Metadata Mapping (Design Page)

Assigned to: Rafal Korytkowski

Mentor: Rafal Korytkowski

Background

OpenMRS has a mechanism to map concepts to external sources (`concept_map` tables); however, OpenMRS does not yet have a means to map metadata to external sources. We have tried using UUIDs to uniquely (and globally) specify metadata across systems; however, this has failed us in several ways: (1) UUIDs end up needing to be manually entered by developers and implementers; (2) there are many cases where metadata created separately (with different UUIDs) are actually referring to the same thing; and, (3) modules introducing their own metadata have a difficult time properly working with existing metadata.

Goals

The goal of Metadata Mapping is to solve many of the problems around metadata management by providing an easy & explicit way for metadata within a system (encounter types, location, etc.) to be mapped to external vocubularies.

Use case examples:

- Developers and implementers can refer to metadata by its more human-friendly mapping (e.g., source + code) instead of having to manually manipulate UUIDs.
- Implementations with multiple servers no longer need to synchronize metadata UUIDs across systems; rather, tools can be designed to use metadata mappings to define sameness of metadata across servers.
- Modules can use metadata mappings to allow administrators to create new metadata or map to existing metadata as needed.

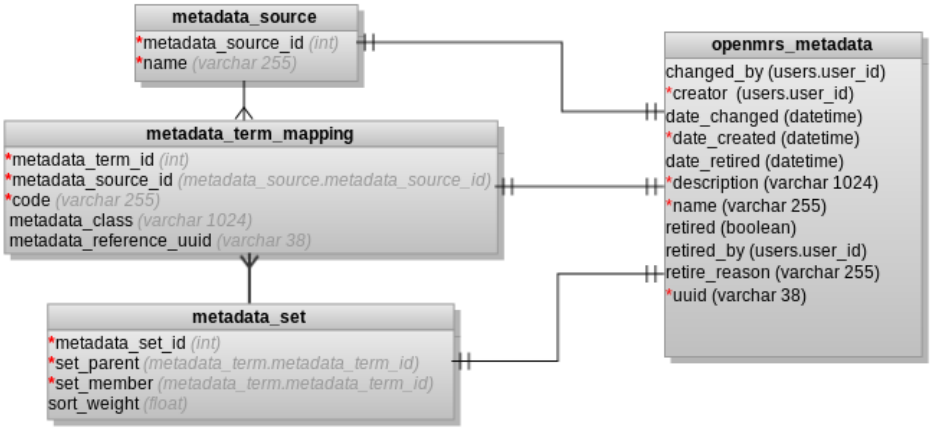
A "1.0" version of Metadata Mapping would include a `MetadataService` within openmrs-core (could be introduced by a module initially or for the foreseeable future) and REST API endpoints to manage & search metadata mappings.

Assumptions

- Source provides the "namespace" of codes and should be universally unique. We should encourage use of fully qualified names (modules defining metadata should use their fully qualified module ID, organizations should use their domain name, etc.); otherwise, managing the uniqueness of metadata sources (e.g., creating a registry) is out of scope for this effort.
- We want to map from human-friendly source + code to the actual metadata (not just a row in a database table), so we map to Class and instance information (as "reference") instead of just linking to a row in a metadata table.
- All metadata mapping are "SAME AS" relationships. Other types of mapping relationships are out of scope for this effort.

Data Model Design

The following design is based on the proposal of the Design Forum 2015-06-22:



Note that in this diagram, the one-to-one mapping between `metadata_source` and `openmrs_metadata`, for example, indicates an *inheritance* in object design terms: MetadataSource inherits the fields of OpenmrsMetadata. In the database schema, this relationship will be implemented so that the relation `metadata_source`, for example, includes all the columns of `openmrs_metadata`.

Notes on data model design

- `metadata_source` is used to define a unique source (authority) for each namespace of metadata terms.

    - `name` should be fully qualified and universally unique.
- `metadata_term_mapping` table provides both the term *and* its mapping to local metadata.
    - `code` should be unique within the given source.
    - `metadata_class` refers to the Java class for the metadata.
    - `metadata_reference` is a unique reference to the metadata within the class (e.g., uuid)
- `metadata_set` is used to define relating grouping of metadata similar to what OpenMRS has traditionally done within global properties and similar to FHIR's ValueSet for metadata terms.
    - `sort_weight` is used to optionally give members of a metadata set a reliable sequence.

API Design

The common operations for metadata mapping will include these use cases:

- Manage mappings (add or remove mappings, should support bulk operations)
- Search for mappings
    - By source
    - By code
    - By metadata object (given an OpenmrsMetadata instance, return any associated mappings)
- Search for metadata (given a mapping UUID or source+code, return the OpenmrsMetadata instance)

API Call examples:

Fetch a location with the given source and code:

```
Location location = metadataService.getItem(Location.class, "SOME-SOURCE", "CODE");
// should be implemented as: generic <T> getItem(T type, String source, String code);
```

Fetch all visit types from the given source:

```
List<VisitType> visitTypes = metadataService.getItems(VisitType.class, "SOME-SOURCE");
// should be implemented as: generic <T> getItems(T type, String source);
```

Fetch all visit types for the given set:

```
List<VisitType> visitTypes = metadataSerivce
  .getItems(VisitType.class, "SOME-SOURCE", "CODE");
// should be implemented as: generic <T> getItems(T type, String source, String code);
// source = metadataSet.source.name, code = metadataSet.code
```

Retire term:

```
List<MetadataTermMapping> termMappings = metadataService.getMetadataTermMappings(location);
metadataService.retireMapping(termMappings.get(0), "some reason");
```

Create term:

```
MetadataSource source = metadataService.getSourceByName("SOME-SOURCE");
MetadataTermMapping termMapping = new MetadataTermMapping(source, "CODE", location);
metadataService.saveMetadataTermMapping(termMapping);
```

Create terms in bulk:

```
List<MetadataTermMapping> termMappings = Arrays.asList(mapping1, mapping2, mapping3);
metadataService.saveMetadataTermMappings(termMappings);
```

Get term by source and code:

```
MetadataTermMapping termMapping = metadataService.getMetadataTermMapping(mappingSource, "CODE");
```

Get terms by source:

```
List<MetadataTermMapping> termMappings = metadataService.getMetadataTermMappings(source);
```

Get term by uuid:

```
MetadataTermMapping termMapping = metadataService.getMetadataTermMappingByUuid(String uuid);
```

Get source by uuid:

```
MetadataSource source = metadataService.getMetadataSourceByUuid(String uuid);
```

MetadataTermMapping, MetadataSource, MetadataSet should support all CRUD operations.