Metadata Deploy Module

- Overview
- Usage
  - Bundles
  - Synchronization
- Extending
  - Supporting new types
- Links
- Change log

Overview

It's common for modules or distributions of modules to require certain metadata objects to exist in the installation. One solution to this problem has to been to bundle metadata sharing packages with the code and have an activator method install those on start up (if they aren't already installed). The major weaknesses of this approach are:

- The metadata is not readable or editable in the code. The packages typically have to be edited on an external server, exported and embedded into the code.
- Package installation is slow so it's usually not appropriate to use the same packages in unit tests.
- The metadata may be subsequently modified or removed by users breaking expectations in the code.

The metadata deploy module seeks to provide a new mechanism of bundling metadata with module or distribution code that addresses these issues. That mechanism should allow module developers to adhere to these principles:

- Metadata should be directly readable or editable from the code and packages only used as a last resort for data which is difficult to describe in any other way.
- Metadata installation should be fast enough that it can be used within unit tests.
- When a module starts, it should have a guarantee that its metadata exists in the database exactly as expected.

Usage

This is a support module for developers and as such there is no UI. The following section describes how to use the module in another module or distribution.

Bundles

Bundles are containers for metadata which allow it to be grouped into meaningful categories. For example a distribution might decide to group its metadata by program area and so would create separate bundles for different program areas, e.g. *CommonMetadata*, *HivMetadata*, *TbMetadata* etc. Grouping metadata in this way can make it easier to find specific items and also allow you to be selective about which bundles are required to run particular unit tests.

A bundle has two purposes:

1. To provide identifiers for all of its metadata
2. To perform the installation of that metadata

The general pattern for bundle Java class is this:

General bundle pattern

```
@Component
@Requires(...) // Dependencies on other bundles
public class ExampleMetadata extends AbstractMetadataBundle {

    public static final class _ObjectType1 {
        public static final String OBJECT1_NAME = "..."; // Typically the UUID of object #1
    }

    public static final class _ObjectType2 {
        public static final String OBJECT2_NAME = "...";
        public static final String OBJECT3_NAME = "...";
    }

    public void install() {
        // Do the actual installation of the items
    }
}
```

A real world bundle might look something like the following example:

Example bundle

```
@Component
@Requires({ BaseMetadata.class })
public class MyMetadata extends AbstractMetadataBundle {

    public static final class _EncounterType {
        public static final String ENCOUNTER_TYPE1 = "d3e3d723-7458-4b4e-8998-408e8a551a84";
    }

    public static final class _Form {
        public static final String FORM1 = "4b296dd0-f6be-4007-9eb8-d0fd4e94fb3a";
        public static final String FORM2 = "89994550-9939-40f3-afa6-173bce445c79";
    }

    @Override
    public void install() {
        install(encounterType("Encounter Type #1", "Something...", _EncounterType.ENCOUNTER_TYPE1));

        install(form("Form #1", null, _EncounterType.ENCOUNTER_TYPE1, "1", _Form.FORM1));
        install(form("Form #2", null, _EncounterType.ENCOUNTER_TYPE1, "1", _Form.FORM2));

        // A form that should be retired if it exists
        uninstall(possible(Form.class, "73d34479-2f9e-4de3-a5e6-1f79a17459bb"), "Because...");
    }
}
```

Object identification and fetching

The bundle pattern gives us a convenient name-spaced way to reference the metadata items throughout the rest of the module code, e.g. `MyMetadata._Form.FORM1`. This gives us the unique identifier of that object that can then be passed to a relevant fetch method. In this example `_Form` is the class of FORM1, prefixed with an underscore to avoid a name conflict with the actual `Form` class. We can fetch the form object in one of two ways:

1. Assuming that it exists and throwing an exception otherwise - we use this for metadata which our module can't function without.
2. Not assuming it exists and explicitly checking whether fetch returns null - we use this for metadata which may or may not exist.

The `MetadataUtils` class provides two static methods to handle these cases. The first of these throws a `MissingMetadataException` if the requested item doesn't exist, e.g.

Fetching a form which is assumed to exist

```
Form form1 = MetadataUtils.existing(Form.class, MyMetadata._Form.FORM1);
```

An application could choose to handle the exception, but more likely not as we're assuming the item exists and *failing fast* if it doesn't. The exception will provide the developer with the information needed to quickly track down the problem.

The second of these methods returns `null` if the item doesn't exist so the onus is on the calling module to handle that, e.g.

Fetching a form which may or may not exist

```
Form form1 = MetadataUtils.possible(Form.class, MyMetadata._Form.FORM1);
if (form1 == null) {
  // Module must be careful to handle this safely itself
}
```

Inside a bundle class there are equivalent `possible(...)` or `existing(...)` methods.

For most objects, the identifier is the *UUID* as this uniquely identifies the object across different installations. The exceptions are classes like `Role`, `Privilege` and `GlobalProperty` as these objects can be uniquely identified via their name, and so using UUIDs to reference these objects adds unnecessary complexity.

Object installation

The metadata deploy module provides a way to quickly install transient OpenMRS objects into the database. By "install" we mean:

> *Object installation guarantees that there will be an object in the database with the given identifier that matches exactly the object described in the code*

In practice this means:

- If a matching object doesn't exist in the database, it is created new.
- If a matching object exists in the database, it is completely overwritten. This is to ensure that the object is exactly as the code expects.

The module looks for a matching object using the following logic:

- If another object exists with that identifier
- If the class handler finds an alternate match. This is used for objects that have more than one unique property, e.g. you can't have two `Program` objects with the same name.

In the example bundle above, `encounterType(...)` and `form(...)` are statically imported methods which function as convenience constructors for those classes. They produce a transient object which the `install(...)` method then installs to the database.

Object sources

When dealing with large collections of objects it it sometimes more appropriate to define objects in a resource file rather than actual code. An object source is anything that produces transient objects for installation. Module developers can create their own object sources and pass these to `install(...)` to have metadata deploy install all objects from that source.

For example you might have a list of locations in a CSV file with the format: name, description, UUID. You could create a CSV object source to load these, e.g.

Defining a CSV based object source

```
public class LocationCsvSource extends AbstractCsvResourceSource<Location> {

        public LocationMflCsvSource(String csvFile) throws IOException {
                super(csvFile, false);
        }

        @Override
        public Location parseLine(String[] line) {
                Location location = new Location();
                location.setName(line[0]);
                location.setDescription(line[1]);
                location.setUuid(line[2]);
                return location;
        }
}
```

This source could be used and installed inside a bundle as follows:

Installing from a source

```
install(new LocationCsvSource("locations.csv"));
```

Object uninstallation

Sometimes a module might want to remove an object. Assuming that the object may or may not exist, this can easily done from inside a bundle with a combination of `possible(...)` and `uninstall(...)`, e.g.

Uninstalling an object

```
uninstall(possible(Form.class, "73d34479-2f9e-4de3-a5e6-1f79a17459bb"), "Give a reason...");
```

Depending on the object class, the object might be retired, voided or purged. If the object doesn't exist then `possible(...)` returns `null` and `uninstall(...)` does nothing.

Installing packages

If you need to use metadata sharing packages you can install those from inside a bundle, e.g.

Bundle containing a package

```
@Component
@Requires({ BaseMetadata.class })
public class LocationsMetadata extends AbstractMetadataBundle {

    public static final class Package {
        public static final String LOCATIONS = "5856a8fc-7ebc-46e8-929c-5ae2c780ab54";
    }

    @Override
    public void install() {
        install(packageFile("locations-1.zip", null, Package.LOCATIONS));
    }
}
```

If the package is already installed in the database at that version, then it won't be installed again.

> Package filenames must be appended with the version number of the package so that the module can determine the version without extracting the package contents.

Bundle dependencies

Metadata in one bundle might reference metadata in another, and so can't be installed until the metadata in the other bundle has been installed. This can be enforced using a *Requires* relationship, e.g.

Dependencies between bundles

```
@Component
@Requires({ CommonMetadata.class })
public class TbMetadata extends AbstractMetadataBundle { ... }
```

Installing bundles during startup

Once a module or distribution has defined its bundles, it can instruct the metadata deploy module to install these during startup, e.g.

Bundle installation

```
public void started() {
        MetadataDeployService svc = Context.getService(MetadataDeployService.class);
        svc.installBundles(Context.getRegisteredComponents(MetadataBundle.class));
}
```

Installing during unit tests

Because bundles are components, they can be autowired into test classes and installed before tests are run, e.g.

Bundle use in a unit test

```
public class MyClassTest extends BaseModuleContextSensitiveTest {

        @Autowired
        private CommonMetadata commonMetadata;

        @Before
        public void setup() throws Exception {
            commonMetadata.install();
        }

        @Test
        public void testMethod() { ... }
}
```

Note that when you invoke the install method of a bundle directly, only that bundle will be installed and not any of its required bundles. If a bundle has dependencies, these should be explicitly installed first in the test class.

Synchronization

Object installation ensures that individual metadata objects exist as expected in the database. Sometimes we are also concerned with the set of *all* objects of that type and this is where synchronization comes in:

> *Object synchronization guarantees that the set of all objects of a specific type in the database matches those in an object source*

So synchronization differs from simply "installing all objects from a source" in this fundamental way: a**ny existing object in the database that is not found in the source is uninstalled.**

Defining the synchronization logic

The logic for a synchronization is provided via a custom implementation of `ObjectSynchronization`. For example if we have a list of locations in a CSV file (same format as previous example), we could define a simple synchronization operation as follows:

Location synchronization example

```
@Component
public class LocationSynchronization implements ObjectSynchronization<Location> {
        @Autowired
        private LocationService locationService;

        @Override
        public List<Location> fetchAllExisting() {
                return locationService.getAllLocations(true);
        }

        @Override
        public Object getObjectSyncKey(Location obj) {
                return obj.getUuid();
        }

        @Override
        public boolean updateRequired(Location incoming, Location existing) {
                return true; // Always update the existing object (not very efficient)
        }
}
```

In this example we instruct metadata deploy to always update existing objects in the database. A more efficient approach here is to compare the two objects and only update if they are actually different.

This synchronization can then be performed during a bundle installation by passing it and a suitable object source to the `sync(...)` method, e.g.

Synchronization within a bundle

```
@Component
public class LocationsMetadata extends AbstractMetadataBundle {

        @Autowired
        private LocationSynchronization locationSync;

        @Override
        public void install() throws Exception {
                sync(new LocationCsvSource("locations.csv"), locationSync);
        }
}
```

To maximize performance, internally metadata deploy will maintain a cache of all existing objects.

You can synchronize on something other than the main identifier of an object by returning something else from `getObjectSyncKey(...)`. This is useful if you are synchronizing against a list of objects defined outside of OpenMRS which have some other unique identifier.

Extending

Supporting new types

The module currently has support for most metadata objects in OpenMRS core but may be missing some. If you find it is missing support for an class defined in OpenMRS core then submit a ticket. If you need it to support a class defined outside of core then you can provide support for that type yourself as described below.

Providing new constructors

The class `CoreConstructors` contains convenience constructors for many different classes. It is straightforward to define your own class of constructors which can be statically imported into a bundle class. You can use this to define constructors for new types, or additional constructors for core types, e.g.

Custom constructor class example

```
public class MyConstructors {
  public static MyMetadataType myMetadataType(String name, String description, String uuid) {
    MyMetadataType obj = new MyMetadataType();
    obj.setName(name);
    obj.setDescription(description);
    obj.setUuid(uuid);
    return obj;
  }
}
```

Which can then be used in a bundle like this:

Custom constructor use in a bundle example

```
import static com.example.MyConstructors.*;

public class MyMetadata extends AbstractMetadataBundle {

  public static final class _MyMetadataType {
    public static final String EXAMPLE = "88B7D480-5147-4B44-AC66-9F5A20822F7";
  }

  public void install() {
    install(myMetadataType("Name", "Testing", _MyMetadataType.EXAMPLE));
  }
}
```

Providing new handlers

If you want to include a new object class in a bundle then you will also have to tell the module how to handle that class. You can do this by providing a new "deploy handler" component. For example the deploy handler for locations looks like this:

The location deploy handler

```
@Handler(supports = { Location.class })
public class LocationDeployHandler extends AbstractObjectDeployHandler<Location> {
        @Autowired
        @Qualifier("locationService")
        private LocationService locationService;

        @Override
        public Location fetch(String uuid) {
                return locationService.getLocationByUuid(uuid);
        }

        @Override
        public Location save(Location obj) {
                return locationService.saveLocation(obj);
        }

        @Override
        public void uninstall(Location obj, String reason) {
                locationService.retireLocation(obj, reason);
        }
}
```

Links

- Module repository: https://addons.openmrs.org/#/show/org.openmrs.module.metadatadeploy
- Source code: https://github.com/openmrs/openmrs-module-metadatadeploy
- Issue tracker: https://tickets.openmrs.org/browse/DPLY

Change log

1.8.1

Key	Summary	T	Created	Updated	Due	Assignee	Reporter	P	Status	Resolution
DPLY-41	Program Metadata availability		2017-03-15	2017-03-16		Nicholas Ingosi [X]	Nicholas Ingosi [X]		CLOSED	Fixed

1 issue

1.2

Key	Summary	T	Created	Updated	Due	Assignee	Reporter	P	Status	Resolution
DPLY-8	Add existing and possible methods to MetadataUtils		2014-03-19	2014-03-27		Rowan Seymour	Rowan Seymour		CLOSED	Fixed
DPLY-7	Add support for ConceptSource		2014-03-10	2014-03-10		Rowan Seymour	Rowan Seymour		CLOSED	Fixed
DPLY-6	Add support for FormResource		2014-02-17	2014-02-18		Rowan Seymour	Rowan Seymour		CLOSED	Fixed

3 issues

1.1

type	key	summary	assignee	reporter	priority	status	resolution	created	updated	due

Can't show details. Ask your admin to whitelist this Jira URL.

View these issues in Jira

1.0

Initial public release.