EDIT Common Data Model Library

Reference Documentation (Work in Progress)

Ben Clark
Andreas Müller

EDIT Common Data Model Library: Reference Documentation (Work in Progress)

by Ben Clark and Andreas Müller

2.1

Table of Contents

List of Figures

List of Tables

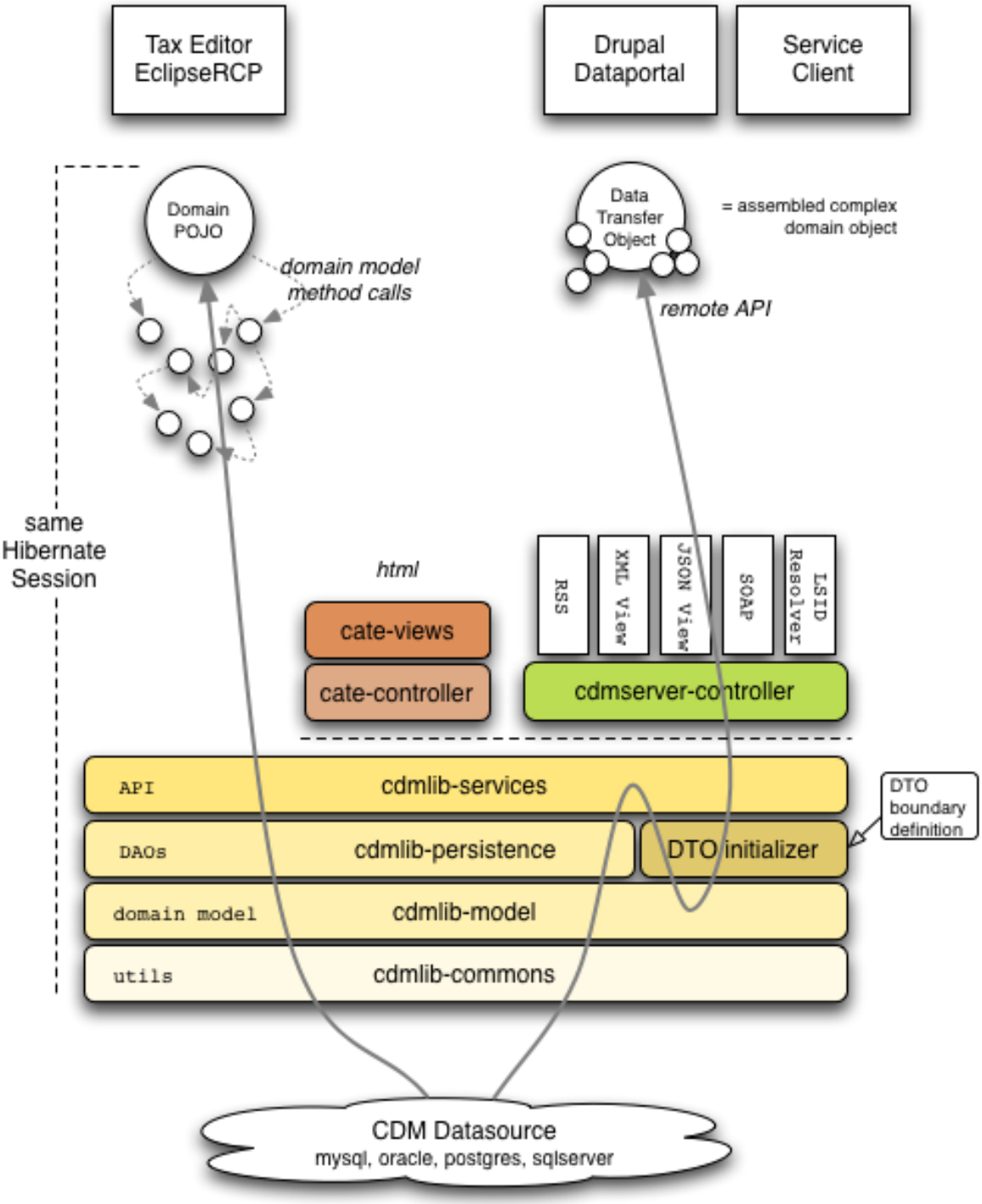
Preface

EDIT's Internet Platform for Cybertaxonomy is a distributed computing platform that helps taxonomists do revisionary taxonomy and taxonomic field work efficiently and expediently via the web. At the core of the platform lies a common data model to enable interoperability between the different components. The model describes all the commonly used data that is dealt with in the platform, and therefore covers taxonomic names and concepts; literature references; authors; (type) specimen; structured descriptive data; molecular data; related (binary) files such as images or compiled keys; controlled vocabularies and terms; and species related content of any kind like economic use or conservation status.

The cyberplatform consists of interoperable but independent components. Platform components can take the form of software applications (desktop or web-based) for human users or (web) services intended to be used by other software applications. The platform as envisioned does not have a single user interface or website; rather, it is a collection of interacting components which may be combined and assembled according to the task in hand. To facilitate the development of core CDM Applications such as the CDM Community Server, the CDM Dataportals, and the Taxonomic Editor, an implementation of the CDM has been created in the java programming language. In addition to CDM model classes being modelled as plain-old-java-objects (pojo's), a set of java components has been created that provide common services across all java applications using the CDM. They serve as the basis of core components of the Internet Platform for Cyberplatform and also allow the development of other applications using the CDM by providing basic functionality that can be extended for a particular purpose.

The CDM Library, as it is known, consists of four major modules that can be used by any java application based on the CDM. These libraries are used as the foundation of the Taxonomic Editor and the CDM Community Server. In addition a web application (the CDM Community Server) is documented here, as its components can be re-purposed or extended by other web applications based on the CDM.

Figure 1. An overview of the main CDM Components



The overall architecture of the EDIT Internet platform for Cybertaxonomy, showing the core components of the CDM Java Library, and their use by desktop (Taxonomic Editor) and web-based (CDM Dataportal, CATE) applications.

This reference documentation is aimed at anyone who would like to understand the software components that make up the core of the cyberplatform: the CDM Java Library and the CDM Server application. More generic information about the applications that make up the cyberplatform, information for end-users of specific applications, and information on the EDIT project itself are beyond the scope of this document. More information about EDIT can be found on the EDIT website, and more information on the specific software applications produced by EDIT can be found on the Work Package 5 website.

Part I. Getting Started

This part of the reference documentation aims to provide simple step-by-step instructions to enable application developers to start using the CDM Java Library in their java application. To do this, we will create a small toy application. The CDM Java Library is packaged and published using the Apache Maven software project managment and comprehension tool. To make life easier, we'll use maven to create our application too. Assuming that Maven (2.0.x+) installed, we begin by creating a new maven application (substituting the group id, artifact id, and version of our application):

```
mvn archetype:create -DgroupId=org.myproject -DartifactId=myapp -Dversion=1.0
```

The next step is to add the EDIT maven repository to your maven *project object model* or *pom* file, thus:

```
. . .
<repositories>
  <repository>
    <id>EditRepository</id>
    <url>http://wp5.e-taxonomy.eu/cdmlib/mavenrepo/</url>
  </repository>
</repositories>
</project>
```

We also need to add the specific dependency that we would like our project to include.

```
. . .
<dependencies>
    <dependency>
      <groupId>eu.etaxonomy</groupId>
      <artifactId>cdmlib-services</artifactId>
      <version>1.1.1</version>
    </dependency>
  </dependencies>
<repositories>
. . .
```

In most cases, application developers will wish to include the cdmlib services (which include the data model and persistence layer too). In some cases, developers might wish to use components from the cdmlib-io and cdmlib-remote packages too. New releases of the CDM Java Library are published in the EDIT Maven Repository, and maven will download and use these artifacts automatically if you change the version number of the dependency specified in your pom file.

All that remains is to set up the cdmlib services within the application context. The CDM Java Library is uses the Spring Framework to manage its components. Whilst it is not mandatory to wire the CDM services and DAOs using Spring, it is certainly easier to configure your application this way. A minimal applicationContext.xml (placed in `src/main/resources`) file might look like this:

```
<import resource="classpath:/eu/etaxonomy/cdm/services.xml" />

<bean id="dataSource"
  lazy-init="true"
  class="eu.etaxonomy.cdm.database.LocalHsqldb"
  init-method="init"
  destroy-method="destroy">
  <property name="driverClassName" value="org.hsqldb.jdbcDriver"/>
```

```
  <property name="username" value="sa"/>
  <property name="password" value=""/>
  <property name="startServer" value="true"/>
  <property name="silent" value="true"/>
</bean>

<bean id="hibernateProperties"
  class="org.springframework.beans.factory.config.PropertiesFactoryBean">
  <property name="properties">
    <props>
      <prop key="hibernate.hbm2ddl.auto">create-drop</prop>
      <prop key="hibernate.dialect">org.hibernate.dialect.HSQLDialect</prop>
      <prop key="hibernate.cache.provider_class">org.hibernate.cache.NoCacheProvider</prop
    </props>
  </property>
</bean>
```

The first element imports the cdmlib service definitions. The two other beans supply a data source and a properties object that the CDM library uses to configure the hibernate session factory and connect to the database. In this case, we're using an in-memory HSQL database, but the CDM can be used with many other databases. The only thing left to do is to start using the CDM services. In real applications, CDM services may well be autowired into components using Spring or another dependency injection mechanism. To keep this example simple, we'll initialize the application context and obtain a service programatically.

```
ApplicationContext context = new ClassPathXmlApplicationContext("applicationContext.xml");

INameService nameService = (INameService)context.getBean("nameServiceImpl");

BotanicalName botanicalName = BotanicalName.NewInstance(Rank.SPECIES());
botanicalName.setGenusOrUninomial("Arum");
botanicalName.setSpecificEpithet("maculatum");
UUID uuid = nameService.saveTaxonName(botanicalName);

System.out.println("Saved \'Arum maculatum\' under uuid " + uuid.toString());
```

In this simple example, we've covered the basics of using the CDM Java Library. We created a simple maven project, and added the repository and a single dependency to our pom file. We then created a simple application context that used the default CDM configuration, and specified a couple of objects that allowed the CDM to connect to a database. Finally we initialized these services by loading the application context, and then retrieved a specific service, and used it to persist a new taxonomic name.

Part II. Common Data Model

The Common Data Model (CDM) is the domain model for the core EDIT cyberplatform components. The CDM is primarily based on the TDWG Ontology and in most cases there is concordance with relevant TDWG standards such as Taxon Concept Transfer Schema (TCS), Structured Descriptive Data (SDD) and Access to Biological Collections Data (ABCD).

The CDM differs from the TDWG standards in its purpose: it is intended to serve as the basis of software applications in the cyberplatform (e.g. the taxonomic editor, the CDM Dataportals) rather than being a standard for data exchange between any resource containing biodiversity information. Whilst it is certainly possible to exchange data as CDM domain objects serialized as XML or JSON (the CDM Server and the CDM Dataportals do this), the common data model is not intended to replace existing TDWG standards as a general purpose exchange standard. It is possible to convert data held in a CDM store into a relevant TDWG standard for exchange and in some cases this may be the desired route for data held in the CDM (e.g. for exchange with an application that is not part of the cyberplatform, but which is capable of understanding data in a TDWG standard).

Thus the CDM is intended for use as

- A domain model for applications, particularly those that enable taxonomists to do revisionary taxonomy and taxonomic field work
- A standard for exchange between applications that are part of the EDIT Internet Platform for Cybertaxonomy

In terms of scope, the CDM covers information core to the vision of the cyberplatform i.e. descriptive and revisionary taxonomy, including taxonomic fieldwork :-

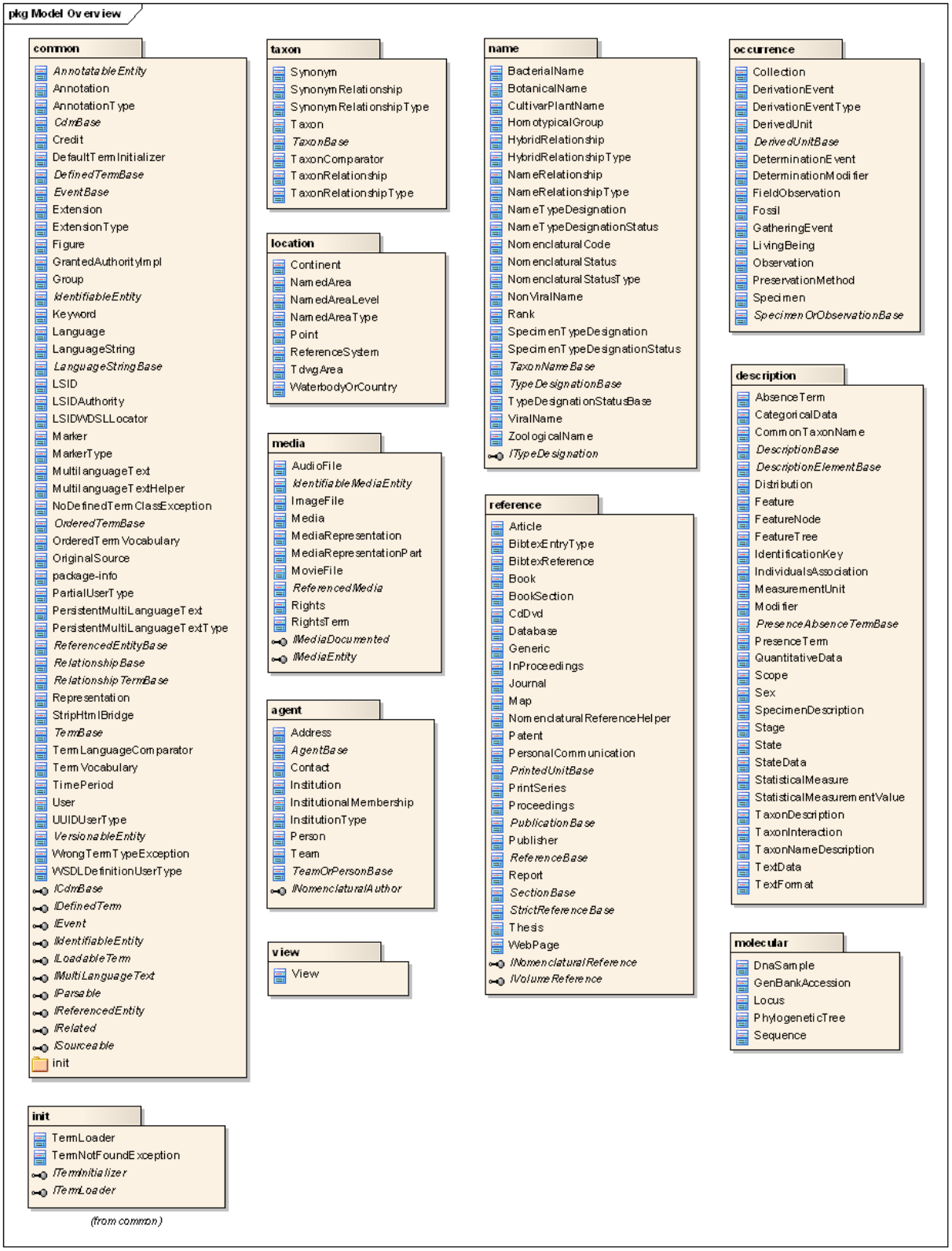
- Taxonomic names and nomenclature, typification
- Taxonomic concepts and relationships between accepted names and synonyms, including the placement of the same taxonomic concept in different taxonomic hierarchies.
- Specimens and Observations of individual organisms, their collection, location, processing and taxonomic determination.
- Structured and unstructured information about names, taxa, and specimens.

In addition to this core area, the CDM covers some related domains that are important:-

- Literature
- People, teams of people and institutions in various roles (i.e. as authors, collectors, artists, rights holders etc)
- Media (images, video and audio files, plus more taxonomy-specific media such as phylogenies and compiled keys)
- Molecular data, such as DNA sequences and loci

As you might expect, there are also a number of data entities representing controlled vocabularies, identity of users (and their roles and permissions), and ancillary data common to all major classes such as multilingual text content, annotations and markers.

Figure 2. A UML Package diagram showing the CDM packages and their members.



Chapter 1. Base Classes

Almost all classes in the CDM implement ICdmBase, an interface that specifies common attributes which are:

```
package eu.etaxonomy.cdm.model.common;

public interface ICdmBase {

/**
 * Returns local unique identifier for the concrete subclass
 * @return
 */
public int getId();

/**
 * Assigns a unique local ID to this object.
 * Because of the EJB3 @Id and @GeneratedValue annotation this id will be
 * set automatically by the persistence framework when object is saved.
 * @param id
 */
public void setId(int id);

public UUID getUuid();

public void setUuid(UUID uuid);

public DateTime getCreated();

/**
 * Sets the timestamp this object was created.
 *
 * @param created
 */
public void setCreated(DateTime created);

public User getCreatedBy();

public void setCreatedBy(User createdBy);
}
```

Although all instances have a primary key (`id`) that is used by any database software, this should not be used to refer to the entity in an application. Instead, a surrogate key (`uuid`) is used to identify entities. Both values are auto-generated, `uuid` when the object is created, `id` at the point the object is persisted (through a call to `save` or `saveOrUpdate`).

Throughout the CDM, temporal data is represented using the Joda Time API rather than the standard java Calender implementation. All CdmBase classes have a property that gives their time of creation (`created`, populated automatically), and the `User` that created the object. The user is retrieved from the security context automatically by the persistence layer (for more on security in the CDM, authentication and authorization, see the section on security). For those applications that do not wish to use the security infrastructure, the User can also be set explicitly by the application.

Versionable Entities

Almost all entities in the CDM are subclasses of `VersionableEntity`. This means that the changing (persistent) state of an entity through time can be recorded in the database, and recovered. This is quite a complex idea and is covered in full in the chapter on versioning. Versionable entities have two additional properties: `updated`, that holds the date-time when the object was last made persistent, and `updatedBy`, that provides the user that last updated the entity. Both work in an identical way to `created` and `createdBy`.

Data model implementation and patterns used across the CDM

It is worth touching on a couple of common patterns used in implementing the CDM in java: *private no-arg constructor*s and*protected access to collection setters*. The ORM technology used in the CDM requires that no-arg constructors exist, and likewise it requires that collections have setter methods as well as getters. However, it is good practice to prevent client application access to these methods to prevent application developers inadvertantly causing mischief (for example, by incorrectly implementing a bidirectional link between a parent and child object).

To instantiate a new CDM entity programmatically, application developers must use one of the public static factory methods provided by the class. Changing the state of single properties is achieved through normal use of getters and setters. In the case of properties that extend `java.util.Collection` or `java.util.Map`, these collections can be changed through `add`*X* and `remove`*X*, where *X* is the property name rather than `set`*X*.

Chapter 2. Annotations and Markers

Many entities in the CDM extend `AnnotatableEntity`. This class contain two properties `annotations` and `markers`. `Annotation`s represent short (currently 4096 characters) statements made by users about the object. `Marker`s are binary (true or false) flags set on an object - markers are be typed using the `MarkerType` vocabulary allowing users to mark objects as *NOT_CHECKED*, *COMPLETE* or any other term.

Chapter 3. Identifiable Entities

Likely to change

Globally unique identifiers, their use and implementation are still an unresolved topic. The implementaton of objects that are identifiable and resolvable in a global sense must, given an open world, reflect the standards and best practices being used by the community as a whole. Consequently this area of the CDM is likely to change to reflect this.

Some entities in the CDM extend `IdentifiableEntity`. In general there is one or two abstract base classes that extend `IdentifiableEntity` in each package. These classes represent important objects that an application might want to share with another application (and hence, publish globally unique identifiers for).

Current implemetation of the GUID in the CDM is based on the LSID Resolution Service implementation of the CATE project. Each `IdentifiableEntity` has an `lsid` property. See the section on GUID Resolution to see how the CDM Java Library makes it easier for you to manage and publish your data.

`titleCache` and `cacheStrategyGenerator`

Classes that extend `IdentifiableEntity` have a `titleCache` attribute. This property is used represent the object (for example, in a list of objects or the title for a page displaying metadata about that object). The `titleCache` is also used by default when sorting lists of entities. Applications are free to use other properties or combinations of properties in representing objects and can supply their own implementations of the interfaces in the eu.etaxonomy.cdm.strategy.cache package. If you do wish to supply custom cache strategies, you will need to inject them into your data entities, overriding the default strategy (using something like Spring's *spring-managed* configuration mechanism).

Recording Provenance using `originalSource`

Indicating ownership and use using `rights` property

Identifiable entities are significant enough that users may wish to indicate ownership of the copyright of the thing that they represent, or to provide some statement of a licence under which that data may be used. This information is held in the `rights` element as a series of `Right` objects, each representing a single rights statement.

Note that the assertation of rights in the data is not the same as access rights in terms of application-level security which is dealt with in a later section.

Part III. Persistence Layer

Even the most basic of taxonomic applications have a requirement for users to be able to save the information that they create. In addition, a common component of taxonomic applications is the use of a database to provide users with the ability to filter or search their data in one way or another. Some applications will require more advanced functionality, such as auditing or versioning of data. All of this logic is contained in the persistence layer, providing clean separation between data access and more taxonomy-centric business logic in the service layer.

Persistence is not a simple problem to solve, especially in application developed in Object-Oriented languages, with large amounts of data, or with many users accessing data at the same time. The CDM Library uses the Hibernate object/relational persistence and query service as the basis of its persistence layer. Several member projects of the Hibernate stable, including Hibernate Annotations, Hibernate Search and Hibernate Envers (part of Hibernate Core) provide the basis of the more advanced persistence-related functionality in the CDM Library. As a consequence some of the behaviour of the CDM Library is constrained by the underlying ORM technology. The advantage of using an ORM is that the same software can be used with multiple database systems with (almost) no changes to the application. Currently the CDM Library has been tested with (version numbers & platforms in brackets)

- IBM DB2
- H2 (default local database used by the Taxonomic Editor, 1.0.73)
- HSQLDB
- MySQL (4.1.20: linux; 5.1.32: windows)
- ODBC
- Oracle Database 11*g*
- PostgreSQL
- Microsoft SQL Server 2000
- Microsoft SQL Server 2005
- Sybase Advantage Database Server

In theory, application developers should not need to use the persistence layer directly, but should instead use the API, which provides a *facade* over the persistence layer and extra business logic that most applications using the CDM will require.

Chapter 4. Basic Persistence

The persistence layer of the CDM primarily consists of a set of *data access objects (DAOs)*. These DAOs are generic, strongly typed, and form a hierachy that reflects the inheritance of the data entities that they provide access too. The root DAO implements `ICdmEntityDao`.

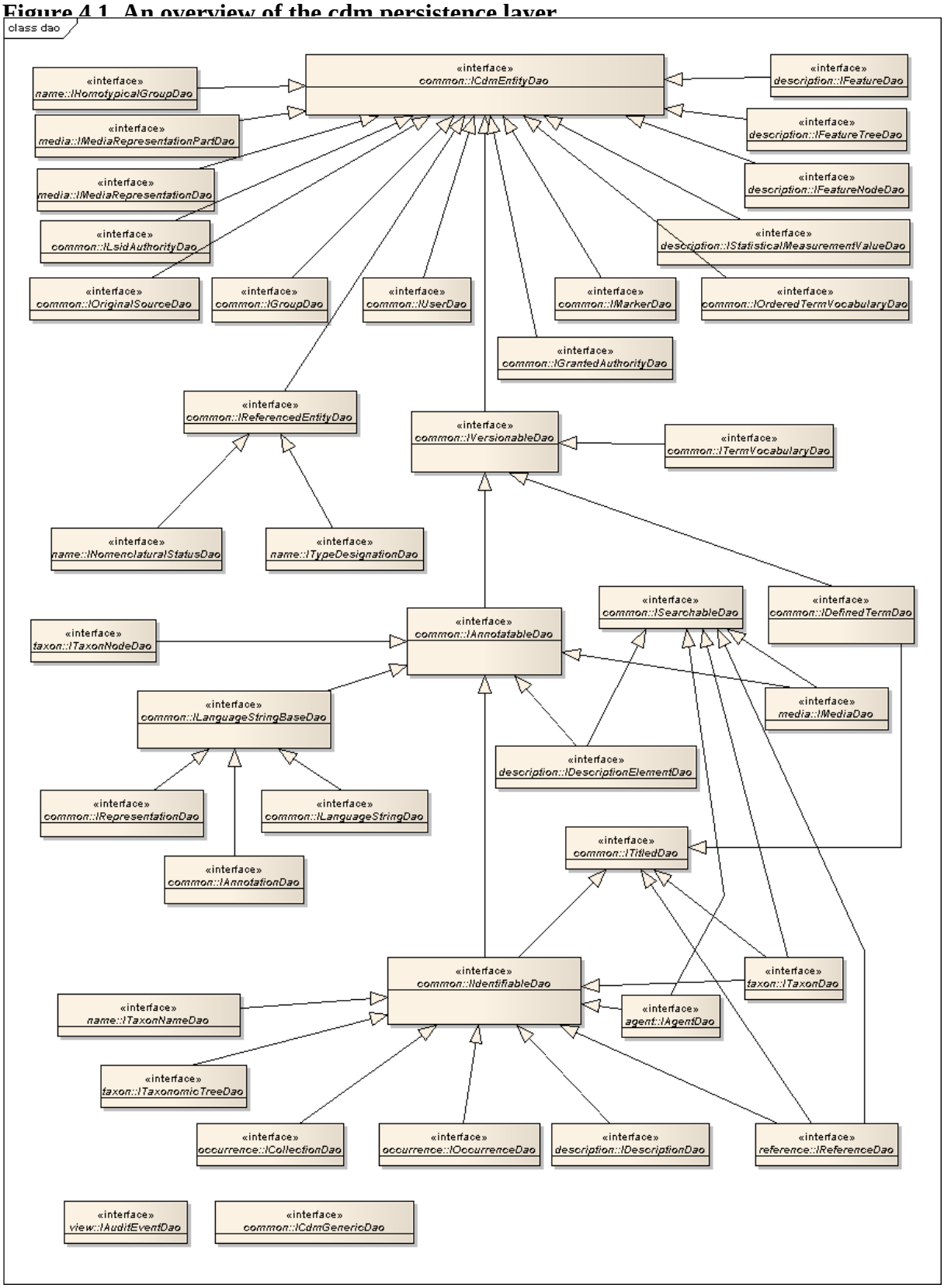
Table 4.1. `ICdmEntityDao` methods

Method	Description
`UUID saveOrUpdate(T newOrTransientEntity);`	Makes a new object persistent, or persists the state of a transient object.
`Map<UUID,T> save(Collection<T> newInstances);`	Makes a collection of new objects persistent.
`UUID save(T newEntity);`	Makes a new object persistent.
`UUID update(T newEntity);`	Makes changes to a transient object persistent.
`UUID merge(T newEntity);`	Merges the state of a detached object into the persisted version.
`UUID delete(T persistentEntity);`	Deletes a persistent object.

`List<T> list(Class<? extends T> clazz,`
`Integer limit,`
`Integer start,`
`List<OrderHint> orderHints,`
`List<String> propertyPaths);`
	Returns a (sub-)list of objects matching the type *`clazz`*, sorted according to the order hints and initialized according to the propertyPaths.

`int count(Class<? extends T> clazz);`	Returns a count of objects matching the type *`clazz`*.
`T find(UUID uuid);`	Returns an object of type T matching the supplied uuid if it exists.
`Collection<T> find(Collection<UUID> uuids);`	Returns a collection of objects of type T matching the uuids supplied, if they exist.
`T load(UUID uuid,` `Collection<String> propertyPaths);`	Returns an object of type T with properties initialized according to the rules described below.
`Set<T> load(Collection<UUID> uuids,` `Collection<String> propertyPaths);`	Returns a collection of objects of type T matching the uuids supplied, if they exist, initialized according to the rules described below.
`boolean exists(UUID uuid);`	Returns true if there is an object of type T in the database matching the supplied uuid.
`Class<T> getType();`	Returns the class of objects that this DAO provides access to.

Figure 4.1. An overview of the cdm persistence layer



The DAO hierarchy in the CDM persistence layer. Data Access Objects are strongly typed and their hierarchy follows the hierarchy of major entities in the CDM.

Object Initialization

DAO methods that return objects, return entities without any relationships initialized by default (to learn more about initialization of related entities, lazy-loading etc, please consult the hibernate documentation). Because some applications (particularly stateless multi-user applications with concise units of work i.e. web applications), may wish to limit the length of transactions, it is important to be able to explicitly initialize related entities according to the particular use-case. The CDM library allows application developers to do this on a per-method call basis.

Properties of the root object specified using java-beans-like syntax and passed using the *`propertyPaths`* parameter will be initialialised before the object(s) are returned and can safely used. Applications that access other properties (that are part of related entities) outside of the transaction in which the entity was retrieved (i.e. the entity is detached) are likely to throw a `LazyInitializationException`. In addition to specifying properties by name, developers can also use an asterisk (*) to represent all *-to-many properties, and a dollar sign ($) to represent all *-to-one properties of the root entity or a related entity. Using a wildcard terminates the property path (i.e. it is not valid syntax to include characters after a wildcard in a propertyPath expression - the wildcard must be the final character in the string).

Listing objects and sorting lists

In addition to allowing single objects and collections of objects matching specific UUIDs to be returned, the GenericDAO also allows lists of objects of type T to be returned (to allow browsing of the entire collection of entities, for example). In many cases, applications will wish to restrict the total number of objects returned to a subset of the total available objects (to reduce resource requirements, or increase speed of rendering of a response, for example). This can be achieved by supplying non-null *`limit`* and *`start`* parameters to restrict the total number of objects returned. These parameters are analogous to the "limit" and "offset" parameters in SQL and are zero-based (i.e. the first result is 0, not 1).

Lists of objects are returned sorted according to the *`orderHints`* parameter. Like the *`propertyPaths`* parameter, `OrderHint` objects take a java-beans-style string that indicates the property or related entity that the list of returned objects should be ordered by, and a `SortOrder` that determined whether the list is sorted in ascending or descending order.

Chapter 5. Versioning

A significant use-case that the CDM aims to support is that of web-based or networked nomenclators, taxonomic treatments, and other applications that serve authoritative, dynamic data for (re-)use by taxonomists and other software applications. As an example, a CDM store containing a web-based monograph or revision of a particular plant or animal family might be referenced by other taxonomists, or other taxonomic databases that deal with the same taxa. To allow applications to record and resolve changes to data over time, for example, to allow users or client applications to determine how a taxonomic classification or species page has been altered since they last accessed that information, the CDM has a fine-grained versioning functionality that records changes to objects and their relationships, and allows the prior state of the dataset to be reconstructed.

The CDM uses hibernate-envers, a versioning / auditing library that is part of the hibernate core library. The versioning functionality is limited by the features that envers provides. Envers stores changes to entities on a per-transaction basis. Consequently, it is not possible to resolve changes that take place within the same transaction. Each transaction results in the creation of an `AuditEvent` object that provides metadata about the audit event and also allows the state of the database at that point to be reconstructed (because an `AuditEvent` represents a point in time across the entire database, rather than on a per-object basis). To learn more about envers and the way that it versions data, check out the presentation given by its creator, Adam Warski here.

Versioning is enabled by default, and calls to methods like `save`, `update`, and `delete`, will automatically result in data being versioned. Application developers only need to be aware of the existence of versioning when reading data, and only then if they wish to retrieve an object in its prior state. If applications wish to retrieve objects from the current state of the database, they do not need to perform any additional operations.

Because versions of objects are related to a global `AuditEvent`, and because applications may call several service layer methods when retrieving data for presentation in a particular view, the CDM stores the `AuditEvent` in the static field of an object called `AuditEventContextHolder`, allowing the CDM and any application code to discover which particular `AuditEvent` a view relates to without needing to pass the `AuditEvent` explicitly as a method parameter (this pattern is borrows from the `SecurityContext` class in Spring-Security).

To query the CDM at a particular `AuditEvent`, applications need to place the `AuditEvent` in to the `AuditEventContextHolder` and then call DAO methods as usual.

```
// This would retrieve the current version of the taxon with a matching uuid.
Taxon taxon = taxonDao.find(uuid);

// Set the audit event you're interested in
AuditEventContextHolder.getContext().setAuditEvent(auditEvent);

// This method call now retrieves the taxon with a matching uuid at the audit even
// or null if the taxon did not exist at that point.
Taxon taxon = taxonDao.find(uuid);

// Now clear the context
AuditEventContextHolder.clearContext();

// Further calls to the persistence layer will return the most recent objects
```

Not all DAO methods are available in non-current contexts, either because they require certain methods that Envers doesn't currently support (such as case-insensitive string comparison), or are across

relationships - currently envers does not support queries that place restrictions on related entities. In some cases this will be addressed in future releases of envers, and the CDM will incorporate these new releases as they occur. Some methods rely on the free-text-search functionality provided by hibernate search. Because hibernate search (and apache Lucene) are based on an optimized set of index files that reflect the current state of the database, it is not possible to search these indices at prior events. It is unlikely that the free-text-search functionality will ever be available in non-current contexts. If an application calls such a method in a non-current context, an `OperationNotSupportedInPriorViewException` is thrown, giving applications an operation to recover.

Objects retrieved in prior contexts can be initialized using the *`propertyPaths`* parameter, or (if the transaction is still open) by calling accessor methods in domain objects directly (just as you would with normal hibernate-managed entities).

In addition to being able to retrieve objects at a given state, the DAOs implement the `IVersionableDao` interface that offers five specific methods for working with versioned objects.

Table 5.1. `IVersionableDao` methods

Method	Description
`List<AuditEventRecord<T>> getAuditEvents(T t, Integer limit, Integer start, AuditEventSort sort, List<String> propertyPaths);`	Returns a list of audit events (in order) which affected the state of entity t. The events returned start at the `AuditEventContext` and go forward in time (*`AuditEventSort.FORWARDS`*) or backwards in time (*`AuditEventSort.BACKWARDS`*). If the `AuditEventContext` is set to null, or to *`AuditEvent.CURRENT_VIEW`*, then all relevant AuditEvents are returned.
`int countAuditEvents(T t, AuditEventSort sort);`	Returns a count of audit events which affected the state of entity t.
`AuditEventRecord<T> getNextAuditEvent(T t);`	Convenience method which returns a record of the next (relative to the audit event in context).
`AuditEventRecord<T> getPreviousAuditEvent(T t);`	Convenience method which returns a record of the previous (relative to the audit event in context).
`boolean existed(UUID uuid);`	Returns true if an object with uuid matching the one supplied either currently exists, or existed previously and has been deleted from the current view.

Chapter 6. Free Text Search

The CDM supports high-performance free-text ("google-like") searching of the data that it stores. It uses the hibernate-search library to integrate the popular apache Lucene search software into the CDM. The persistence layer includes hibernate-search integration by default, so objects are added to the lucene index when applications `save` entities, and the indices are updated when applications `update` or `delete` objects. All fields are converted to lowercase during indexing, and queries are converted to lowercase during parsing. Several properties are indexed per object type, and it is possible to search individual fields or combinations of fields. The basic syntax used for free text queries is described on the lucene website.

All classes have a default field that is searched when a field is not specified. In the case of classes that extend `IdentifiableEntity` the *`titleCache`* field is used. By default, query strings are broken into individual terms and objects are returned that match any of the terms (e.g. *Acherontia atropos*). To return objects that match all terms, in any order, the an AND operator can be used (e.g. *Acherontia AND atropos*). By enclosing individual terms in double quotes, you can specify that terms must appear in a certain order (e.g. *"Acherontia atropos"*).

To search a specific property, prepend the name of the property, followed by a colon to the query (e.g. *nameCache:"Acherontia atropos"*). Properties of related entities can be searched too, provided that they have been indexed, using java-beans-like dot-notation. For example, to return all references written by Schott you could use *authorTeam.titleCache:Schott*, and to return all publications written in the 1940's you could use either *datePublished.start:194** or *datePublished.start:[1940* TO 1949*]* (to specify a range).

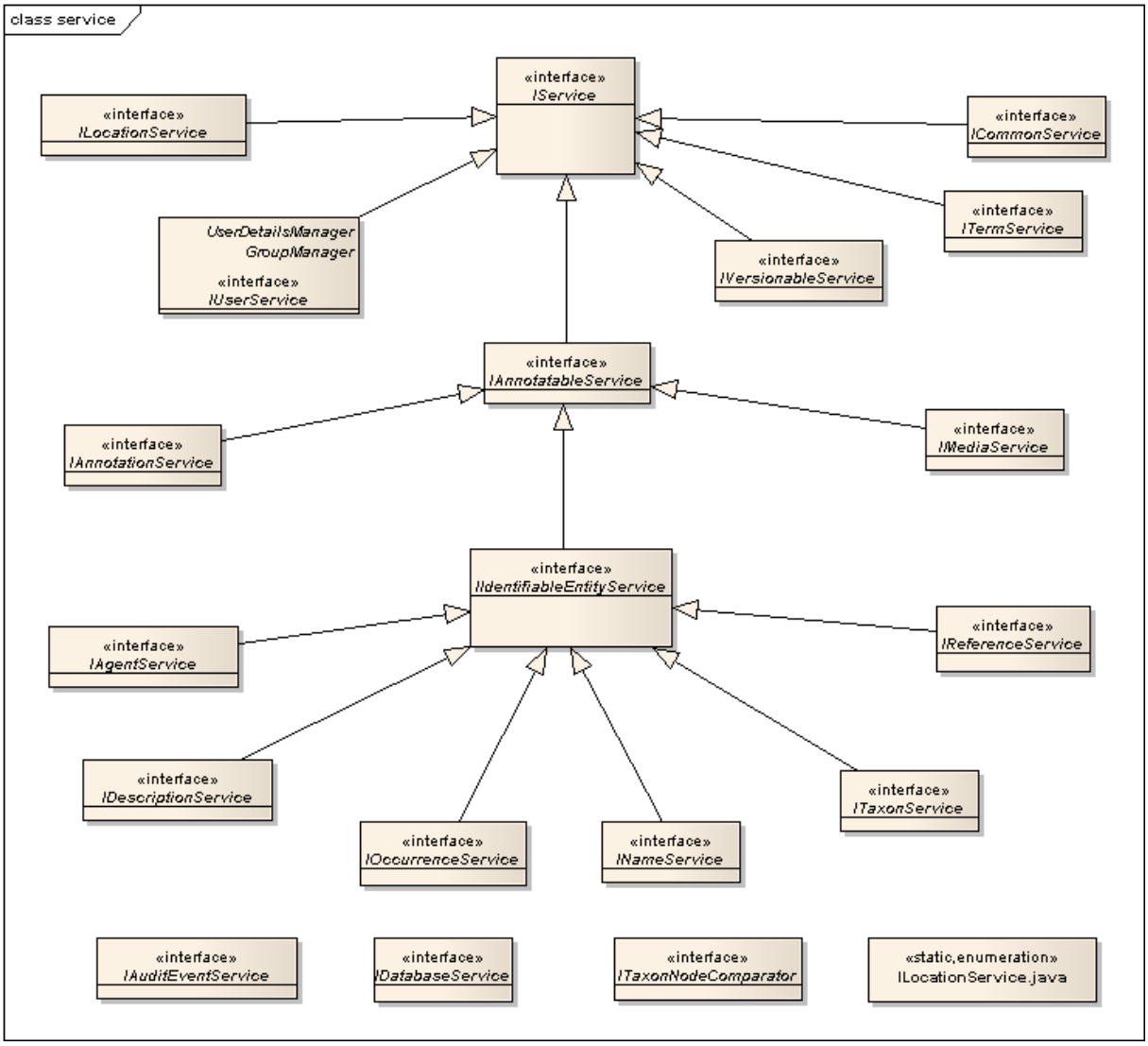
Part IV. API Methods

Apart from the Common Data Model classes themselves, the CDM Service layer contains the components most likely to be used directly by applications based upon the CDM Java Library. This layer contains a set of basic service objects that can be used as a facade over the persistence logic.

Chapter 7. Services

The service layer of the CDM contains a set of service objects that are intended to provide basic query, search and persistence functionality for the CDM objects, plus business logic to support common tasks. These objects are intended to be singleton services used across the whole application. As with the persistence layer, the services are strongly typed, generic service objects, with a single service per (significant base) class. All service classes implement `IService` and most implement `IVersionableService`, providing access to generic base methods to deal with the class.

Figure 7.1. An overview of the cdm service layer



The Service layer in the CDM Java Library. There is a service for each major type of data that the CDM deals with.

Table 7.1. `IService` methods

Method	Description
`UUID saveOrUpdate(T newOrTransientEntity);`	Makes a new object persistent, or persists the state of a transient object.
`Map<UUID,T> save(Collection<T> newInstances);`	Makes a collection of new objects persistent.
`UUID save(T newEntity);`	Makes a new object persistent.
`UUID update(T newEntity);`	Makes changes to a transient object persistent.
`UUID merge(T newEntity);`	Merges the state of a detached object into the persisted version.
`UUID delete(T persistentEntity);`	Deletes a persistent object.
`List<T> list(Class<? extends T> clazz, Integer pageSize, Integer pageNumber, List<OrderHint> orderHints, List<String> propertyPaths);`	Returns a (sub-)list of objects matching the type *`clazz`*, sorted according to the order hints and initialized according to the propertyPaths.
`Pager<T> page(Class<? extends T> clazz, Integer pageSize, Integer pageNumber, List<OrderHint> orderHints, List<String> propertyPaths);`	Returns a paged (sub-)list of objects matching the type *`clazz`*, sorted according to the order hints and initialized according to the propertyPaths.
`int count(Class<? extends T> clazz);`	Returns a count of objects matching the type *`clazz`*.
`T find(UUID uuid);`	Returns an object of type T matching the supplied uuid if it exists.
`Collection<T> find(Collection<UUID> uuids);`	Returns a collection of objects of type T matching the uuids supplied, if they exist.
`T load(UUID uuid, Collection<String> propertyPaths);`	Returns an object of type T with properties initialized according to the rules described below.
`Set<T> load(Collection<UUID> uuids, Collection<String> propertyPaths);`	Returns a collection of objects of type T matching the uuids supplied, if they exist, initialized according to the rules described below.
`boolean exists(UUID uuid);`	Returns true if there is an object of type T in the database matching the supplied uuid.
`Class<T> getType();`	Returns the class of objects that this Service provides access to.

Table 7.2. `IVersionableService` methods

Method	Description
`Pager<AuditEventRecord<T>> pageAuditEvents(T versionableEntity, Integer pageSize, Integer pageNumber, AuditEventSort sort);`	Makes a new object persistent, or persists the state of a transient object.
`AuditEvent getNextAuditEvent(T t);`	Makes a collection of new objects persistent.
`AuditEvent getPreviousAuditEvent(T t);`	Makes a new object persistent.
`boolean existed(UUID uuid);`	Makes a new object persistent.

Paging Resultsets

In addition to being able to return results as a `java.util.List`, service layer methods can return results as a `Pager`. Pagers contain a sublist of the total result set, plus a count of the total number of matching objects. In addition, they contain a number of convenience methods to facilitate the rendering of paged resultsets, including the generation of labels for pages, based upon the matching objects.

Chapter 8. Globally Unique Identifier Resolution

Likely To Change

Globally unique identifiers, their use and implementation are still an unresolved topic. The implementaton of objects that are identifiable and resolvable in a global sense must, given an open world, reflect the standards and best practices being used by the community as a whole. Consequently this area of the CDM is likely to change to reflect this.

The service layer implements a number of services designed to serve as the basis of a LSID Resolution Service. This includes implementations of `LSIDAuthorityService`, `LSIDMetadataService`, and `LSIDDataService`. Note that these are service-layer implementations - the http-specific components can be found in the cdmlib-remote package.

In addition to implementations of the three core LSID Resolution services, the service layer holds the `LSIDRegistry`, the component that maps LSID authority + namespace combinations onto CDM classes. The implementation assumes that a given authority and namespace will map onto a single CDM base class, but that authorities may use different namespaces for the same class of objects. In addition, the `LSIDRegistry` provides a way of controlling which authority + namespace combinations a CDM application will respond to. For example, it is possible that an application will store objects with identifiers published by another (foreign) authority, but doesn't wish to serve metadata about these objects. By only registering specific authority + namespace combinations in the `LSIDRegistry`, a CDM store can resolve some combinations but not others.

The three most common methods used are the `getAuthorityWSDL` and `getAvailableServices` methods that return a `javax.xml.transform.Source` within an `ExpiringResponse` object suitable for rendering in a response to a client, and `getMetadata`, that returns an `IIdentifiableEntity` within a `MetadataResponse`. If the authority+namespace is not resolved, or if the object cannot be resolved, or if the client requests metadata in an unavailable format, an exception is thrown.

Chapter 9. Security and Identity within the CDM Library

The CDM Library uses the Spring Security sub-project as the basis of its security implementation. The best place to get information on using Spring Security is the project website.

Spring Security is based around a non-intrusive and non-invasive architecture that can be configured as needed by a particular application. The CDM Java Library does not have any restricted or protected methods by default - it is likely that each application based on the CDM will wish to protect services in a different way. The CDM service layer does provide a number of classes that make it straightforward to set up.

In addition to providing generic components for authentication and authorization, Spring Security provides a number of components that can be used by web applications. Details on authentication and authorization concepts applied to web applications can be found in the documentation for the cdmlib-remote package.

Identity

Identity in Spring Security is based around the `UserDetails` interface, that provides access to the principal's username, password, granted authorities and other details. The CDM provides the `User` class that implements this interface. In addition, it provides implementations of the `GrantedAuthority` and a `Group` class to allow group authorities (permissions that belong to a group of individuals rather than belonging to a single `User`). Creation of new user accounts, manipulation of account details, permissions, and group membership is achieved through an implementation of `IUserService` provided by the library.

The CDM provides some basic auditing functionality by storing the user account and timestamp each time an object is modified (and a transaction is comitted). The user details are retrieved from the `SecurityContextHolder` provided by Spring Security. If authentication is set up (see below) and the user is logged in, then this data will be present automatically in the `SecurityContext`. In the case of applications that do not use Spring Security, the `User` object must be placed into the `SecurityContext` explicitly for the user details to be recorded in this way.

Authentication

To enable authentication within your application, a small number of additional beans need to be added to the application context, thus (note the use of the *security* spring-security namespace):

```
<security:authentication-manager alias="authenticationManager"/>

<bean id="daoAuthenticationProvider" class="org.springframework.security.providers
  <security:custom-authentication-provider/>
  <property name="userDetailsService" ref="userService"/>
  <property name="saltSource" ref="saltSource"/>
  <property name="passwordEncoder" ref="passwordEncoder"/>
</bean>

<bean id="passwordEncoder" class="org.springframework.security.providers.encoding.

<bean id="saltSource" class="org.springframework.security.providers.dao.salt.Refle
  <property name="userPropertyToUse" value="getUsername"/>
```

```
</bean>
```

In the case of web applications, application developers will probably want to authenticate users transparently, using the servlet filter provided by spring security. For desktop applications, you can also authenticate a user programatically:

```
UsernamePasswordAuthenticationToken token = new UsernamePasswordAuthenticationToke
authenticationManager.authenticate(token);
```

Authorization

As with authentication, web applications based upon the CDM may find the standard methods provided by Spring Security or protecting URLs to be sufficient in most cases. To protect service methods, or to secure desktop applications, developers can also use global method security by specifying a pointcut expression that matches the service and method that they wish to protect, and a granted authority that is allowed to access the method thus:

```
<security:global-method-security>
  <security:protect-pointcut expression="execution(* eu.etaxonomy.cdm.api.service.
</security:global-method-security>
```

Part V. CDM Input / Output Layer

This part describes the input output routines:

Part VI. CDM Server

This part describes the cdm-server application: