that the application can use to register itself as a content handler. The first parameter for the method is `java.lang.String classname` that is defined as follows:

- *“classname – the application class name that implements this content handler. The value MUST NOT be null and MUST implement the lifecycle of the Java runtime environment”*

This would suggest, that only MEG applications can act as content handlers on OSGi, but as there does not appear to be major technical reasons for disallowing OSGi services to also act as content handlers, this specification will define how they too can act as content handlers.

Another problem with parameter `classname` is that it cannot be used as a unique service identifier in OSGi. We therefore need to redefine the semantics of the `classname` parameter when using services as content handlers.

### 5.4.3 Content Handler Identification

A Content Handler API implementation in the OSGi-MEG environment must allow both MEGlets and services to be registered as content handlers. This registration can be done either statically or dynamically as defined by the Content Handler API. Static registration attributes are the same as defined in the package description of the Content Handler API and the dynamic registration parameters as defined for method `Registry.register()`. However, the semantics of the `classname` parameter in both static and dynamic registrations must be defined for OSGi to be as follows:

- When a MEGlet or an OSGi service is registered as a content handler, it must define the `service.pid` service property and use its value as the `classname`.

Static registration of a service as a content handler must fail if the `service.pid` service property is undefined. Dynamic registration of a service as a content handler must fail if the `service.pid` service property passed in as a parameter does not represent a service that has been registered into the system.

This definition for the `classname` parameter semantics also applies to methods: `Registry.getRegistry()`, `Registry.getServer()` and `Registry.unregister()`.

For both services and applications, the static registration attributes are placed into the manifest file of the bundle that contains them.

### 5.4.4 Content Handler Access Control

The Content Handler API includes a mechanism allowing content handlers to optionally limit their accessibility to a predefined list of invokers. This mechanism requires that the invokers can define a unique identifier for themselves. CHAPI defines that MIDlets can define this identifier using the attribute `MIDlet-<n>-ID`, where `<n>`

refers to a specific MIDlet within the MIDlet suite JAR. Since this attribute is MIDP specific, we need a corresponding way to identify MEGlets and OSGi services. Fortunately, OSGi already includes a way to identify both services and MEGlets. This can be achieved using the `service.pid` service property.

MEGlets and services that themselves act as content handlers can use the attribute `MicroEdition-Handler-<n>-ID` to identify themselves as defined in the Content Handler API. MEGlets and services that are not content handlers, should define the `service.pid` service property to identify themselves for the access control purposes in the Content Handler API.

In case the MEGlet or service does not define an ID for itself using either attribute `MicroEdition-Handler-<n>-ID` or the `service.pid` service property, the Content Handler API implementation must automatically generate a default ID for the MEGlet or service used with the CHAPI defined access control mechanism. This default ID is generated as follows:

- For MEGlets, it is the name of the application class implementing the MEGlet lifecycle, i.e. the name of the class extending the `org.osgi.meglet.Meglet` base class.
- For services, it is the name of the class implementing the service interface of the service.

### 5.4.5 Method Descriptions

Definitions of some Content Handler API methods also need to be further clarified in an OSGi based Content Handler API implementation. Below a list of these methods and clarifications:

#### 5.4.5.1 *ContentHandler.getAppName()*

For MEGlets, the return value must be the `application.name` property of the application descriptor.

A service acting as a content handler should define a `service.name` property that will contain the service name. This will be used as the return value from this method. If `service.name` is not defined but `service.pid` is defined, the value of `service.pid` will be returned. If none of these properties is defined, the implementation will return an implementation specific value that signifies a service without a name. In some implementations, this could be “-”, in others it could be a localized string describing that the name is unknown, for example, “unknown”.

#### 5.4.5.2 *ContentHandler.getAuthority()*

For both MEGlets and OSGi services, the return value must be the subject of the signing certificate in case the bundle containing the MEGlet or the service has been signed and the signature verification has been successful. Otherwise `null` must be returned.

#### 5.4.5.3 *ContentHandler.getID()*

See section titled “Content Handler Access Control”.

#### 5.4.5.4 *ContentHandler.getVersion()*

For MEGlets, the return value must be the `application.version` property of the application descriptor if it has been defined. Otherwise `null` is returned for a content handler MEGlet.

A service acting as a content handler should define a `service.version` property that will contain the implementation version of the service. This will be used as the return value from this method. If `service.version` is not defined, the implementation must return `null`.

#### *5.4.5.5 Invocation.getID() and Invocation.getInvokingID()*

See section titled "Content Handler Access Control".

#### *5.4.5.6 Invocation.getInvokingAppName()*

Return value is obtained in the same way as for ContentHandler.getAppName().

#### *5.4.5.7 Invocation.getInvokingAuthority()*

Return value is obtained in the same way as for ContentHandler.getAuthority().

#### *5.4.5.8 Registry.getID() and Registry.getIds()*

See section titled "Content Handler Access Control".

#### *5.4.5.9 Registry.getRegistry(), Registry.getServer(), Registry.register(), Registry.unregister()*

See section titled "Content Handler Identification".

---

## **5.5 Internal Interactions**

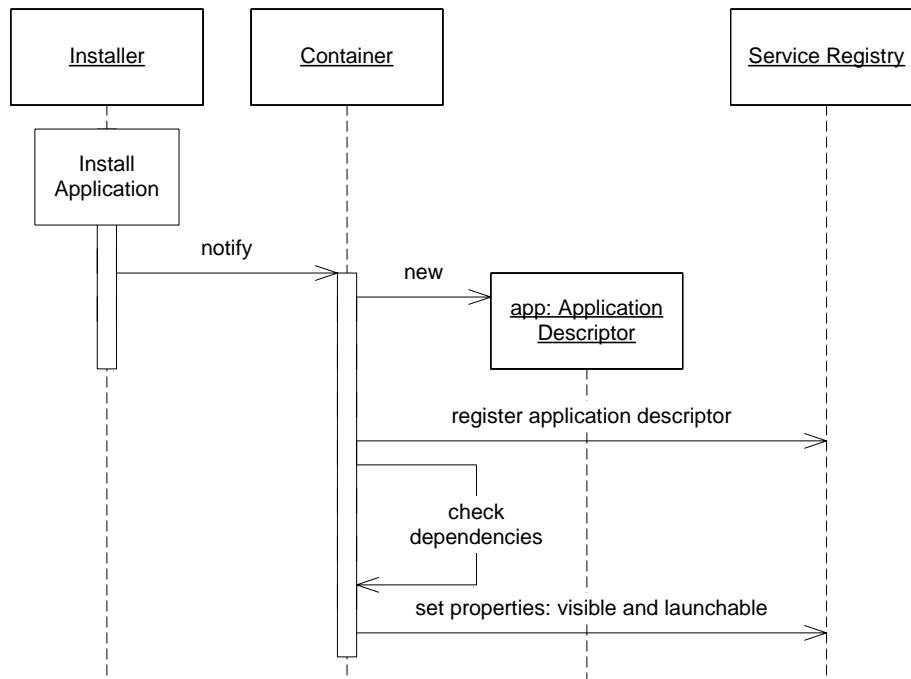
The content of this chapter is informal. It introduces a possible way of operation to help understand the overall system. On the figures there is a Lifecycle Manager that is a conceptual entity who initiates the illustrated operations. For example, it can be human user acting via an application, a management application or a local agent of a remote management server.

### **5.5.1 General Interactions for the Application Management Framework**

The following diagrams illustrate a possible way of the operation of the application management framework in a general case.

#### *5.5.1.1 Application Installation*

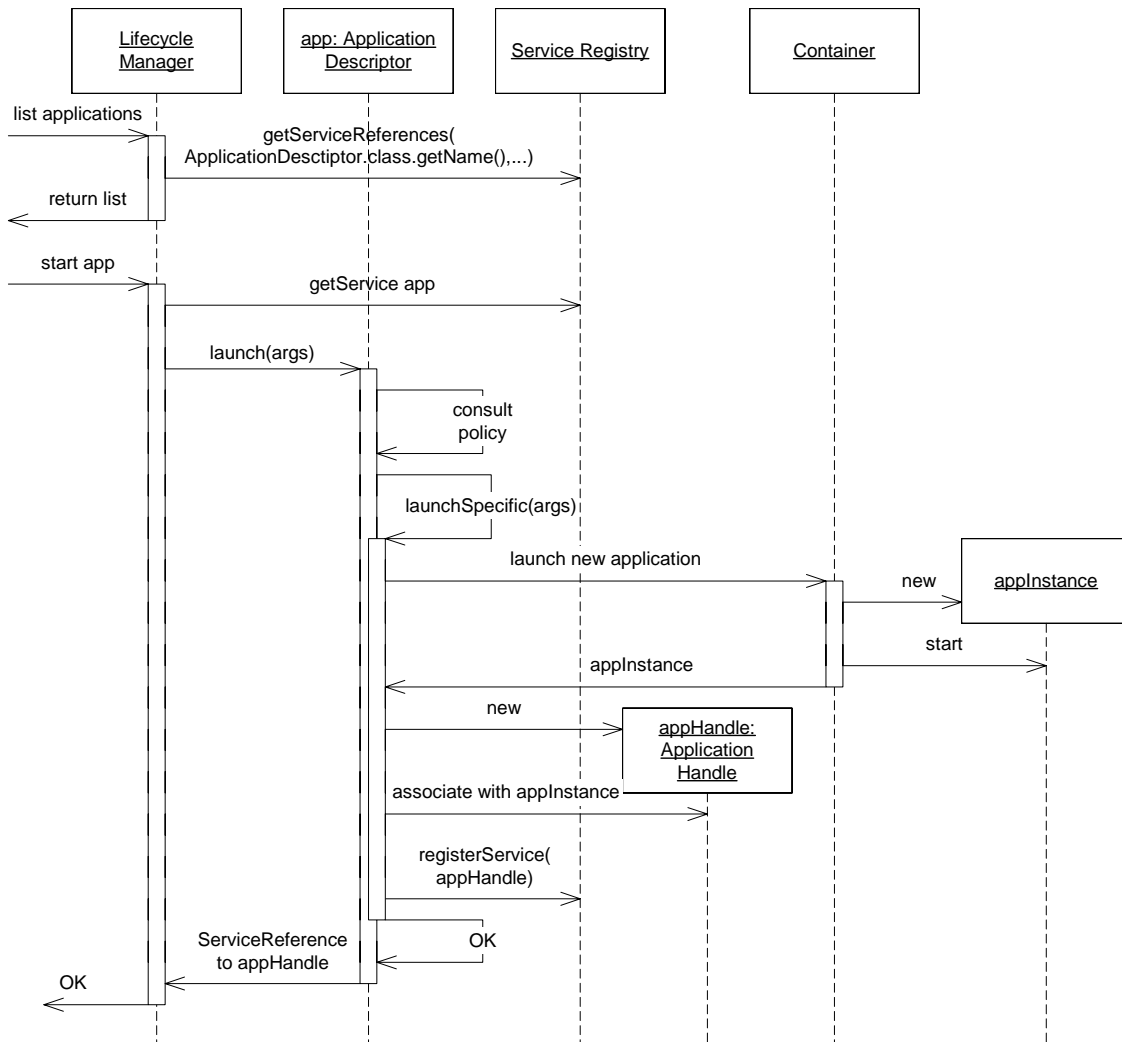
On the figure below an entity of the device notifies the proper application container about the installation of a new application. The actual installation may be done prior to the notification or may be done by the application container. At the end of the successful installation the container must register a specialized application descriptor which properly represents the installed application. If the installed application's dependencies are fulfilled (if any) then the application descriptor's `application.visible` and `application.launchable` properties should be set to true.



### 5.5.1.2 Launching an Application

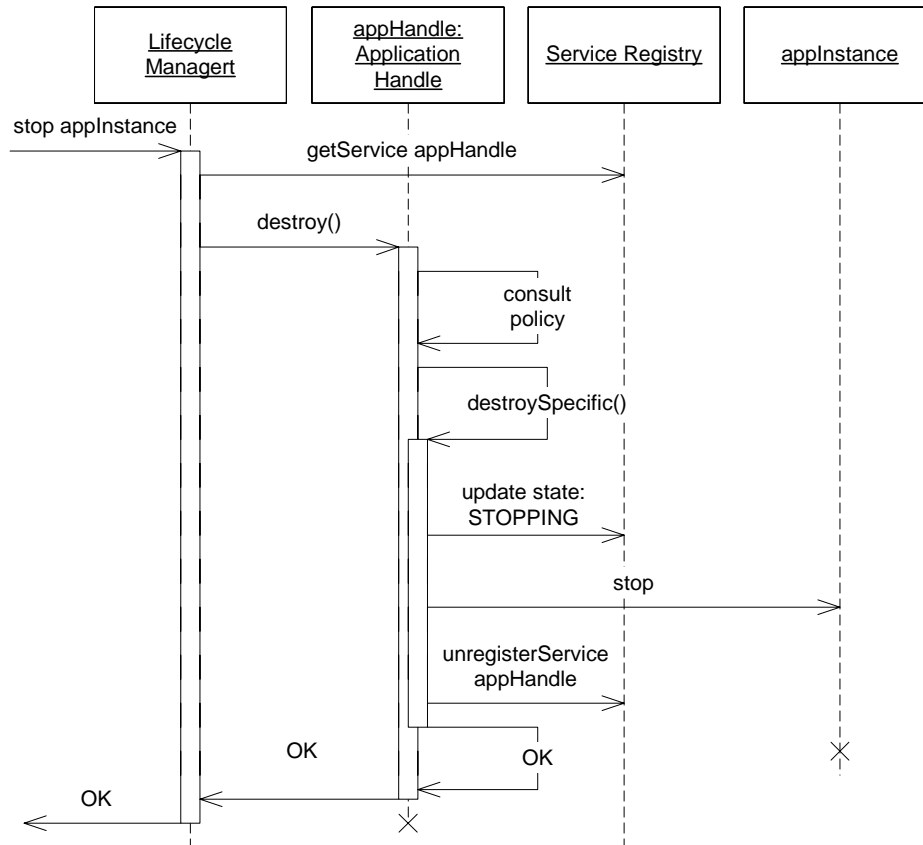
Firstly the appropriate application descriptor is fetched from the service registry on which the operation will be made. Its `launch()` method is called and startup arguments are passed to the application instance that will be launched. The application instance may not receive the startup arguments if its application model does not support it. The `launch()` method checks if the a new application instance can be launched, e.g. necessary rights are granted, the application is not lock and the application is not a singleton that already has an instance.

If the application can be launched then the `launchSpecific()` method will create and start a new application instance according to its application model. It will create a specific application handle and associate the newly created application instance to it. It will register the application handle with proper service properties. The value of `application.state` service property must be `RUNNING`. The call chain returns the service object to the application handle.



### 5.5.1.3 Destroying an Application Instance

To destroy an application the proper application handle has to be fetched from the service registry to call its `destroy()` method. It checks if the instance can be destroyed, e.g. necessary permissions are granted, it calls the `destroySpecific()` method to let its implementation to destroy the instance in application model specific way. Firstly it sets the `application.state` service property to `STOPPING` then stops the application instance. Finally it unregisters the application handle.



## 5.5.2 Interactions for the Meglets

The following diagrams illustrate a possible way of the operation of the application management framework when the managed applications are Meglets.

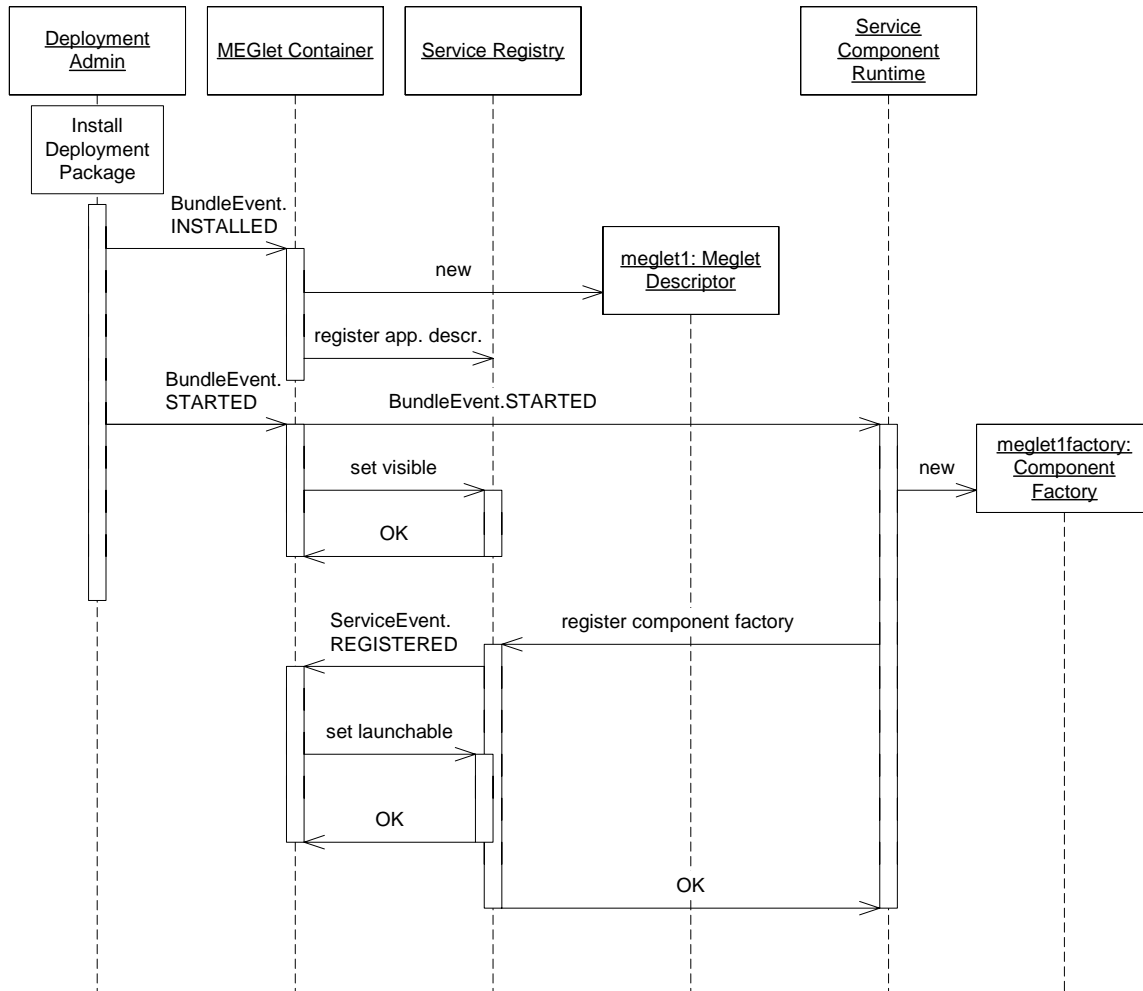
### 5.5.2.1 Installing a Meglet

The bundle that contains the Meglet is deployed as part of a deployment package. The Meglet container gets notified about bundle installation and if the bundle contains a Meglet the Meglet container will create a Meglet descriptor. It registers the Meglet descriptor of which service properties are coming from then meglets.xml file contained by the installed bundle.

The Meglet container and the SCR gets notified that the bundle has been started. The Meglet container will set the application.visible service property of the application descriptor to true. The SCR will create a new component factory service to track the dependencies of the Meglet. If the dependencies are fulfilled the SCR registers the component factory service.

The Meglet container gets notified about the registration of the component factory service and sets the application.launchable service property of the Meglet descriptor to true. Now the Meglet is ready for launching.

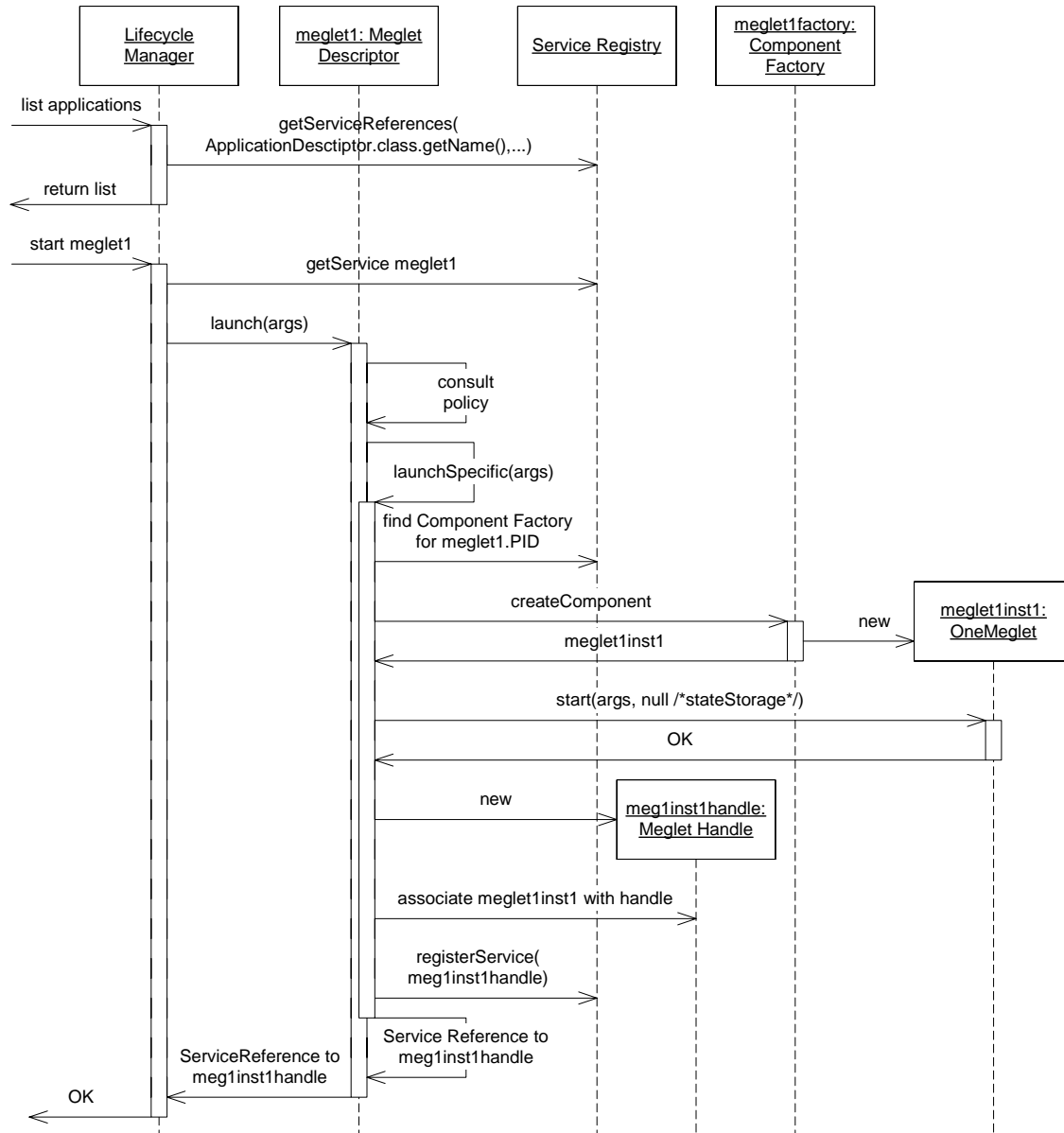




### 5.5.2.2 Launching a Meglet

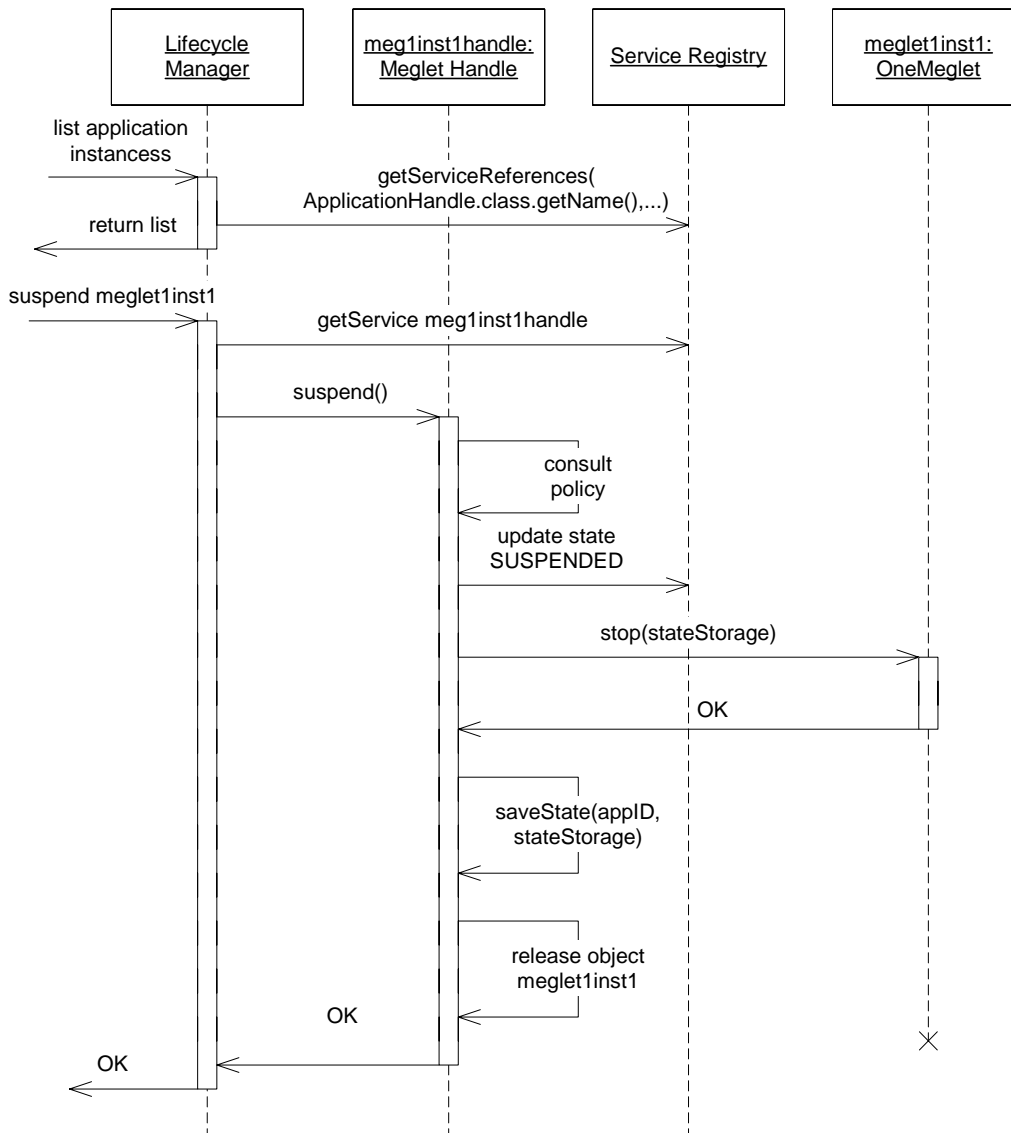
To launch a Meglet the proper Meglet descriptor has to be queried from the service registry to call its `launch()` method. After it checks if the Meglet can be launched, e.g. necessary rights are granted, the application is not lock and the application is not a singleton that already has an instance, it calls the `launchSpecific()` method. It will find the associated component factory which will create a new instance of the associated Meglet. The `launchSpecific()` method starts the newly created Meglet instance.

It creates a Meglet handle, and associates the Meglet instance to it. It registers the Meglet handle with the proper service properties, the value of the `application.state` service property must be `RUNNING`. Finally it returns with the service object to the registered Meglet handle.



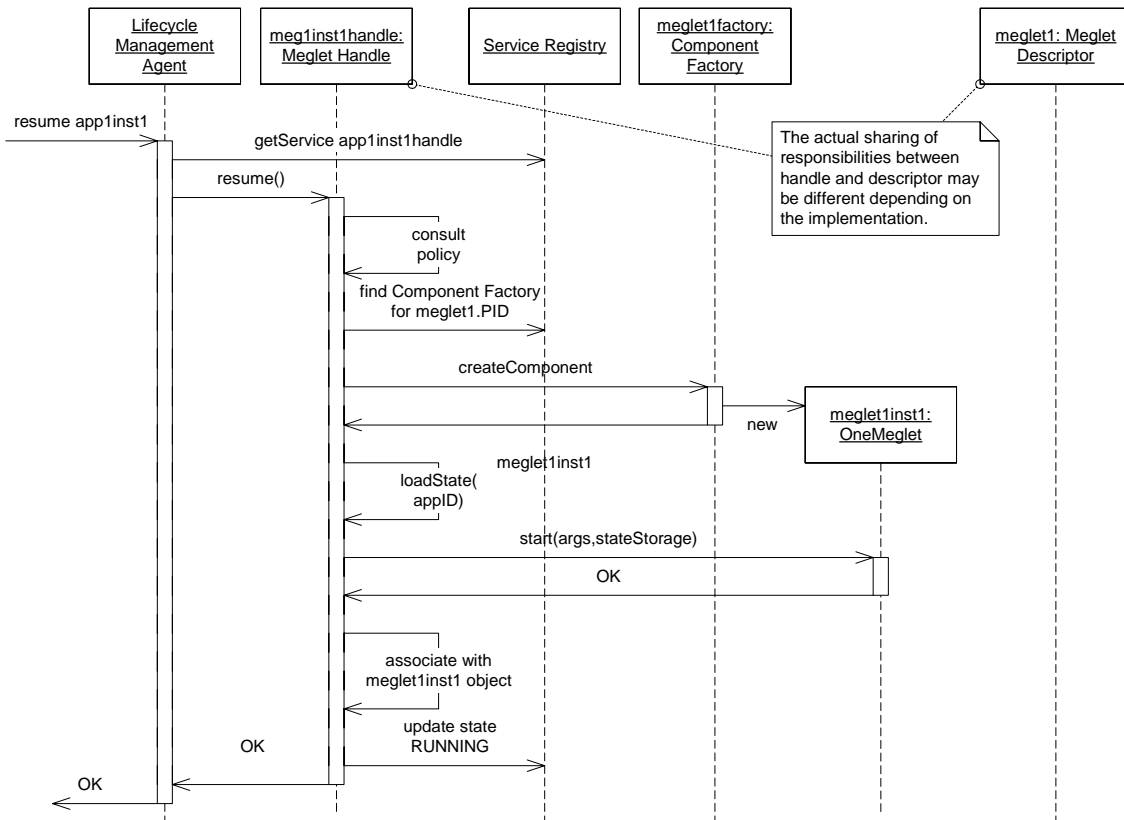
### 5.5.2.3 Suspending a Meglet Instance

The proper Meglet handle is fetched from the service registry and its `suspend()` method is called. It will check the execution policies and if it is allowed to suspend the Meglet instance it sets the Meglet handle's `application.state` service property to `SUSPENDED`. It calls the `stop()` method of the Meglet instance and passes a non-null output stream to it. The Meglet instance can save its inner state to the output stream. The Meglet handle stores the data written to the output stream into a persistent storage. Finally, the Meglet handle releases the associated Meglet instance.



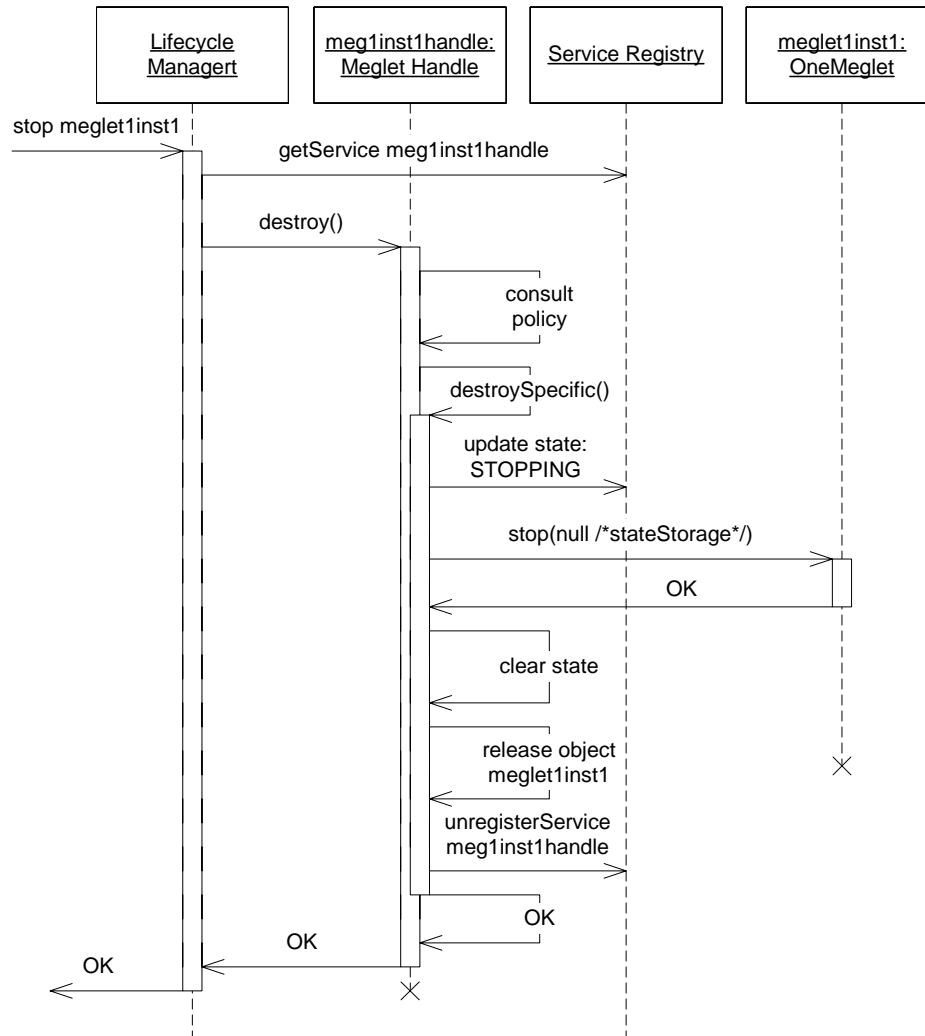
#### 5.5.2.4 Resuming a Meglet Instance

The Meglet handle has to be queried from the service registry to call its `resume()` method. It will check the execution policies, and if suspending is allowed it will fetch the component factory service associated to the Meglet to be resumed. The component factory service will create a new instance of the Meglet. The Meglet handle loads the data from the persistent storage. Then calls the Meglet instance's `start()` method and it passes an input stream to the Meglet instance to restore the state that was saved during suspending. Associates the started Meglet instance to the Meglet handle and sets the `application.state` service property of the Meglet handle to `RUNNING`.



### 5.5.2.5 Stopping a Meglet Instance

The proper Meglet handle is fetched to stop a Meglet instance. The `destroy()` method will be called, that checks if stopping is allowed. Then it calls the `destroySpecific()` method. It updates the `application.state` service property of the Meglet handle to `STOPPING` and stops the Meglet instance. It clears the lastly saved state from the persistent storage – if it has not happened yet – and clears the association between the Meglet handle and the Meglet instance. Finally it unregisters the Meglet handle.



## 5.6 Timer Event

Timer event is sent when the date and time it describes reached. This event must represent a valid date and time. The specified date and time is interpreted in the device's local time. The topic of the event is always "org/osgi/timer". The following properties are specific to timer event to describe the date and time it represents.

Name	Type	Notes
year	Integer	The year of the specified date.
month	Byte	The month of the year. Must be within range of 1 to 12. 1 is January.
day	Byte	The day of the month. Must be a valid day of the specified month of the specified year. Value of 1 is the first day of the month.

day_of_week	Byte	The day of the week. Must be within range of 0 to 6. 0 is Sunday.
hour	Byte	The hour of the day. Must be within the range of 0 to 23.
minute	Byte	The minute of the specified hour. Must be within the range of 0 to 59.
second	Byte	The second of the specified minute. Must be within the range of 0 to 59.
millisecond	Integer	The millisecond of the specified second. Must be within the range of 0 to 999.

It can be assumed that the framework makes reasonable effort to fire the specified timer event in time. However, firing the event is not guaranteed, implementations may drop events if it is necessary in order to preserve the stability and integrity of the device.

### 5.6.1 Timer Event Examples

The following examples are LDAP filters for the timer event to specify certain time in the device local time. The topic is always "org/osgi/timer".

- "( & ( hour = 12 ) ( minute = 0 ) ( second = 0 ) ( millisecond = 0 ) )" means noon in every day
- "( & ( day\_of\_week = 0 ) ( month = 1 ) ( minute = 0 ) ( second = 0 ) ( millisecond = 0 ) )" means every hour in every Sunday in January of every year
- "( & ( minute = 0 ) ( second = 0 ) ( millisecond = 0 ) )" means every whole hour
- "( & ( | ( minute = 0 ) ( minute = 30 ) ) ( second = 0 ) ( millisecond = 0 ) )" means every half an hour
- "( millisecond = 0 )" means every second
- "( & ( day\_of\_week = 6 ) ( day <= 7 ) ( hour = 12 ) ( minute = 0 ) ( second = 0 ) ( millisecond = 0 ) )" noon of first Saturday of every month

It is important to note that filters on timer event must be constructed carefully. If a filter contains a particular time unit then all of the smaller units should be added to the expression with "&" operator. If a unit is missing from a filter then it will match with any value. It may have the timer event be fired more frequently than intended and may cause system overload.

### 5.6.2 Sending Timer Events

Timer events should be sent only if someone is interested in them. It is not expected to send those timer events which nobody is interested in.

Even though the filter format enables specifying milliseconds, the implementation doesn't necessarily support the millisecond resolution when sending timer events. The timer event may be triggered at the first tick after the specified time. This is not intended to be real-time timer event, so the event may be sent later than the specified time, depending on the time resolution and the availability of resources. The implementation should make reasonable efforts to be accurate in sending timer events.

It is important to note that if a property is not included into the LDAP filter then it matches any value. Thus e.g. forgetting the "( millisecond = 0 )" clause from the filter potentially cause firing timer event in every single milliseconds. So specify the filter carefully.

# 6 API Specification

[Overview](#) [Package](#) [Class](#) [Tree](#) [Deprecated](#) [Index](#) [Help](#)
[PREV CLASS](#) [NEXT CLASS](#)

SUMMARY: [INNER](#) | [FIELD](#) | [CONSTR](#) | [METHOD](#)
[FRAMES](#) [NO FRAMES](#)

DETAIL: [FIELD](#) | [CONSTR](#) | [METHOD](#)

## 6.1 org.osgi.service.application

### Class ApplicationDescriptor

java.lang.Object

|

+--org.osgi.service.application.ApplicationDescriptor

Direct Known Subclasses:

[MegletDescriptor](#)

public abstract class **ApplicationDescriptor**

extends java.lang.Object

An OSGi service that represents an installed application and stores information about it. The application descriptor can be used for instance creation.




## Field Summary

static java.lang.String	<a href="#">APPLICATION_AUTOSTART</a> The property key for the autostart property of the application.
static java.lang.String	<a href="#">APPLICATION_ICON</a> The property key for the localized icon of the application.
static java.lang.String	<a href="#">APPLICATION_LAUNCHABLE</a> The property key for the launchable property of the application.
static java.lang.String	<a href="#">APPLICATION_LOCKED</a> The property key for the locked property of the application.
static java.lang.String	<a href="#">APPLICATION_NAME</a> The property key for the localized name of the application.
static java.lang.String	<a href="#">APPLICATION_PID</a> The property key for the unique identifier (PID) of the application.
static java.lang.String	<a href="#">APPLICATION_SINGLETON</a> The property key for the singleton property of the application.
static java.lang.String	<a href="#">APPLICATION_VENDOR</a> The property key for the name of the application vendor.
static java.lang.String	<a href="#">APPLICATION_VERSION</a> The property key for the version of the application.
static java.lang.String	<a href="#">APPLICATION_VISIBLE</a> The property key for the visibility property of the application.

## Constructor Summary

[ApplicationDescriptor](#) ()

## Method Summary

protected abstract org.osgi.framework.BundleContext	<a href="#">getBundleContext</a> () Retrieves the bundle context of the container to which the specialization of the application descriptor belongs
abstract java.lang.String	<a href="#">getPID</a> () Gets the identifier of the represented application.
java.util.Map	<a href="#">getProperties</a> (java.lang.String locale) Returns the properties of the application descriptor as key-value pairs.
boolean	<a href="#">isLocked</a> () Returns a boolean indicating whether the application is locked.
org.osgi.framework.ServiceReference 	<a href="#">launch</a> (java.util.Map arguments) Launches a new instance of an application.
protected abstract org.osgi.framework.ServiceReference 	<a href="#">launchSpecific</a> (java.util.Map arguments) Called by launch() to create and start a new instance in an application model specific way.
void 	<a href="#">lock</a> () Sets the lock state of the application.
org.osgi.framework.ServiceReference	<a href="#">schedule</a> (java.util.Map arguments, java.lang.String topic, java.lang.String eventFilter, boolean recurring) Schedules the application at a specified event.
void	<a href="#">unlock</a> () Unsets the lock state of the application.

### Methods inherited from class java.lang.Object

clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

## Field Detail

### 6.1.1 APPLICATION\_NAME

public static final java.lang.String **APPLICATION\_NAME**  
The property key for the localized name of the application.



### 6.1.2 APPLICATION\_ICON

```
public static final java.lang.String APPLICATION_ICON
```

The property key for the localized icon of the application.

### 6.1.3 APPLICATION\_PID

```
public static final java.lang.String APPLICATION_PID
```

The property key for the unique identifier (PID) of the application.

### 6.1.4 APPLICATION\_VERSION

```
public static final java.lang.String APPLICATION_VERSION
```

The property key for the version of the application.

### 6.1.5 APPLICATION\_VENDOR

```
public static final java.lang.String APPLICATION_VENDOR
```

The property key for the name of the application vendor.

### 6.1.6 APPLICATION\_SINGLETON

```
public static final java.lang.String APPLICATION_SINGLETON
```

The property key for the singleton property of the application.

### 6.1.7 APPLICATION\_AUTOSTART

```
public static final java.lang.String APPLICATION_AUTOSTART
```

The property key for the autostart property of the application.

### 6.1.8 APPLICATION\_VISIBLE

```
public static final java.lang.String APPLICATION_VISIBLE
```

The property key for the visibility property of the application.

### 6.1.9 APPLICATION\_LAUNCHABLE

```
public static final java.lang.String APPLICATION_LAUNCHABLE
```

The property key for the launchable property of the application.

### 6.1.10 APPLICATION\_LOCKED

```
public static final java.lang.String APPLICATION_LOCKED
```

The property key for the locked property of the application.

## Constructor Detail

### 6.1.11 ApplicationDescriptor

```
public ApplicationDescriptor()
```

## Method Detail

### 6.1.12 getPID

```
public abstract java.lang.String getPID()
```

Gets the identifier of the represented application. This identifier (PID) must be unique on the device, and must not change when the application is updated. This value is also available as a service property "service.pid" of this application descriptor.

**Throws:**

`IllegalStateException` - if the application descriptor is unregistered

---

### 6.1.13 `getProperties`

```
public java.util.Map getProperties(java.lang.String locale)
```

Returns the properties of the application descriptor as key-value pairs. The return value contains the locale aware and unaware properties as well. Some of the properties can be retrieved directly with methods in this interface.

**Parameters:**

`locale` - the locale string, it may be null, the value null means the default locale.

**Returns:**

service properties of this application descriptor service, according to the specified locale. If locale is null then the default locale's properties will be returned. (Since service properties are always exist it cannot return null.)

**Throws:**

`IllegalStateException` - if the application descriptor is unregistered

---

### 6.1.14 `launch`

```
public final org.osgi.framework.ServiceReference launch(java.util.Map arguments)  
throws SingletonException,  
       java.lang.Exception
```

Launches a new instance of an application. The args parameter specifies the startup parameters for the instance to be launched, it may be null.

The following steps are made:

- Check for the appropriate permission.
- Check the locking state of the application. If locked then return null otherwise continue.
- If the application is a singleton and already has a running instance then throw `SingletonException`.
- Calls the `launchSpecific()` method to create and start an application instance.
- Returns the `ServiceReference` returned by the `launchSpecific()`

The caller has to have `ApplicationAdminPermission(applicationPID, "launch")` in order to be able to perform this operation.

**Parameters:**

`arguments` - Arguments for the newly launched application, may be null

**Returns:**

the `ServiceReference` to registered `ApplicationHandle` which represents the newly launched application instance

**Throws:**

[SingletonException](#) - if the call attempts to launch a second instance of a singleton application

`java.lang.SecurityException` - if the caller doesn't have "launch" `ApplicationAdminPermission` for the application.

`java.lang.Exception` - if starting the application(s) failed

`IllegalStateException` - if the application descriptor is unregistered

---

### 6.1.15 launchSpecific

```
protected abstract org.osgi.framework.ServiceReference  
launchSpecific(java.util.Map arguments)
```

throws

```
java.lang.Exception
```

Called by `launch()` to create and start a new instance in an application model specific way. It also creates and registers the application handle to represent the newly created and started instance and registers it.

**Parameters:**

`arguments` - the startup parameters of the new application instance, may be null

**Returns:**

the service reference of the registered application model specific application handle for the newly created and started instance.

**Throws:**

`java.lang.Exception` - if any problem occurs.

---

### 6.1.16 schedule

```
public final org.osgi.framework.ServiceReference schedule(java.util.Map arguments,  
                                                         java.lang.String topic,
```

```
java.lang.String eventFilter,
```

```
                                                         boolean recurring)
```

```
                                                         throws java.io.IOException
```

Schedules the application at a specified event. Schedule information should not get lost even if the framework or the device restarts so it should be stored in a persistent storage. It has to register the returned service.

**Parameters:**

`arguments` - the startup arguments for the scheduled application, may be null

`topic` - specifies the topic of the triggering event, it may contain a trailing asterisk as wildcard, the empty string is treated as "", must not be null

`eventFilter` - specifies and LDAP filter to filter on the properties of the triggering event, may be null

`recurring` - if the recurring parameter is false then the application will be launched only once, when the event firstly occurs. If the parameter is true then scheduling will take place for every event occurrence; i.e. it is a recurring schedule

**Returns:**

the service reference of the registered scheduled application service

**Throws:**

`NullPointerException` - if the topic is null

`java.io.IOException` - may be thrown if writing the information about the scheduled application requires operation on the permanent storage and I/O problem occurred.

`java.lang.SecurityException` - if the caller doesn't have "schedule" `ApplicationAdminPermission` for the application.

`IllegalStateException` - if the application descriptor is unregistered

---

### 6.1.17 lock

```
public final void lock()
```

Sets the lock state of the application. If an application is locked then launching a new instance is not possible. It does not affect the already launched instances.

**Throws:**

`java.lang.SecurityException` - if the caller doesn't have "lock" `ApplicationAdminPermission` for the application.

`IllegalStateException` - if the application descriptor is unregistered

### 6.1.18 unlock

```
public final void unlock()
```

Unsets the lock state of the application.

**Throws:**

`java.lang.SecurityException` - if the caller doesn't have "lock" `ApplicationAdminPermission` for the application.

`IllegalStateException` - if the application descriptor is unregistered

### 6.1.19 isLocked

```
public final boolean isLocked()
```

Returns a boolean indicating whether the application is locked.

**Throws:**

`IllegalStateException` - if the application descriptor is unregistered

### 6.1.20 getBundleContext

```
protected abstract org.osgi.framework.BundleContext getBundleContext()
```

Retrieves the bundle context of the container to which the specialization of the application descriptor belongs

**Returns:**

the bundle context of the container

**Throws:**

`IllegalStateException` - if the application descriptor is unregistered

[Overview](#) [Package](#) [Class](#) [Tree](#) [Deprecated](#) [Index](#) [Help](#)

[PREV CLASS](#) [NEXT CLASS](#)

SUMMARY: [INNER](#) | [FIELD](#) | [CONSTR](#) | [METHOD](#)

[FRAMES](#) [NO FRAMES](#)

DETAIL: [FIELD](#) | [CONSTR](#) | [METHOD](#)

[Overview](#) [Package](#) [Class](#) [Tree](#) [Deprecated](#) [Index](#) [Help](#)

[PREV CLASS](#) [NEXT CLASS](#)

SUMMARY: [INNER](#) | [FIELD](#) | [CONSTR](#) | [METHOD](#)

[FRAMES](#) [NO FRAMES](#)

DETAIL: [FIELD](#) | [CONSTR](#) | [METHOD](#)

## 6.2 org.osgi.service.application

### Class ApplicationHandle

```
java.lang.Object
```

```
    |
    +--org.osgi.service.application.ApplicationHandle
```

**Direct Known Subclasses:**

[MegletHandle](#)

```
public abstract class ApplicationHandle
```

```
extends java.lang.Object
```

`ApplicationHandle` is an OSGi service interface which represents an instance of an application. It provides the functionality to query and manipulate the lifecycle state of the represented application instance. It defines constants for the lifecycle states.

## Field Summary


static int	<a href="#">RUNNING</a> The application instance is running.
static int	<a href="#">STOPPING</a> The application instance is stopping.

## Constructor Summary

[ApplicationHandle](#)()



## Method Summary

	void	<a href="#">destroy</a> () The application instance's lifecycle state can be influenced by this method.
	protected abstract void	<a href="#">destroySpecific</a> () Called by the <a href="#">destroy()</a> method to perform application model specific steps to stop and destroy an application instance safely.
	abstract org.osgi.framework.ServiceReference	<a href="#">getApplicationDescriptor</a> () Retrieves the application descriptor which represents the application of this application instance.
	abstract java.lang.String	<a href="#">getInstanceID</a> () Returns the unique identifier of this instance.
	abstract int	<a href="#">getState</a> () Get the state of the application instance.

### Methods inherited from class java.lang.Object

clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait



## Field Detail



### 6.2.1 STOPPING

public static final int **STOPPING**

The application instance is stopping. This is the state of a stopping application instance.

### 6.2.2 RUNNING

public static final int **RUNNING**

The application instance is running. This is the initial state of a newly created application instance.

## Constructor Detail

### 6.2.3 ApplicationHandle

```
public ApplicationHandle()
```

## Method Detail

### 6.2.4 getState

```
public abstract int getState()
```

Get the state of the application instance.

**Returns:**

the state of the application.

**Throws:**

`IllegalStateException` - if the application handle is unregistered

### 6.2.5 getInstanceID

```
public abstract java.lang.String getInstanceID()
```

Returns the unique identifier of this instance. This value is also available as a service property of this application handle's service.pid.

**Returns:**

the unique identifier of the instance

**Throws:**

`IllegalStateException` - if the application handle is unregistered

### 6.2.6 getApplicationDescriptor

```
public abstract org.osgi.framework.ServiceReference getApplicationDescriptor()
```

Retrieves the application descriptor which represents the application of this application instance. It should not be null.

**Returns:**

the service reference of the registered application descriptor which represents the application of this application instance, should not be null

**Throws:**

`IllegalStateException` - if the application handle is unregistered

### 6.2.7 destroy

```
public final void destroy()
```

throws `java.lang.Exception`

The application instance's lifecycle state can be influenced by this method. It lets the application instance to perform operations to stop the application safely, e.g. saving its state to a permanent storage.

The method must check if the lifecycle transition is valid; a STOPPING application cannot be stopped. If it is invalid then the method must exit. Otherwise the lifecycle state of the application instance must be set to STOPPING. Then the `destroySpecific()` method must be called to perform any application model specific steps for safe stopping of the represented application instance.

At the end the `ApplicationHandle` must be unregistered. This method should free all the resources related to this `ApplicationHandle`.

When this method is completed the application instance has already made its operations for safe stopping, the `ApplicationHandle` has been unregistered and its related resources has been freed. Further calls on this application should not be made because they may have unexpected results.

**Throws:**

java.lang.SecurityException - if the caller doesn't have "manipulate" ApplicationAdminPermission for the corresponding application.  
java.lang.Exception - is thrown if an exception or an error occurred during the method execution.  
IllegalStateException - if the application handle is unregistered

## 6.2.8 destroySpecific

protected abstract void **destroySpecific**()

throws java.lang.Exception

Called by the destroy() method to perform application model specific steps to stop and destroy an application instance safely.

### Throws:

java.lang.Exception - is thrown if an exception or an error occurred during the method execution.

[Overview](#) [Package](#) [Class](#) [Tree](#) [Deprecated](#) [Index](#) [Help](#)

[PREV CLASS](#) [NEXT CLASS](#)

SUMMARY: [INNER](#) | [FIELD](#) | [CONSTR](#) | [METHOD](#)

[FRAMES](#) [NO FRAMES](#)

DETAIL: [FIELD](#) | [CONSTR](#) | [METHOD](#)

[Overview](#) [Package](#) [Class](#) [Tree](#) [Deprecated](#) [Index](#) [Help](#)

[PREV CLASS](#) [NEXT CLASS](#)

SUMMARY: [INNER](#) | [FIELD](#) | [CONSTR](#) | [METHOD](#)

[FRAMES](#) [NO FRAMES](#)

DETAIL: [FIELD](#) | [CONSTR](#) | [METHOD](#)

## 6.3 org.osgi.service.application Interface ScheduledApplication

public interface **ScheduledApplication**

It is allowed to schedule an application based on a specific event. ScheduledApplication service keeps the scheduling information. When the specified event is fired a new instance must be launched. Note that launching operation may fail because e.g. the application is locked or the application is a singleton and already has an instance.

## Method Summary

org.osgi.framework.ServiceReference	<a href="#">getApplicationDescriptor</a> ( ) Retrieves the ApplicationDescriptor which represents the application and necessary for launching.
java.util.Map	<a href="#">getArguments</a> ( ) Queries the startup arguments specified when the application was scheduled.
java.lang.String	<a href="#">getEventFilter</a> ( ) Queries the event filter for the triggering event.
java.lang.String	<a href="#">getTopic</a> ( ) Queries the topic of the triggering event.
boolean	<a href="#">isRecurring</a> ( ) Queries if the scheduling is recurring.
void	<a href="#">remove</a> ( ) Cancels this schedule of the application.

## Method Detail

### 6.3.1 getTopic

```
public java.lang.String getTopic()
```

Queries the topic of the triggering event. The topic may contain a trailing asterisk as wildcard.

**Returns:**

the topic of the triggering event

**Throws:**

`IllegalStateException` - if the scheduled application service is unregistered

### 6.3.2 getEventFilter

```
public java.lang.String getEventFilter()
```

Queries the event filter for the triggering event.

**Returns:**

the event filter for triggering event

**Throws:**

`IllegalStateException` - if the scheduled application service is unregistered

### 6.3.3 isRecurring

```
public boolean isRecurring()
```

Queries if the scheduling is recurring.

**Returns:**

true if the scheduling is recurring, otherwise returns false

**Throws:**

`IllegalStateException` - if the scheduled application service is unregistered

### 6.3.4 getApplicationDescriptor

```
public org.osgi.framework.ServiceReference getApplicationDescriptor()
```

Retrieves the ApplicationDescriptor which represents the application and necessary for launching.

**Returns:**

the service reference to the application descriptor that represents the scheduled application

**Throws:**

`IllegalStateException` - if the scheduled application service is unregistered

### 6.3.5 getArguments

```
public java.util.Map getArguments()
```

Queries the startup arguments specified when the application was scheduled. The method returns a copy of the arguments, it is not possible to modify the arguments after scheduling.

**Returns:**

the startup arguments of the scheduled application. It may be null if null argument was specified.

**Throws:**

`IllegalStateException` - if the scheduled application service is unregistered

### 6.3.6 remove

```
public void remove()
```

Cancels this schedule of the application.

**Throws:**



java.lang.SecurityException - if the caller doesn't have "schedule" ApplicationAdminPermission for the scheduled application.

IllegalStateException - if the scheduled application service is unregistered

## Overview Package Class Tree Deprecated Index Help

PREV CLASS NEXT CLASS

SUMMARY: INNER | FIELD | CONSTR | [METHOD](#)

[FRAMES](#) [NO FRAMES](#)

DETAIL: FIELD | CONSTR | [METHOD](#)

## Overview Package Class Tree Deprecated Index Help

PREV CLASS NEXT CLASS

SUMMARY: INNER | FIELD | [CONSTR](#) | [METHOD](#)

[FRAMES](#) [NO FRAMES](#)

DETAIL: FIELD | [CONSTR](#) | [METHOD](#)

## 6.4 org.osgi.service.application

### Class SingletonException

java.lang.Object

|  
+-- java.lang.Throwable

|  
+-- java.lang.Exception

|  
+-- org.osgi.service.application.SingletonException

#### All Implemented Interfaces:

java.io.Serializable

public class **SingletonException**

extends java.lang.Exception

An exception that is raised when a singleton application (i.e. it can have zero or one instance at a time) have one instance and launching a new instance is attempted.

#### See Also:

[Serialized Form](#)

## Constructor Summary

[SingletonException](#) ( )

Parameterless constructor.

[SingletonException](#) (java.lang.String message)

Constructor to create the exception with the specified message.

### Methods inherited from class java.lang.Throwable

fillInStackTrace, getLocalizedMessage, getMessage, printStackTrace,  
printStackTrace, printStackTrace, toString

### Methods inherited from class java.lang.Object

clone, equals, finalize, getClass, hashCode, notify, notifyAll, wait, wait, wait

## Constructor Detail

### 6.4.1 SingletonException

```
public SingletonException()
    Parameterless constructor.
```

### 6.4.2 SingletonException

```
public SingletonException(java.lang.String message)
    Constructor to create the exception with the specified message.
Parameters:
    message - the message for the exception
```

[Overview](#) [Package](#) [Class](#) [Tree](#) [Deprecated](#) [Index](#) [Help](#)

PREV CLASS NEXT CLASS

SUMMARY: INNER | FIELD | [CONSTR](#) | [METHOD](#)

[FRAMES](#) [NO FRAMES](#)

DETAIL: FIELD | [CONSTR](#) | METHOD

[Overview](#) [Package](#) [Class](#) [Tree](#) [Deprecated](#) [Index](#) [Help](#)

PREV CLASS [NEXT CLASS](#)

SUMMARY: INNER | [FIELD](#) | [CONSTR](#) | [METHOD](#)

[FRAMES](#) [NO FRAMES](#)

DETAIL: [FIELD](#) | [CONSTR](#) | METHOD

## 6.5 org.osgi.service.application Class ApplicationAdminPermission

```
java.lang.Object
|
+-- java.security.Permission
|   |
|   +-- java.security.BasicPermission
|       |
|       +-- org.osgi.service.application.ApplicationAdminPermission
```

### All Implemented Interfaces:

java.security.Guard, java.io.Serializable

```
public class ApplicationAdminPermission
    extends java.security.BasicPermission
```

This class implements permissions for manipulating applications and application instances. ApplicationAdminPermission can be targeted to a particular application (using its PID as an identifier) or to all applications (when no PID is specified).

ApplicationAdminPermission may be granted for different actions: launch, schedule, manipulate, schedule and lock. There are some implication rules between these:

- schedule => launch
- manipulate => launch

### See Also:

[Serialized Form](#)

## Field Summary

static java.lang.String	<a href="#">LAUNCH</a> Allows launching all the other applications or those with the specified unique identifier.
static java.lang.String	<a href="#">LOCK</a> Allows setting/unsetting the locking state of other applications or those with the specified unique identifier.
static java.lang.String	<a href="#">MANIPULATE</a> Allows manipulating the lifecycle state of instances of all the other applications or the instances of applications where application has the specified unique identifier.
static java.lang.String	<a href="#">SCHEDULE</a> Allows scheduling all the other applications or those with the specified unique identifier.

## Constructor Summary

[ApplicationAdminPermission](#)(java.lang.String actions)

Constructs an ApplicationAdminPermission.

[ApplicationAdminPermission](#)(java.lang.String pid, java.lang.String actions)

Constructs an ApplicationAdminPermission.

### Methods inherited from class java.security.BasicPermission

equals, getActions, hashCode, implies, newPermissionCollection

### Methods inherited from class java.security.Permission

checkGuard, getName, toString

### Methods inherited from class java.lang.Object

clone, finalize, getClass, notify, notifyAll, wait, wait, wait

## Field Detail

### 6.5.1 LAUNCH

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

Allows launching all the other applications or those with the specified unique identifier.

### 6.5.2 SCHEDULE

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

Allows scheduling all the other applications or those with the specified unique identifier. Permission to schedule an application implies the permission launch on the same application.

### 6.5.3 MANIPULATE

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

Allows manipulating the lifecycle state of instances of all the other applications or the instances of applications where application has the specified unique identifier. Permission to manipulate instances of an application implies the permission launch on the same application.

## 6.5.4 LOCK

```
public static final java.lang.String LOCK
```

Allows setting/unsetting the locking state of other applications or those with the specified unique identifier.

## Constructor Detail

### 6.5.5 ApplicationAdminPermission

```
public ApplicationAdminPermission(java.lang.String actions)
```

Constructs an ApplicationAdminPermission. It is equal to the ApplicationAdminPermission(null, actions) call.

#### Parameters:

**actions** - - comma-separated list of the desired actions granted on the applications. It must not be null. The order of the actions in the list is not significant.

### 6.5.6 ApplicationAdminPermission

```
public ApplicationAdminPermission(java.lang.String pid,
                                   java.lang.String actions)
```

Constructs an ApplicationAdminPermission. The name is the unique identifier of the target application. If the name is null then the constructed permission is granted for all targets.

#### Parameters:

**pid** - - unique identifier of the application, it may be null. The value null indicates "all application".

**actions** - - comma-separated list of the desired actions granted on the applications. It must not be null. The order of the actions in the list is not significant.

#### Throws:

**NullPointerException** - is thrown if the actions parameter is null

#### Overview Package Class Tree Deprecated Index Help

PREV CLASS [NEXT CLASS](#)

SUMMARY: INNER | [FIELD](#) | [CONSTR](#) | [METHOD](#)

[FRAMES](#) [NO FRAMES](#)

DETAIL: [FIELD](#) | [CONSTR](#) | [METHOD](#)

#### Overview Package Class Tree Deprecated Index Help

PREV CLASS [NEXT CLASS](#)

SUMMARY: INNER | [FIELD](#) | [CONSTR](#) | [METHOD](#)

[FRAMES](#) [NO FRAMES](#)

DETAIL: [FIELD](#) | [CONSTR](#) | [METHOD](#)

## 6.6 org.osgi.service.application.meglet

### Class MegletDescriptor

```
java.lang.Object
```

```
|
+--org.osgi.service.application.ApplicationDescriptor
```

```
|
+--org.osgi.service.application.meglet.MegletDescriptor
```

```
public final class MegletDescriptor
```

```
extends ApplicationDescriptor
```

Specialization of the application descriptor. Represents a Meglet and provides generic methods inherited from the application descriptor. It is a service.

#### Fields inherited from class org.osgi.service.application.[ApplicationDescriptor](#)

[APPLICATION\\_AUTOSTART](#), [APPLICATION\\_ICON](#), [APPLICATION\\_LAUNCHABLE](#),  
[APPLICATION\\_LOCKED](#), [APPLICATION\\_NAME](#), [APPLICATION\\_PID](#), [APPLICATION\\_SINGLETON](#),  
[APPLICATION\\_VENDOR](#), [APPLICATION\\_VERSION](#), [APPLICATION\\_VISIBLE](#)

## Constructor Summary

[MegletDescriptor](#) ( )

## Method Summary

protected org.osgi.framework.BundleContext	<a href="#">getBundleContext</a> ( ) Retrieves the bundle context of the container to which the specialization of the application descriptor belongs
java.lang.String	<a href="#">getPID</a> ( ) Returns the unique identifier of the represented Meglet.
protected org.osgi.framework.ServiceReference	<a href="#">launchSpecific</a> (java.util.Map arguments) Called by launch() to create and start a new instance in an application model specific way.

#### Methods inherited from class org.osgi.service.application.[ApplicationDescriptor](#)

[getProperties](#), [isLocked](#), [launch](#), [lock](#), [schedule](#), [unlock](#)

#### Methods inherited from class java.lang.Object

clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

## Constructor Detail

### 6.6.1 MegletDescriptor

public [MegletDescriptor](#) ( )

## Method Detail

### 6.6.2 getPID

public java.lang.String [getPID](#) ( )

Returns the unique identifier of the represented Meglet.

**Overrides:**

[getPID](#) in class [ApplicationDescriptor](#)

**Returns:**

the unique identifier of the represented Meglet

**Throws:**

`IllegalStateException` - if the Meglet descriptor is unregistered

### 6.6.3 launchSpecific

protected `org.osgi.framework.ServiceReference`

**launchSpecific**(`java.util.Map` arguments)

throws `java.lang.Exception`

Called by `launch()` to create and start a new instance in an application model specific way. It also creates and registers the application handle to represent the newly created and started instance.

**Overrides:**

[launchSpecific](#) in class [ApplicationDescriptor](#)

**Parameters:**

arguments - the startup parameters of the new application instance, may be null

**Returns:**

the service reference to the application model specific application handle for the newly created and started instance.

**Throws:**

`java.lang.Exception` - if any problem occurs.

### 6.6.4 getBundleContext

protected `org.osgi.framework.BundleContext` **getBundleContext**()

Retrieves the bundle context of the container to which the specialization of the application descriptor belongs

**Overrides:**

[getBundleContext](#) in class [ApplicationDescriptor](#)

**Returns:**

the bundle context of the container

**Throws:**

`IllegalStateException` - if the Meglet descriptor is unregistered

[Overview](#) [Package](#) **[Class](#)** [Tree](#) [Deprecated](#) [Index](#) [Help](#)

PREV CLASS [NEXT CLASS](#)

SUMMARY: INNER | [FIELD](#) | [CONSTR](#) | [METHOD](#)

[FRAMES](#) [NO FRAMES](#)

DETAIL: [FIELD](#) | [CONSTR](#) | [METHOD](#)

[Overview](#) [Package](#) **[Class](#)** [Tree](#) [Deprecated](#) [Index](#) [Help](#)

[PREV CLASS](#) [NEXT CLASS](#)

SUMMARY: INNER | [FIELD](#) | [CONSTR](#) | [METHOD](#)

[FRAMES](#) [NO FRAMES](#)

DETAIL: [FIELD](#) | [CONSTR](#) | [METHOD](#)

## 6.7 org.osgi.service.application.meglet

### Class MegletHandle

`java.lang.Object`

```

|
+--org.osgi.service.application.ApplicationHandle
    |
    +--org.osgi.service.application.meglet.MegletHandle

```

public final class **MegletHandle**

extends [ApplicationHandle](#)

This service represents a Meglet instance. It is a specialization of the application handle and provides features specific to the Meglet model.

## Field Summary

static int	<a href="#"><b>SUSPENDED</b></a> The Meglet instance is suspended.
------------	---

Fields inherited from class `org.osgi.service.application.ApplicationHandle`

[RUNNING](#), [STOPPING](#)

## Constructor Summary

[MegletHandle](#)( )

## Method Summary

protected void	<a href="#"><b>destroySpecific</b></a> ( ) Destroys a Meglet according to the Meglet model.
<code>org.osgi.framework.ServiceReference</code>	<a href="#"><b>getApplicationDescriptor</b></a> ( ) Returns service reference to the application descriptor of the Meglet to which this Meglet instance belongs to.
<code>java.lang.String</code>	<a href="#"><b>getInstanceID</b></a> ( ) Returns the instance id of the Meglet instance.
int	<a href="#"><b>getState</b></a> ( ) Returns the state of the Meglet instance specific to the Meglet model.
void	<a href="#"><b>resume</b></a> ( ) Resumes the Meglet instance.
void	<a href="#"><b>suspend</b></a> ( ) Suspends the Melet instance.

Methods inherited from class `org.osgi.service.application.ApplicationHandle`

[destroy](#)

Methods inherited from class `java.lang.Object`

`clone`, `equals`, `finalize`, `getClass`, `hashCode`, `notify`, `notifyAll`, `toString`, `wait`, `wait`, `wait`

## Field Detail

### 6.7.1 SUSPENDED

```
public static final int SUSPENDED
```

The Meglet instance is suspended.

## Constructor Detail

### 6.7.2 MegletHandle

```
public MegletHandle()
```

## Method Detail

### 6.7.3 getState

```
public int getState()
```

Returns the state of the Meglet instance specific to the Meglet model.

**Overrides:**

[getState](#) in class [ApplicationHandle](#)

**Returns:**

the state of the Meglet instance

**Throws:**

`IllegalStateException` - if the Meglet handle is unregistered

### 6.7.4 getInstanceID

```
public java.lang.String getInstanceID()
```

Returns the instance id of the Meglet instance. Must be unique on the device.

**Overrides:**

[getInstanceID](#) in class [ApplicationHandle](#)

**Returns:**

the instance id of the Meglet instance

**Throws:**

`IllegalStateException` - if the Meglet handle is unregistered

### 6.7.5 getApplicationDescriptor

```
public org.osgi.framework.ServiceReference getApplicationDescriptor()
```

Returns service reference to the application descriptor of the Meglet to which this Meglet instance belongs to.

**Overrides:**

[getApplicationDescriptor](#) in class [ApplicationHandle](#)

**Returns:**

the application descriptor of the Meglet to which this Meglet instance belongs to

**Throws:**

`IllegalStateException` - if the Meglet handle is unregistered

### 6.7.6 destroySpecific

```
protected void destroySpecific()
```

throws `java.lang.Exception`

Destroys a Meglet according to the Meglet model. It calls the associated Meglet instance's `stop()` method with null parameter.

**Overrides:**

[destroySpecific](#) in class [ApplicationHandle](#)

Following copied from class: `org.osgi.service.application.ApplicationHandle`

**Throws:**



`java.lang.Exception` - is thrown if an exception or an error occurred during the method execution.

### 6.7.7 suspend

```
public void suspend()
```

```
    throws java.lang.Exception
```

Suspends the Meglet instance. It calls the associated Meglet instance's `stop()` method and passes a non-null output stream as a parameter. It must preserve the contents of the output stream written by the Meglet instance even across device restarts. The same content must be provided to a resuming Meglet instance.

#### Throws:

`java.lang.SecurityException` - if the caller doesn't have "manipulate" `ApplicationAdminPermission` for the corresponding application.

`IllegalStateException` - if the Meglet handle is unregistered

### 6.7.8 resume

```
public void resume()
```

```
    throws java.lang.Exception
```

Resumes the Meglet instance. It calls the associated Meglet instance's `start()` method and passes a non-null input stream as a parameter. It must have the same contents that was saved by the suspending Meglet instance. The same startup arguments must also be passed to the resuming Meglet instance that was passed to the first starting instance.

#### Throws:

`java.lang.SecurityException` - if the caller doesn't have "manipulate" `ApplicationAdminPermission` for the corresponding application.

`IllegalStateException` - if the Meglet handle is unregistered

[Overview](#) [Package](#) [Class](#) [Tree](#) [Deprecated](#) [Index](#) [Help](#)

[PREV CLASS](#) [NEXT CLASS](#)

[FRAMES](#)

[NO FRAMES](#)

SUMMARY: [INNER](#) | [FIELD](#) | [CONSTR](#) | [METHOD](#)

DETAIL: [FIELD](#) | [CONSTR](#) | [METHOD](#)

[Overview](#) [Package](#) [Class](#) [Tree](#) [Deprecated](#) [Index](#) [Help](#)

[PREV CLASS](#) [NEXT CLASS](#)

[FRAMES](#) [NO FRAMES](#)

SUMMARY: [INNER](#) | [FIELD](#) | [CONSTR](#) | [METHOD](#)

DETAIL: [FIELD](#) | [CONSTR](#) | [METHOD](#)

## 6.8 org.osgi.meglet

### Class Meglet

`org.osgi.meglet.Meglet`

```
public abstract class Meglet
```

Abstract base class for Meglets. Developers are expected to extend this class to implement the behavior and event handling for their applications.

This class provides many convenience methods which cannot be overridden.

## Constructor Summary

[Meglet](#) ( )

## Method Summary

protected void	<a href="#">activate</a> (org.osgi.service.component.ComponentContext context) Called by the framework when this component is activated.
protected void	<a href="#">deactivate</a> (org.osgi.service.component.ComponentContext context) Called by the framework when this component is deactivated.
protected org.osgi.service.component.ComponentContext	<a href="#">getComponentContext</a> () Returns the component context assigned to the Meglet instance.
protected org.osgi.service.event.EventAdmin	<a href="#">getEventAdmin</a> () Retrieves the event channel service if available.
protected org.osgi.service.log.LogService	<a href="#">getLogService</a> () Retrieves the log service if available.
protected java.util.Dictionary	<a href="#">getProperties</a> () Retrieves the service properties of the application descriptor of this application instance.
void	<a href="#">handleEvent</a> (org.osgi.service.event.Event event) Receives events.
protected void	<a href="#">registerForEvents</a> (java.lang.String topic, java.lang.String eventFilter) Registers the Meglet to listen to events with the specified topic and eventFilter.
protected void	<a href="#">requestStop</a> () The application can ask the environment for stopping.
protected void	<a href="#">requestSuspend</a> () The application can ask the environment for suspending.
void	<a href="#">start</a> (java.util.Map args, java.io.InputStream stateStorage) This method is called by the framework to start this application instance.
void	<a href="#">stop</a> (java.io.OutputStream stateStorage) This method is called by the framework to stop this application instance.
protected void	<a href="#">unregisterForEvents</a> (java.lang.String topic, java.lang.String eventFilter) Unregisters the Meglet not to listen to the specified event.

## Constructor Detail

### 6.8.1 Meglet

```
public Meglet()
```

## Method Detail

### 6.8.2 getComponentContext

```
protected final org.osgi.service.component.ComponentContext getComponentContext()
```

Returns the component context assigned to the Meglet instance.

**Returns:**

the component context

### 6.8.3 start

```
public void start(java.util.Map args,  
                 java.io.InputStream stateStorage)  
    throws java.lang.Exception
```

This method is called by the framework to start this application instance. Must not be called directly.

**Parameters:**

*args* - The startup parameters for the application instance as key-value pairs.

*stateStorage* - the input stream from where the application can load its saved inner state. It is null if a brand new instance of the Meglet is started, so no previously saved state exist to be loaded. It is not mandated to close the stream.

**Throws:**

*java.lang.Exception* - if any error occurs

### 6.8.4 stop

```
public void stop(java.io.OutputStream stateStorage)  
    throws java.lang.Exception
```

This method is called by the framework to stop this application instance. Must not be called directly. The method must free all the used resources.

**Parameters:**

*stateStorage* - the output stream to where the application can save its inner state. If it is null then the Meglet instance is stopped permanently so saving its inner state is not expected. It is not mandated to close the stream.

**Throws:**

*java.lang.Exception* - if any error occurs

### 6.8.5 handleEvent

```
public void handleEvent(org.osgi.service.event.Event event)
```

Receives events. By default it does nothing so the developer has to override it to deal with events.

This method must not be called directly.

**Parameters:**

*event* - the event to be handled, not null

### 6.8.6 requestSuspend

```
protected final void requestSuspend()
```

The application can ask the environment for suspending. The environment must call the stop() method at some point with non-null stateStorage parameter.

### 6.8.7 requestStop

```
protected final void requestStop()
```

The application can ask the environment for stopping. The environment must call the stop() method at some point with null stateStorage parameter.

---

### 6.8.8 getEventAdmin

```
protected final org.osgi.service.event.EventAdmin getEventAdmin()
```

Retrieves the event channel service if available. Otherwise returns null.

**Returns:**

the event channel service, or null if it is not available

---

### 6.8.9 getLogService

```
protected final org.osgi.service.log.LogService getLogService()
```

Retrieves the log service if available. Otherwise returns null.

**Returns:**

the retrieved log service, or null if it is not available

---

### 6.8.10 registerForEvents

```
protected final void registerForEvents(java.lang.String topic,  
                                       java.lang.String eventFilter)
```

Registers the Meglet to listen to events with the specified topic and eventFilter.

**Parameters:**

topic - the topic of the event the subscriber is interested to, it may contain a trailing asterisk as wildcard, the empty string is treated as "\*", must not be null

eventFilter - LDAP filter for the event as defined in Generic Event Mechanism RFC, may be null

**Throws:**

NullPointerException - if topic is null

---

### 6.8.11 unregisterForEvents

```
protected final void unregisterForEvents(java.lang.String topic,  
                                          java.lang.String eventFilter)
```

Unregisters the Meglet not to listen to the specified event. It works only if it was registered by the registerForEvents() method to listen to event with exactly the same topic and eventFilter.

**Parameters:**

topic - the topic of the event the subscriber is interested to, it may contain a trailing asterisk as wildcard, the empty string is treated as "\*", must not be null

eventFilter - LDAP filter for the event as defined in Generic Event Mechanism RFC, may be null

**Throws:**

NullPointerException - if topic is null

---

### 6.8.12 getProperties

```
protected final java.util.Dictionary getProperties()
```

Retrieves the service properties of the application descriptor of this application instance.

**Returns:**

the properties of the application descriptor

---

### 6.8.13 activate

```
protected final void activate(org.osgi.service.component.ComponentContext context)
```

Called by the framework when this component is activated. Called before the start().

**Parameters:**

context - the component context received from the framework

### 6.8.14 deactivate

protected final void

**deactivate**(org.osgi.service.component.ComponentContext context)

Called by the framework when this component is deactivated. Called after the stop().

**Parameters:**

context - the component context received from the framework

[Overview](#) [Package](#) **[Class](#)** [Tree](#) [Deprecated](#) [Index](#) [Help](#)

[PREV CLASS](#) [NEXT CLASS](#)

SUMMARY: [INNER](#) | [FIELD](#) | [CONSTR](#) | [METHOD](#)

[FRAMES](#) [NO FRAMES](#)

DETAIL: [FIELD](#) | [CONSTR](#) | [METHOD](#)

## 7 Considered Alternatives

At the Newark meeting it was suggested that application descriptor might take over then entire function of ApplicationAdmin. While I was editing the document to reflect the decisions made at the meeting I realized that it was still needed as a target for scheduling applications. The name might better be changed to ApplicationScheduler since this is the only remaining function. (Added by Rick DeNatale.)

Then the scheduling functionality was also moved to the application descriptor and ScheduledApplication became a service. The rationale behind moving ApplicationAdmin's functionality to the application descriptor is that an extra interface could be eliminated. All of the methods of application admin had the application descriptor parameter to define on which application to perform the selected operation.

The application descriptor and ApplicationHandle became an abstract class instead of being an interface. They have final methods which are intended to be implemented by the implementer of the application management framework. In these final methods the generic policies of the framework can be ensured in a way that cannot be circumvented by specialized implementations provided to support various application models, like the Meglet model. They contain some protected methods which cannot be invoked from outside of the object but can be overridden by the implementer of the specialized classes to provide a behavior specific to the supported application model.

---

## 8 Security Considerations

---

*Description of all known vulnerabilities this may either introduce or address as well as scenarios of how the weaknesses could be circumvented.*

When an application (A) launches another application (B), application B will be constrained by application A's permission. In other words, launching an application is not surrounded with a `doPrivileged()` call of in the application management framework. This mechanism is used to make sure that application A cannot obtain higher privileges unnoticed by launching a more trusted application. If B wouldn't be constrained by A's permissions, then it would be possible for A to use B – by specifying the appropriate startup parameters – to perform some action for which A has no permission. If B wants to perform some action independently from its launcher, then it should surround that action with a `doPrivileged()` call. To ensure that applications launched by the user can execute properly, `AllPermission` should be granted to Desktop Manager. When launch of an application is triggered by some event for which it was scheduled, the application management framework launches the application in `doPrivileged()` call, with all permission. As startup parameters can be specified for the scheduling, a malicious application with schedule privilege can perform the attack described above. This implies that the right to schedule an application is much stronger and therefore, it should be granted to applications carefully.

---

## 9 Document Support

---

---

### 9.1 References

- [1]. Bradner, S., Key words for use in RFCs to Indicate Requirement Levels, RFC2119, March 1997.
- [2]. RFC 80 – Declarative Services  
<http://membercvs.osgi.org/rfcs/rfc0080/>
- [3]. RFC 97 – Generic Event Mechanism  
<http://membercvs.osgi.org/rfcs/rfc0097/>
- [4]. RFP 52 – Application Model Requirements  
[http://membercvs.osgi.org/rfps/rfp-0052\\_MEG\\_WS\\_Application\\_Model.doc](http://membercvs.osgi.org/rfps/rfp-0052_MEG_WS_Application_Model.doc)
- [5].
- [6].
- [7].
- [8].

[9].

<http://membercvs.osgi.org/rfcs/rfc0097/>

---

## 9.2 Author's Address

Name	Venkat Amirisetty
Company	Motorola, Inc
Address	805, E. Middlefield Road, Mountain View, CA 94043, USA
Voice	+1-650-318-3362
e-mail	vamirisetty@motorola.com

---

## 9.3 Acronyms and Abbreviations

---

## 9.4 End of Document