RFC 16 - Meta data for properties

Members Only, **RC4**
cpeg-rfc_16_meta-1_00G

19 Pages

Abstract

Meta data is data about the types of other data. This document proposes an API for representing meta data in the OSGi. The API is defined, explained and related to other approaches.

1 Contents

2 Document Information

2.1 Status

This document proposes a meta data standard for properties to be reviewed by the Core Platform Expert Group. The document is a proposal for the release 2 umbrella release. Distribution is OSGi members limited.

2.2 Acknowledgement

This document was reviewed, discussed and contributed by David Bowen (SUN), Jan Luehe (SUN), Maria Ivanova (ProSyst), Petri Niska (Nokia) and K. Hellden (Gatespace).

2.3 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 [11].

`Source code is shown in this typeface.`

2.4 Revision History

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

Revision	Date	Comments
First draft.	24 January 2001	First draft. Peter Kriens (lifted from RFC 7 draft E)
Draft B	20 February 2001	After review of CM subgroup
Draft C	6 march 2001	After Dallas meeting comments were added (Peter Kriens)
Draft E	12 April 2001	After San Francisco meeting. Changed default value to String[] and some clarifications

3 Introduction

Properties (a set of key-value pairs) have become an important part of today's programming paradigm. They are an escape of the rigid type safety requirements of modern languages. Type safety works well for normal software development when many programmers must work together on large systems but lack the flexibility needed to receive structured information from the outside world. Properties have several characteristics that make them suitable when information needs to be communicated between different parties that "speak" different languages. They are simple, resilient to change and allow the receiver to adapt.

This RFC was driven by [4] (RFC-7) which uses properties for Configuration Management (CM). During the development of that RFC it became clear that these properties needed to be described in a computer readable form with the so called meta data. This meta data could then be used to automatically create user interfaces for management systems or be translated into management information specifications (CIM, SNMP).

However, the same properties types used in the CM proposal is also used in the framework. Other proposals in progress also have made subsets and there is a tendency to standardize on the property types defined by the framework. Therefore it was decided to move this problem to a separate RFC so it could be referenced from several other proposals.

This RFC is associated with RFP 16.

4 Motivation and rational

4.1 Requirements on meta typing

The purpose of this RFC is to define a number of interfaces that describe the possible values of a set of properties. Bundles are expected to include this information in the Jar file and make it accessible.

4.1.1 Basic requirements

1 A conceptual model for the organization of properties. E.g. are properties scoped by a class or is a class a collection of globally defined properties?

2 Aligned with appropriate standard bodies

3 A bundle must be able to use the interfaces to allow other bundles safe access to the meta type information.

4 A management system must be able to access the information from the Jar file without requiring the bundle to run.

5 Will not require more than the minimal profile as requested by RFP 0006 Request For Proposal for Execution Profile.

6 Minimal overhead in size for a bundle

7 Localization must be parametrized and not assuming the default location of the environment. This allows servlets to serve information in the language selected in the browser.

4.1.2 Requirements on information in meta data

1 Name of the property.

2 Required or optional status of a property.

3 Cardinality (once, 0..n)

4 Localized label for the property

5 A human readable description of the property semantics and constraints

6 Java related type, e.g. String, Integer etc.

7 Enumerated values with localized labels

8 Validation support

4.2 The types of the properties

The OSGi framework is already using the LDAP filter syntax and the usage of the properties in this RFC and the framework registry already are very close to the LDAP attribute model. Therefore, the meta type interface classes are closely modelled according to LDAP. For further reading on ASN.1 defined properties and X.500 object classes and attributes see [12].

The LDAP model assumes a global name spaces for attributes. Object classes group a number of attributes. If an object class inherits the same attribute from different parents then a single copy ends up in the class. This implies that a property, for example “cn”, should always be the common name and the type must be a string. It is not allowed to have a property “cn” in another objectclass that is an integer. This model is compatible with the OSGi approach towards property definitions.

4.3 Related standards

There are many related standards which are applicable in this context. However, except for Java beans, all other standards are based on document formats. This is not deemed applicable due to the overhead this would require in the execution environment. One of the primary requirements of this RFC is to have the meta type information available in runtime with minimal overhead. A document format would require a parser during runtime.

Another consideration is the applicability of these standards. Most of these standards were developed for management systems where resources are not a concern. Usually, a meta type standard is used to describe the data structures needed to control “another” computer via a network. This other computer does not usually require the meta type information because it is *implementing* this information.

However, there also exists some traditional cases where a management systems uses the meta type information to control objects in an OSGi environment as in RFC 7 [4]. Therefore, the concepts and the syntax of the meta type information must be mappable to these popular standards. I.e. these standards must be able to describe objects in an OSGi environments. This is usually not a problem because the meta type languages used by current management systems are very powerful.

4.4 Beans

The intention of the beans packages in Java comes very close to the meta type information needed. However, Beans can not be used for the following reasons:

1 Beans require a large number of classes that are likely to be optional for an OSGi profile.

2 Beans have been closely coupled to the graphic subsystem (AWT) and applets. Neither of these packages is available on an OSGi profile

3 Beans are closely coupled with the type safe Java classes. The advantage of properties is that no type safety is used allowing two parties to have an independent versioning model. (no shared classes)

4 Beans allow all possible Java objects, not the OSGi subset as required by this RFC

5 No explicit localization

6 No support for optional properties

5 Technical Discussion

5.1 Meta type interfaces

The meta type model consists of 2 meta data interfaces and one access interface: ObjectClassDefinition, AttributeDefinition and MetaTypeProvider. These classes are defined in the org.osgi.service.metatype package.

The ObjectClassDefinition interface is used to group the attributes which are defined in AttributeDefinitions.

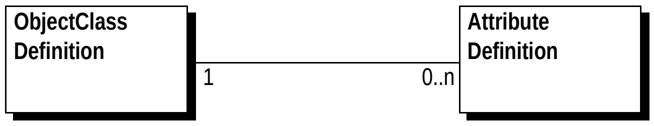

Fig. 1 Meta type classes

5.1.1 ObjectClassDefinition

```
public interface ObjectClassDefinition {
     final int REQUIRED      = 1;
     final int OPTIONAL      = 2;
     final int ALL           = 0xFFFFFFFF;
     public String getName();
     String getID();
     String getDescription();
     AttributeDefinition[] getAttributeDefinitions(int filter);
     InputStream getIcon(int size) throws IOException;
}
```

The ObjectClassDefinition interface contains the information about the overall set of properties. It has the following elements.

1 A name which can be returned in different locales.

2 ObjectClassDefintions share a global namespace in the registry. They share this problem with LDAP/X.500 object classes. In these standards the OSI Object Identifier (OID) is used to uniquely identify object classes. If such an OID exists, (which can be requested at several standard organisations and many companies already have a node in the tree) it can be returned here. Otherwise, a unique id should be returned which can be a java class name (reverse domain name) or generated with a GUID algorithm. Note that all LDAP defined object classes already have an OID associated. It is strongly advised to define the object classes from existing LDAP schemes which will give the OID for free. Many such schemes exist ranging from postal addresses to DHCP parameters.

3 A description of the class for human consumption.

4 A list of attribute definitions where it can be filtered on required, optional or all attributes. (Notice that in X.500 the mandatory or required status of an attribute is part of the ObjectClass and not of the attribute).

5 An icon, with an optional hint for its size.

5.1.2 AttributeDefinition interface

```
public interface AttributeDefinition {
    final int STRING       = 1;
    final int LONG         = 2;
    final int INTEGER      = 3;
    final int SHORT        = 4;
    final int CHAR         = 5;
    final int BYTE         = 6;
    final int DOUBLE       = 7;
    final int FLOAT        = 8;
    final int BIGINTEGER   = 9;
    final int BIGDECIMAL   = 10;
    final int BOOLEAN      = 11;

    String getName();
    String getID();
    String getDescription();
    int getCardinality();
    int getType();
    String validate(String value);
    String[] getOptionValues();
    String[] getOptionLabels();
    String[] getDefaultValue();
}
```

The AttributeDefinition interfaces defines the following elements:

1. Defined names (final ints) for the data types as restricted in the OSGi framework for the properties. This is called the syntax in OSI terms.

2. AttributeDefintions share a global namespace in the registry. They share this problem with LDAP/X.500 attributes. In these standards the OSI Object Identifier (OID) is used to uniquely identify attributes. If such an OID exists, (which can be requested at several standard organisations and many companies already have a node in the tree) it can be returned here. Otherwise, a unique id should be returned which can be a java class name (reverse domain name) or generated with a GUID algorithm. Note that all LDAP defined attributes already have an OID associated. It is strongly advised to define the object classes from existing LDAP schemes which will give the OID for free. Many such schemes exist ranging from postal addresses to DHCP parameters.

3. A localized name intended to be used in user interfaces.

4. A localized description that defines the semantics of the attribute and possible constraints. This should be usable for tooltips.

5. An indication if this attribute should be stored as unique value, a Vector or an array of values. Also the maximum cardinality of the type.

6. The data type, as limited by the OSGi framework registry property types.

7. A validation function to verify if a possible value is correct.

8. A list of values and alist of localized labels. Intended for popup menus in GUIs, allowing the user a choice from a set.

9. A default value. The return type of this is a String[]. For cardinality=0, this must be an array of 1 String. For other cardinalities, the array must not contain more than |cardinality| String objects. In that case, it may contain 0 objects.

5.1.3 MetaTypeProvider interface

```
public interface MetaTypeProvider {
    ObjectClassDefinition getObjectClassDefinition(String pid,
        String locale);
    String[] getLocales();
}
```

The meta type provider interface gives access to the meta type information. It is used in management systems and runtime management. It suports locales so that the text used in AttributeDefinitions and ObjectClassDefinitions can be adapted to different locales.

The pid is given as a parameter in getObjectClassDefinition method so that a single object can be registered as different services with their own PID.

Locale objects are represented in String objects because not all profiles support Locale. The String holds the standard Locale presentation of <language> [_ <country> [_ <variation>]].

5.2 Example

AttributeDefinition and ObjectClassDefinition classes are intended to be easy to use for bundles. This example shows a potential implementation for this classes (except the get methods) and their usage.

First the ObjectClassDefinition class is implemented in OCD. The name is made very short because the class is used to instantiate the static structures. Normally many of these objects are instantiated very close to each other and long names would make these lists of instantiations very long.

```
class OCD implements ObjectClassDefinition {
    String                  name;
    String                  id;
    String                  description;
    AttributeDefinitionrequired[];
    AttributeDefinitionoptional[];

    public OCD( String name, String id, String description,
        AttributeDefinition required[],
        AttributeDefinition optional[] ) {
        this.name = name;
```

```
          this.id = id;
          this.description = description;
          this.required = required;
          this.optional = optional;
     }
     .... All the get methods
}
```

The second class is the AD class that implements the AttributeDefinition. The name is so short for the same reason as in OCD. Note the two different constructors to simplify the common case.

```
class AD implements AttributeDefinition {
     String         name;
     String         id;
     String         description;
     int            cardinality;
     int            syntax;
     String[]       values;
     String[]       labels;
     String[]       deflt;

     public AD( String name, String id, String description, int syntax,
          int cardinality, String values[], String labels[], String deflt[] ) {
          this.name = name;
          this.id = id;
          this.description = description;
          this.cardinality = cardinality;
          this.syntax = syntax;
          this.values = values;
          this.labels = labels;
     }
     public AD( String name, String id, String description, int syntax ) {
          this( name, id, description, syntax, 0, null, null, null );
     }
     ... All the get methods + validate
}
```

The last part is the example that imlements the MetatTypeProvider. Only one locale is supported, the US locale. The OIDs used in this example are the actual official OIDs

```
public class Example implements MetaTypeProvider {

     final static AD cn =
          new AD( "cn",           "2.5.4.3", "Common name",       AD.STRING );
     final static AD sn =
          new AD( "sn",           "2.5.4.4", "Sur name",          AD.STRING );
     final static AD description =
          new AD( "description",  "2.5.4.13","Description",       AD.STRING );
     final static AD seeAlso =
          new AD( "seeAlso",      "2.5.4.34", "See Also reference", AD.STRING );
     final static AD telephoneNumber =
          new AD( "telephoneNumber", "2.5.4.20", "Telephone number", AD.STRING );
     final static AD userPassword =
          new AD( "userPassword", "2.5.4.3", "User password", AD.STRING );
     final static ObjectClassDefinition person =
          new OCD( "person", "2.5.6.6", "Defines a person",
               new AD[] { cn, sn },
               new AD[] { description, seeAlso, telephoneNumber, userPassword }
     );

     public ObjectClassDefinition getObjectClassDefinition( String pid, String
locale) {
          return person;
     }
    public String[] getLocales() { return new String[] { “en_US” }; }
}
```

The example code shows that the attributes/properties are defined in AD objects using final static. The examples groups a number of attributes/properties together in an OCD.

As can be seen from this example, the footprint issues for using AttributeDefinition, ObjectClassDefinition and MetaTypeProvider are minimal.

6 Security Considerations

No special security issues are foreseen for this API because the API only supplies information which is not security sensitive.

7 org.osgi.service.metatype package

Package org.osgi.service.metatype

Class Summary	
Interfaces	
`AttributeDefinition`	An interface to describe an attribute.
`MetaTypeProvider`	Provides access to meta types.
`ObjectClassDefinition`	Description for the data type information of an objectclass.

org.osgi.service.metatype
AttributeDefinition

Syntax
```
public interface AttributeDefinition
```

Description
An interface to describe an attribute. An AttributeDefinition defines a description of the data type of a property/ attribute.

Member Summary	
Fields	
int	BIGDECIMAL
int	BIGINTEGER
int	BOOLEAN
int	BYTE
int	CHARACTER
int	DOUBLE
int	FLOAT
int	INTEGER
int	LONG
int	SHORT
int	STRING
Methods	
int	getCardinality()
String[]	getDefaultValue()
String	getDescription()
String	getID()
String	getName()
String[]	getOptionLabels()
String[]	getOptionValues()
int	getType()
String	validate(String)

Fields

BIGDECIMAL
```
public static final int BIGDECIMAL
```
The BIGDECIMAL(10) type. Attributes of this type should be stored as BigDecimal objects, Vector with Big-Decimal or BigDecimal[] depending on getCardinality().

BIGINTEGER

`public static final int` **`BIGINTEGER`**

The BIGINTEGER(9) type. Attributes of this type should be stored as BigInteger objects, Vector with BigInteger or BigInteger[] depending on getCardinality().

BOOLEAN

`public static final int` **`BOOLEAN`**

The BOOLEAN(11) type. Attributes of this type should be stored as Boolean objects, Vector with Boolean or boolean[] depending on getCardinality().

BYTE

`public static final int` **`BYTE`**

The BYTE(6) type. Attributes of this type should be stored as Byte objects, Vector with Byte or byte[] depending on getCardinality().

CHARACTER

`public static final int` **`CHARACTER`**

The CHARACTER(5) type. Attributes of this type should be stored as Character objects, Vector with Character or char[] depending on getCardinality().

DOUBLE

`public static final int` **`DOUBLE`**

The DOUBLE(7) type. Attributes of this type should be stored as Double objects, Vector with Double or double[] depending on getCardinality().

FLOAT

`public static final int` **`FLOAT`**

The FLOAT(8) type. Attributes of this type should be stored as Float objects, Vector with Float or float[] depending on getCardinality().

INTEGER

`public static final int` **`INTEGER`**

The INTEGER(3) type. Attributes of this type should be stored as Integer objects, Vector with Integer or int[] depending on getCardinality().

LONG

`public static final int` **`LONG`**

The LONG(2) type. Attributes of this type should be stored as Long objects, Vector with Long or long[].

SHORT

```
public static final int SHORT
```

The SHORT(4) type. Attributes of this type should be stored as BigDecimal objects, Vector with BigDecimal or BigDecimal[] depending on getCardinality().

STRING

```
public static final int STRING
```

The STRING(1) type. Attributes of this type should be stored as String objects, Vector with String or String[] depending on getCardinality().

Methods

getCardinality()

```
public int getCardinality()
```

Return the cardinality of this attribute. The OSGi framework can store multi valued attributes in arrays ([]) or in vectors. The return value is defined as follows:

```
x = Integer.MIN_VALUE    no limit, but use Vector
x < 0       -x = max occurrences, store in Vector
x > 0      x = max occurrences, store in array []
x = Integer.MAX_VALUE    no limit, but use array []
x = 0 1 occurrence required
```

getDefaultValue()

```
public java.lang.String[] getDefaultValue()
```

Return a default for this attribute. The Object must be of the appropriate type as defined by the cardinality and getType(). The return type is a list of Strings that can be converted to the appropriate type. The cardinality of the return array must follow the absolute cardinality of this type. E.g. if the cardinality = 0, the array must contain 1 element. If the cardinality is 1, it must contain 0 or 1 elements. If it is -5, it must contain from 0 to max 5 elements. Note the special case of a 0 cardinality that does not allow 0 elements.

getDescription()

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

Return a description of this attribute. The description may be localized and must describe the semantics of this type and any constraints.

getID()

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

Unique identity for this attribute. Attributes share a global namespace in the registry. E.g. an attribute ’cn’ or ’commonName’ must always be a String and the semantics are always a name of some object. They share this problem with LDAP/X.500 attributes. In these standards the OSI Object Identifier (OID) is used to uniquely identify an attribute. If such an OID exists, (which can be requested at several standard organisations and many companies already have a node in the tree) it can be returned here. Otherwise, a unique id should be returned which can be a java class name (reverse domain name) or generated with a GUID algorithm. Note that all LDAP defined attributes already have an OID associated. It is strongly advised to define the attributes from existing LDAP schemes which will give the OID. Many such schemes exist ranging from postal addresses to DHCP parameters. OID

getName()

`public java.lang.String getName()`

Get the name of the attribute. This name may be localized.

getOptionLabels()

`public java.lang.String[] getOptionLabels()`

Return a list of labels of option values.

The purpose of this method is to allow menus with localized labels. It is associated with getOptionValues. The labels returned here are ordered in the same way as the values in that method.

If the function returns null, there are no option labels available.

This list must be in the same sequence as getOptionValues(). I.e. for each index i in getOptionLables, i in getOptionValues() should be the associated value.

For example, if an attribute can have the value male, female, unknown, this list can return (for dutch) new String[] { “Man”, “Vrouw”, “Onbekend” }.

getOptionValues()

`public java.lang.String[] getOptionValues()`

Return a list of option values that this attribute can take.

If the function returns null, there are no option values available.

Each value must be acceptable to validate() (return “”) and must be a String that can be converted to the data type defined by getType() for this attribute.

This list must be in the same sequence as getOptionLabels(). I.e. for each index i in getOptionValues, i in getOptionLabels() should be the label.

For example, if an attribute can have the value male, female, unknown, this list can return new String[] { “male”, “female”, “unknown” }.

getType()

`public int getType()`

Return the type for this attribute. Defined in the following constants which map to the appropriate Java type. STRING, LONG, INTEGER, CHAR, BYTE, DOUBLE, FLOAT, BIGINTEGER, BIGDECIMAL, BOOLEAN.

validate(String)

`public java.lang.String validate(java.lang.String value)`

Validate an attribute in String form. An attribute might be further constrained in value. This method will attempt to validate the attribute according to these constraints. It can return three different values:

```
null no validation present
“”no problems detected
“...” A localized description of why the value is wrong
```

Parameters:

`value` - The value before turning it into the basic data type

org.osgi.service.metatype
MetaTypeProvider

Syntax
```
public interface MetaTypeProvider
```

Description
Provides access to meta types.

Member Summary	
Methods	
`String[]`	`getLocales()`
`ObjectClassDefinition`	`getObjectClassDefinition(String, String)`

Methods

getLocales()

```
public java.lang.String[] getLocales()
```

Return a list of locales available or null if only 1 The return parameter must be a name that consists of language [_ country [_ variation]] as is customary in the java.util.Locale class. This Locale class is not used because certain profiles do not contain it.

getObjectClassDefinition(String, String)

```
public ObjectClassDefinition getObjectClassDefinition(java.lang.String pid,
            java.lang.String locale)
```

Return the definition of this object class for a locale. The locale parameter must be a name that consists of language [_ country [_ variation]] as is customary in the java.util.Locale class. This Locale class is not used because certain profiles do not contain it.

The implementation should use the locale parameter to match an ObjectClassDefinition. It should follow the customary locale search path by removing the latter parts of the name.

Parameters:

`pid` - The PID for which the type is needed or null if there is only 1

`locale` - The locale of the definition or null for default locale org.osgi.service.metatype

ObjectClassDefinition

Syntax

```
public interface ObjectClassDefinition
```

Description

Description for the data type information of an objectclass.

Member Summary

Fields	
`int`	`ALL`
`int`	`OPTIONAL`
`int`	`REQUIRED`
Methods	
`AttributeDefinition[]`	`getAttributeDefinitions(int)`
`String`	`getDescription()`
`InputStream`	`getIcon(int)`
`String`	`getID()`
`String`	`getName()`

Fields

ALL

```
public static final int ALL
```

Parameter for getAttributeDefinitions(int). ALL indicates that all the definitions are returned. The value is -1.

OPTIONAL

```
public static final int OPTIONAL
```

Parameter for getAttributeDefinitions(int). OPTIONAL indicates that only the optional definitions are returned. The value is 2.

REQUIRED

```
public static final int REQUIRED
```

Parameter for getAttributeDefinitions(int). REQUIRED indicates that only the required definitions are returned. The value is 1.

Methods

getAttributeDefinitions(int)

```
public AttributeDefinition[] getAttributeDefinitions(int filter)
```

Return the attribute definitions. Return a set of attributes. The filter parameter can distinguish between ALL, REQUIRED or the OPTIONAL attributes.

Parameters:
`filter` - ALL, REQUIRED, OPTIONAL

Returns: An array of attribute definitions or null if no attributes are selected

getDescription()

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

Return a description of this objectclass. The description may be localized.

Returns: The localized description of the definition.

getIcon(int)

```
public java.io.InputStream getIcon(int size)
```

Return an inputstream that can be used to create an icon from. Indicate the size and return an inputstream containing an icon. The returned icon maybe larger or smaller than the indicated size.

The icon may depend on the localization.

Parameters:
`sizeHint` - size of an icon, e.g. a 16x16 pixels icon then size = 16

Returns: An InputStream representing an icon or null

Throws:
`IOException`

getID()

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

Return the id of this object class. ObjectDefintions share a global namespace in the registry. They share this problem with LDAP/X.500 attributes. In these standards the OSI Object Identifier (OID) is used to uniquely identify object classes. If such an OID exists, (which can be requested at several standard organisations and many companies already have a node in the tree) it can be returned here. Otherwise, a unique id should be returned which can be a java class name (reverse domain name) or generated with a GUID algorithm. Note that all LDAP defined object classes already have an OID associated. It is strongly advised to define the object classes from existing LDAP schemes which will give the OID for free. Many such schemes exist ranging from postal addresses to DHCP parameters.

Returns: The id or oid

getName()

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

Return the name of this class.

Returns: The name of the described class.

8 Document Support

8.1 References

[1]. RFP 0005 Request For Proposal for Remote Management

[2]. RFP 0006 Request For Proposal for Execution Profile

[3]. RFP 0016 Meta data for properties

[4]. RFC 0007 Configuration Management

[5]. OSGi Framework specification http://www.osgi.org

[6]. Common Information Model http://www.dmtf.org

[7]. SNMP RFCs http://directory.google.com/Top/Computers/Internet/Protocols/SNMP/RFCs

[8]. XSchema, http://www.w3.org/TR/xmlschema-0/

[9]. IDL, http://www.omg.org

[10]. LDAP, http://directory.google.com/Top/Computers/Software/Internet/Servers/Directory/LDAP

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

[12]. Understanding and Deploying LDAP Directory services, Timothy Howes et. al. ISBN 1-57870-070-1, MacMillan Technical publishing.

8.2 Author's Address

Name	Peter Kriens
Company	Ericsson
Address	Finnasandsvagen 22
Voice	+46 70 5950899
e-mail	Peter.Kriens@aQute.se