DIGIT Document Repository

Codename: REPO

DIGIT Document Repository: Codename: REPO

Table of Contents

List of Tables

List of Examples

Chapter 1. Architecture

1. Introduction

This chapter describes the philosophy, decisions, constraints, justifications, significant elements, and any other overarching aspects of the system that shape the design and implementation.

2. Architectural goals and philosophy

The main goal of this project is to provide CMIS compliant services.

To reduce the time to provide the services to third parties, a roadmap was designed to incrementally add new funtionalities:

Roadmap

Release 1.0.0-BETA1	• Complete CMIS Repository services
	• Complete CMIS Repository services
	• Complete CMIS Object services
	• Complete CMIS Navigation services
	• Basic security funtionalities
Release 1.0.0-BETA2	• Complete CMIS Discovery services (properties and fulltext search)
Release 1.0.0-BETA3	• Complete CMIS Versioning services
	• Complete CMIS Relationship services
	• Complete security funtionalities
Release 1.0.0	• Adminitration panel to manage repositories
	• Complete CMIS Renditions services
	• Complete CMIS Multifiling services
	• Complete CMIS change log
Release 1.1.0	• Implement CMIS 1.1 services

3. Assumptions and dependencies

The target enviroment is Java 7. Currently using Tomcat 7 but the project is not using any non-standard funtionality of the servlet container. This means could be replaced in the future with no overhead.

The target Database is Oracle for production and H2 for development and test. As the Entity classes have sequences configured and we have some SQL statetents in the indexing part, we are partially couple to a DB with sequences. In the future we could evaluate other databases like MySQL but this is not in the roadmap for version 1.0.0.

Tools

Tools used during the development that ara not part of the final product.

- Maven (Build automation)
- Jenkins (Continuous inetgration tool)
- Jira (Issue tracker)
- Git (Source control)
- Ant (Autodeployment process)
- Sonar (Quality check)
- JMeter (Stress tests)
- JUnit (Unit tests)
- DBUnit (Unit tests of the persistence layer)
- Docbook (Documentation)

Frameworks and libraries

Components that are part of the product, these are direct dependencies of the package.

- Spring (IoC, MVC, Security, ORM, etc.)
- JPA2 (Hibernate), JDBC
- Apache Chemistry OpenCMIS (CMIS client and server framework)
- Apache Tiles (Used in the administration panel)
- SLF4J (Logging)
- H2 (Embedded database engine, used for the tests and indexing)
- Apache Commons Configuration
- ANTLR (Parsing the CMIS queries)
- Lucene (Indexing)
- Tika (Extracting textual content from documents)

- JMX (MBeans exposed with internal statistics)
- jQuery
- Twitter bootstrap

Third parties systems and components

Systems and components that are not part of the deployable package.

- Oracle Database (Main persistence system)

4. Architecturally significant requirements

Funtional requirements

The project should satisfy the following funtional requirements:

- The system should be able to provide CMIS services
- Selfcontained, be able to be used with no external dependencies
- A server instance may host more than one repository
- Implement the capabilities configurable by repository
- Cover the code with unit testing.

Hardware requirements

See below the list of requirements related to the hardware infrastructure:

- Brief description of the application purpose:

  CMIS 1.1 repository server, this will serve as document repository based on the standards http://en.wikipedia.org/wiki/Content_Management_Interoperability_Services.

- Specific availability requirements (if any), e.g. if the IS is to be accessible worldwide, 24/7/365 availability is needed and any technical intervention must be kept to a bare minimum:

  Initially the system will be accessible from the EC network and should be accessible 24/7 (this service will be used by other applications).

- Local file system needs (if any) and approximative size and growth rate:

  Minimum usage of file system, only cached and temporary files will be created. A provision of 10Gb could be enough. The information will be stored in the database, in which could be at the Sherlock level (200Gb)

- Memory requirements,

  We haven't done memory requirement assessments, standard memory configuration could be applied until stress test will be done.

- 64 bits application or not (for Tomcat parametrisation),

  Java 7 64 bits and Tomcat 7 64bit. Classpath added to `/data/applications/repo/conf`

- HTTPD server as a front-end (if needed),

  Yes

- Dependency on another information system (shared database or service, Trade internal or central (e.g. PDFCode)),

  None

- Authentication mechanism (using various Authentication Handlers) or public access,

- Incoming or outgoing mail,

  No

General information

Application name	Trade document repository
Acronym	REPO
Short description	CMIS 1.1 compliant document repository.
Maximum acceptable unavailability	24 hours
Priority	high
DRP ready	yes
Performance email	no
Url to page with all links of application	
Sensitive data	yes - Limited High
IT Classification	Specific
Authentication	customizable
DB type	Oracle 11
Technologies used	Java 7 64bits; Tomcat 7;
Filesystem used	10GB
Database size	200GB
Memory requirements	Default memory configuration
HTTPD server	required

5. Decisions, constraints, and justifications

There are several constraits

- Use standards
- Be vendor independent
- Use existing components, and adapt them base on our needs

5.1. Relational database

The decision to use relational databases was taken because several factors:

Transactions

With a relational database system, we don't need to implement a transaction layer. All important database providers implements transactions, and this helps us to focus more on the business and less in the infrastructure.

Concurrency

within an alternative storage like a file system we could experiment issues with concurrent users (e.g. locking of files). The database implements a concurrency control out-of-the-box.

Consistency

with referential integrity all the relations between the different parts are managed at the database level. The data consistency is improved and the complexity of maintenance reduced.

Known technology

Databases are a known and reliable technology

Simplicity

Using only one persistence system will reduce the complexity. For queries paginated and sorted would be very complex and bad performance to query two systems and merge the results.

5.2. Persist index data in relational database

The decision to implement our own persistence implementation of indexing system was made based on the following reasons:

Vendor independence

Despite we could use vendor specific solutions, was decided not to use them to have a solution more portable and independent.

This is very convenient for the tests. We could easily change from from a database product to other one.

Easy integrability with other components

Persisting the index information in the database is giving us the oportunity to integrate with the other components easily.

To implement the discovery services it's needed to combine filters using both, the object metadata and the index data, keeping both information in the same system reduces the complexity and improves the performance.

Performance

A common issue of combining information from several sources is the pagination and sorting.

For example, If the end user asks the documents that contains the word China and the author 'Alice':

Example 1.1. CMIS query

```
select * from cmis:document where contains('China') and cmis:createdBy = 'Alice'
```

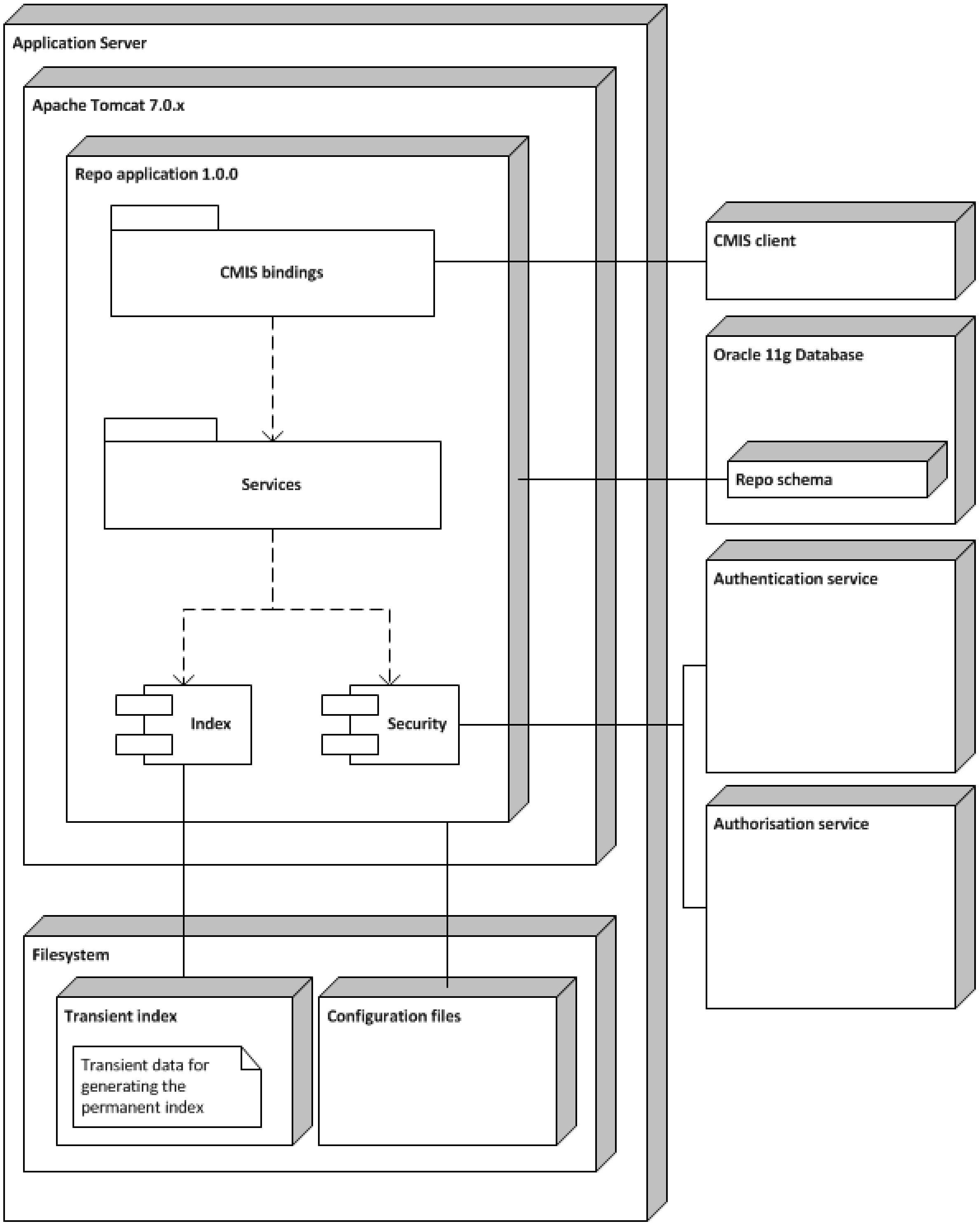
The system needs to search in two subsystems: the metadata subsystem for the author and the fulltext search subsystem to filter by the word.

As the user is only interesed on a page of the results, we could not filter the results in the subsystems. So this ends up with a query on the fulltext search subsystem with all the documents with the word 'China'.

Keeping both subsystems together eliminates this issue and make the queries using less resources and more scalable.

6. Key abstractions

6.1. Deployment diagram



A deployment diagram in the Unified Modeling Language models the physical deployment of artifacts on nodes.

The following diagram was intentionally simplied in terms of internal components, as the goal of the model is to represent the interactions with external services.

The CMIS client is connected to the application and profiting the CMIS bindings, is executing the services.

The application is persisting most of the data in a relational database.

The security component is dealing with an authentication service and an authorisation service. For simplicity in the diagram is only represented one security component, but could be divided into two different scopes: CMIS security related and web application security related.

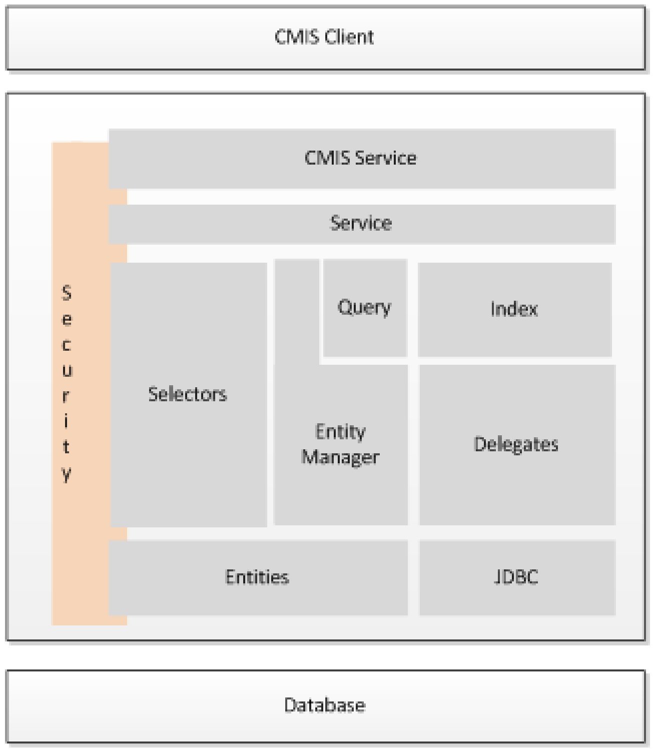
For accessing the administration panel of the repositories, the application is using an authentication and authorisation service. Only to point out that both parts of the security are independent and could be configured with different providers.

The index component is saving temporary information in the filesystem. An embeded database is storing the words of the documents to later process and build the permanent index data.

The configuration files are under the filesystem, Tomcat is able to read these files because the path was added in the classpath.

7. Layers or architectural framework

The application could be divided in several layers: database, persistence, service and CMIS layer; but there are other components as well that could interact at several point on the system: Security, Index and Query.



In the previous diagram, there are there main subsystems: the database, the application itself and the CMIS client.

The database it's an external layer that stores all the information of the repository. We kept this component vendor independent using JPA2 and JDBC. Currently we are using Oracle Database and H2.

The application subsystem is comunicating with the Database with JPA2 Entities and JDBC. On top of the Entities we have an instance of `javax.persistence.EntityManager` for the CRUD operations, and selectors for the queries.

For retriving and storing the files into the database and the indexing process, JDBC is used instead JPA2. The Delegates have the reponsability to group the JDBC code.

The service layer is the responsible of implementing all the interactions with the repository using Entities instances. Up to this point there's no dependency on Apache Chemistry OpenCMIS.

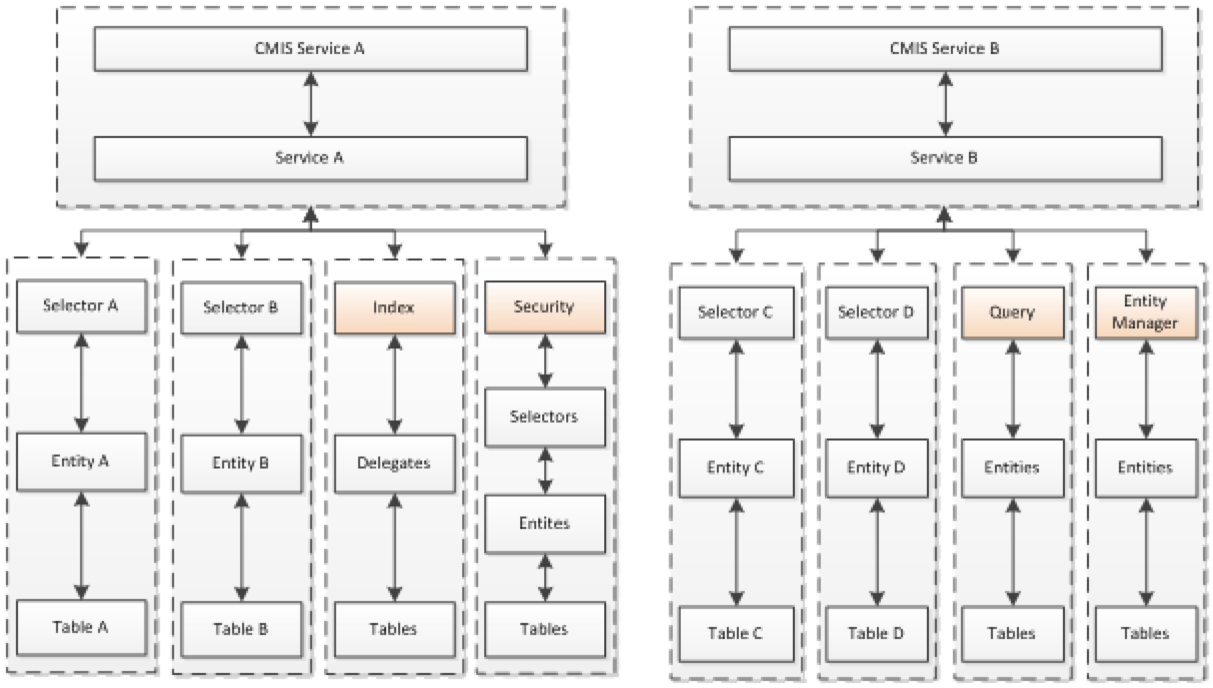
Note

The repository service layer is independent of Apache Chemistry OpenCMIS, but we are using the enumerations from them. Was decided to use the enumeration for avoid duplication. In the improvable case we will implement the server with another framework we could decide to keep the enumerations from Apache Chemistry OpenCMIS of have a local copy.

The CMIS service layer is transforming the data coming from the client into our model. Once the data is translated, a method in the service layer is used.

From bottom to top:

Database	Stores all the information of the repostitory.
Entities	JPA2 Entities mapped to database tables.
Entity Manager	Managing the CRUD operations of the Entities
Selectors	Implementing the entities queries.
Delegates	Contains the code to interact with the Database that is not covered by selectors and Entity Manager.
Query	Implements the parsing and execution of CMIS queries.
Index	Implements the indexing of the files.
Services	Contains the internal logic of the repository.
CMIS Services	This layer transforms from Apache Chemistry OpenCMIS Interfaces to our Entities data structure.
Security	This is an horizontal component, manages the security.
CMIS client	Any application of framework that implements CMIS standard.



7.1. Database

The database design it's following the next principles:

- Try to define as much constraints into the database definition. If a column only allows 'T' or 'F' create a check constraint to apply this rule.
- The delete should run on cascade, this means deleting a row in a table will delete all related data. The main reason is to keep the data consistency.

- Use a single numeric column for primary keys, populated by sequences. This makes the JPA and relational maintenance easier and cleaner.
- All the primary keys are populated by a sequence.
- Several tables includes cmis_id or cmis_object_id, these columns with prefix cmis_ stores the CMIS (client point of view) ids, like `cmis:folder` or `cmis:name`.
- All Objects in the repository have a property called ObjectId that saves an opaque unique key. This information is saved in two places, into the regular property structure and in the object table. This is for checking the uniqueness and for faster query searches.

7.2. Persistence Layer

The persistence layer relies on the database and it's using JPA and JDBC technologies like Hibernate or Spring JDBC Template.

- Every Table in the database has their correspondence JPA entity (except many to many relationship tables).
- Every Entity extends `eu.trade.repo.model.DBEntity`.
- All entities relationships will be defines as `LAZY` explicitly and when needed a query with join fetch with be created.
- All the JPA queries will be named queries defined in the entity with annotations. See the section below for the naming contentions.

7.2.1. Selectors

The selectors are responsible to retrieve information from the database. The `EntityManager` could retrieve the objects using the primary key, but most of the time the input parameters are different.

There is a selector class per Entity and all the methods in the selector should return instances of this Entity.

> **Note**
> All the queries executed must be named queries, there is a proxy wrapping the `EntityManager` that will throw an exception if you try to use non named queries. See the next section for details about the naming conventions of named queries.

By default the collections of the Entities are configured as `LAZY`, so the queries used in the selectors have to explicitily indicate `join fetch` when needed.

There is a second level cache configured for caching the Objects and the queries. Most of the Entities that are prone to be static, like `Repository` or `ObjectType` are cached, so don't use `join fetch` with these entities.

Verify the files `ehcache.xml` and `persistence.xml` to check what Entities are cached and how. The `EntityManager` proxy is enabling the query cache if the named query prefix corresponds to a classname in the file `ehcache.xml`.

7.2.2. Named queries conventions

The entity queries are named as:

- `*EntityName*.by_*Attribute*` for 'dry' entities, all props lazy loaded eg. `permission.by_name` for the query to retrieve a Permission obj. with all associations lazily loaded.

- `*EntityName*.with_*Association*` for entities with some associations resolved. Eg. `permission.with_parent` for a `Permission` object with the parent association resolved/fetched.
- `*EntityName*.with_dependencies` for a fetch with all associations resolved/fetched.

Similarly, the selector offers methods:

- `get*EntityName*` for an entity with no associations resolved. Eg. `getPermission`
- `get*EntityName*With*Association*` for an entity with some associations resolved. Eg. `getPermissionWithParent`
- `load*EntityName*` for an entity with all assocs resolved. Eg. `loadPermission`

7.3. Delegates

The delegates are responsible to implement the interactions with the database that are not covered by the Entity Manager and the selectors.

For example, the delegates implement the stream manipulation not covered by JPA.

7.4. Service layer

The service layer is the core of the application, all the business logic is implemented at this level.

There is a class per CMIS service type, and by design there is no interlinking between services. A service class could use selectors, delegates and components but never will use another service.

7.5. CMIS layer

This layer is connecting the Service layer and the Apache Chemistry Open CMIS bindings. By design this later should be as minimal as possible, it translates the data coming from Chemistry bindings to our model Entities and call the related service.

There is a class per CMIS service type and several helper classes to transform from and to our model classes.

7.6. Package distribution

`eu.trade.repo`

Contains the service class `eu.trade.repo.RepoService` that aggregates of all the services provided by the server. The service layer was divided into classes and then combined into the previous class with a proxy pattern.

This package also contains the service factory `eu.trade.repo.RepoServiceFactory`, that with `eu.trade.repo.web.CmisLifecycleBean` both are the the entry point of the Apache Chemistry OpenCMIS server bindings.

`eu.trade.repo.delegates`

All the delegate classes are under this package.

A delegate is reponisble of altering the content of the database and retriving data with JDBC. Currently there are delegates for managing the persist and retrival of documents in the relational database, and the index generation.

`eu.trade.repo.index`

You could find in this package and subpackages all the classes related with the indexing process.

For further details, read the chapter of Indexing.

`eu.trade.repo.mbean`

The application is exposing throw JMX valuable information about the usage of the server. This package includes the Mbeans with these values, for example call count, total time, last time, average in 5,10 and 15 minutes, etc.

To see the complete list of Mbeans read the section JMX of this chapter.

`eu.trade.repo.model`

This package contains the JPA2 entities. All entities extends the class `eu.trade.repo.model.DBEntitity`.

Each entity may have named queries defined, please check the naming convention in the persistence layer section.

Entities

`eu.trade.repo.model.Acl`

`eu.trade.repo.model.CMISObject`

`eu.trade.repo.model.DBEntity`

`eu.trade.repo.model.ObjectType`

`eu.trade.repo.model.ObjectTypeProperty`

`eu.trade.repo.model.ObjectTypeRelationship`

`eu.trade.repo.model.Permission`

`eu.trade.repo.model.PermissionMapping`

`eu.trade.repo.model.Property`

`eu.trade.repo.model.Rendition`

`eu.trade.repo.model.Repository`

`eu.trade.repo.model.SecurityHandler`

`eu.trade.repo.model.Word`

`eu.trade.repo.model.WordObject`

`eu.trade.repo.model.WordPosition`

`eu.trade.repo.model.WordPositionId`

Enums

`eu.trade.repo.model.ActionParameter`

`eu.trade.repo.model.HandlerType`

`eu.trade.repo.model.IndexingState`

`eu.trade.repo.model.SecurityType`

`eu.trade.repo.query`

Contains the Query component, that includes the parsing of the CMIS query and the JPA query generation based on the parsed information.

The CMIS query is parsed using ANTLR 3, we've created a grammar file that generates the tree structure, AST.

The AST is traversed and generates a JPA query with the interface `javax.persistence.criteria.CriteriaBuilder` from JPA.

`eu.trade.repo.security`

`eu.trade.repo.selectors`

All the selector are under this package

`eu.trade.repo.service`

Contains the main business logic of the application. Could be divided in several subpackages:

`eu.trade.repo.service` This package is the core of the application, this receives and returns Entity instances. It's designed for not to have any dependency with Apache Chemistry OpenCMIS.
`eu.trade.repo.service.cmis` This package uses the previous package and translates the data coming from the CMIS client to our model classes (JPA Entities). By design, this package should not contain logic.
`eu.trade.repo.service.cmis.data.in` This package is resposible to transform the data structures from the Apache Chemistry OpenCMIS into our model classes.
`eu.trade.repo.service.cmis.data.out` This package is resposible to transform the data our model data structures to the Apache Chemistry OpenCMIS interfaces.
`eu.trade.repo.service.interfaces` Interfaces for our services. the implementation of these interfaces are under the package `eu.trade.repo.service`.

`eu.trade.repo.stats`

`eu.trade.repo.util`

Utility and miscellane package.

`eu.trade.repo.web`

The administration panel an the ECAS ticket generator code are under this package.

7.7. Server statistics

JMX

7.8. Logging

8. Architectural views

To be updated in future release.

Chapter 2. Data model

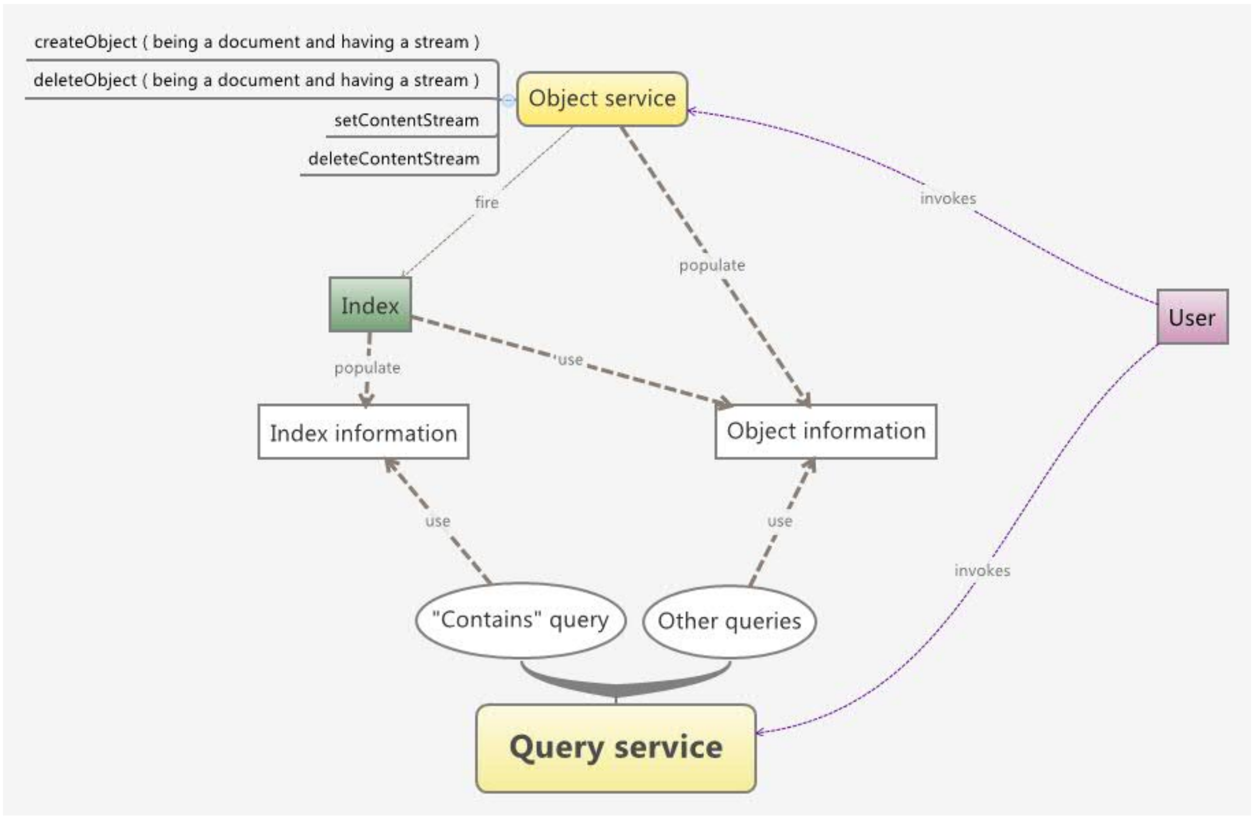
Data model diagram and brief explanation of the tables and columns.

To be updated further in future release.

Chapter 3. Indexing Process

1. Context

The index is an internal CMIS Server functionality. It is a piece of software at the service of the CMIS query service and it is responsible for populating the data structure needed by the query service to provide full text search in documents.



2. Introduction

2.1. What is it

It is a part of the full text search system, responsible for preparing the information in a way as to be retrieved as fast as possible by the subsequent search.

The way it prepares information is a reverse index (see section 'Reverse index'), in a similar way as Lucen does (see section 'Reciprocity between Repo index and Lucene')

2.2. Requirements / Restrictions

- The unavailability or failure of the index functionality should not affect the rest of CMIS server processes (beyond the document at hand would not be indexed and would not be found as result of a 'contains' query)
- Multi-lingual documents can be indexed
- The generated index should be stored in the database to make it more powerful the searching process (pagination, ordering, ...)

2.3. Integration

1. ATOMIC INDEX:
   - Launched for every operation where the stream or metadata of a cmis:object are updated
   - Uses transaction synchronization to avoid interfering with the rest of CMIS processes
2. BACKGROUND JOBS:
   - Jobs scheduled to be launched at certain configured moments
   - Jobs that launch index operations:
     - Clean orphan indexes job: delete all remaining information related to deleted cmis:objects
     - Re-index content/metadata jobs: launch the index process for cmis:objects whose indexes are in error or not done states
   - Other jobs related to the index:
     - Compact transient database job: it aims to improve resources usage. It closes and compacts internal index database at certain configured moments if there are no index processes running.

3. Reverse Index

The index stores words and statistics in order to make word-based search more efficient. Repo's index, as Lucene's, falls into the family of indexes known as inverted index. This means that given a word, the index can list the documents that contains it. This is the inverse of the natural relationship, in wich documents list words.

3.1. Definitions

The fundamental concepts in Repo's index are index, indexable objects, content-index, metadata-index, indexable property, word, word-object, word-position and dictionary.

An index contains a series of indexable objects

- An indexable object is composed by a metadata-index and, optionally, by a content-index.
- A content-index is a set of words (the words contained in a cmis:object related stream)
- A metadata-index is a set of indexable properties ('string' cmis:properties related to the indexable object)
- An indexable property is a set of words (the words contained in the cmis:property's value)
- A word is a string
- A word-object is a word in relation with an indexable object and, optionally, with an indexable property. It contains frequency information.
- A word-position is the position of each occurrence of every word that make up an indexable object.
- A dictionary is a set of words

Each cmis:repository has its own dictionary.

The same string in two different dictionaries is considered as two different words.

The same word in the content of an indexable object or in two different metadata-properties is considered a different word-object.

3.2. Repo index structure

The structure of the Repo index is composed by 3 database tables and 4 fields about indexing state in the cmis:object table.

- INDEX_WORD: contains the words and dictionaries.
- INDEX_WORD_OBJECT: contains word-objects together with information about word's frequency in the indexable object.
- INDEX_WORD_POSITION: contains word-positions ordered within each word-object.

3.3. Reciprocity between Repo index and Lucene

Table 3.1. Reciprocity Repo - Lucene

Lucene	Lucene definition	Repo
Term	A term is a string. The term dictionary registers information about	Word, Word-object, Word-position

Lucene	**Lucene definition**	**Repo**
	term's frequency (FreqDelta) and the term's position (ProxDelta)	
Field	A field is a named sequence of fields	Content, Metadata-property
Document	A document is a sequence of fields.	Indexable object, Metadata

4. Index components

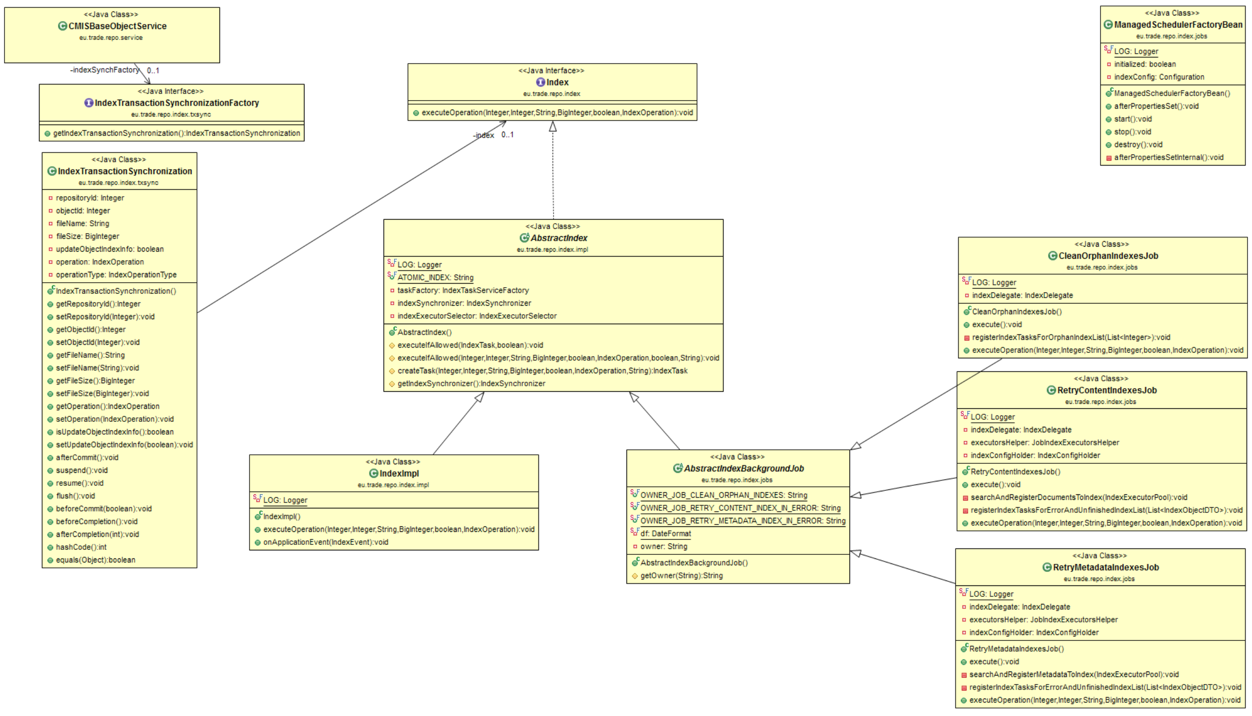
4.1. Index entry points

The entry points to the index functionality are implementations of the Index interface, specifically subclasses of AbstractIndex: IndexImpl and subclasses of AbstractIndexBackgroundJob.

IndexImpl is invoked by the CMIS object service to perform the indexing of single cmis:objects responding to user requests to the CMIS server (atomic index).

Subclasses of AbstractIndexBackgroundJob are processes triggered by the system according to crontab-expressions, that look for orphan or unfinished indexes and perform the convenient index operation for each of them.

The index entry points create IndexTasks with the needed information and pass them to the propper task executor (using IndexExecutorSelector), which in turn will use a ThreadPool to execute the operations specified by IndexTasks. To coordinate the case of having multiple IndexTasks related to the same cmis object, the entry points will use the IndexSynchronizer (thread save).

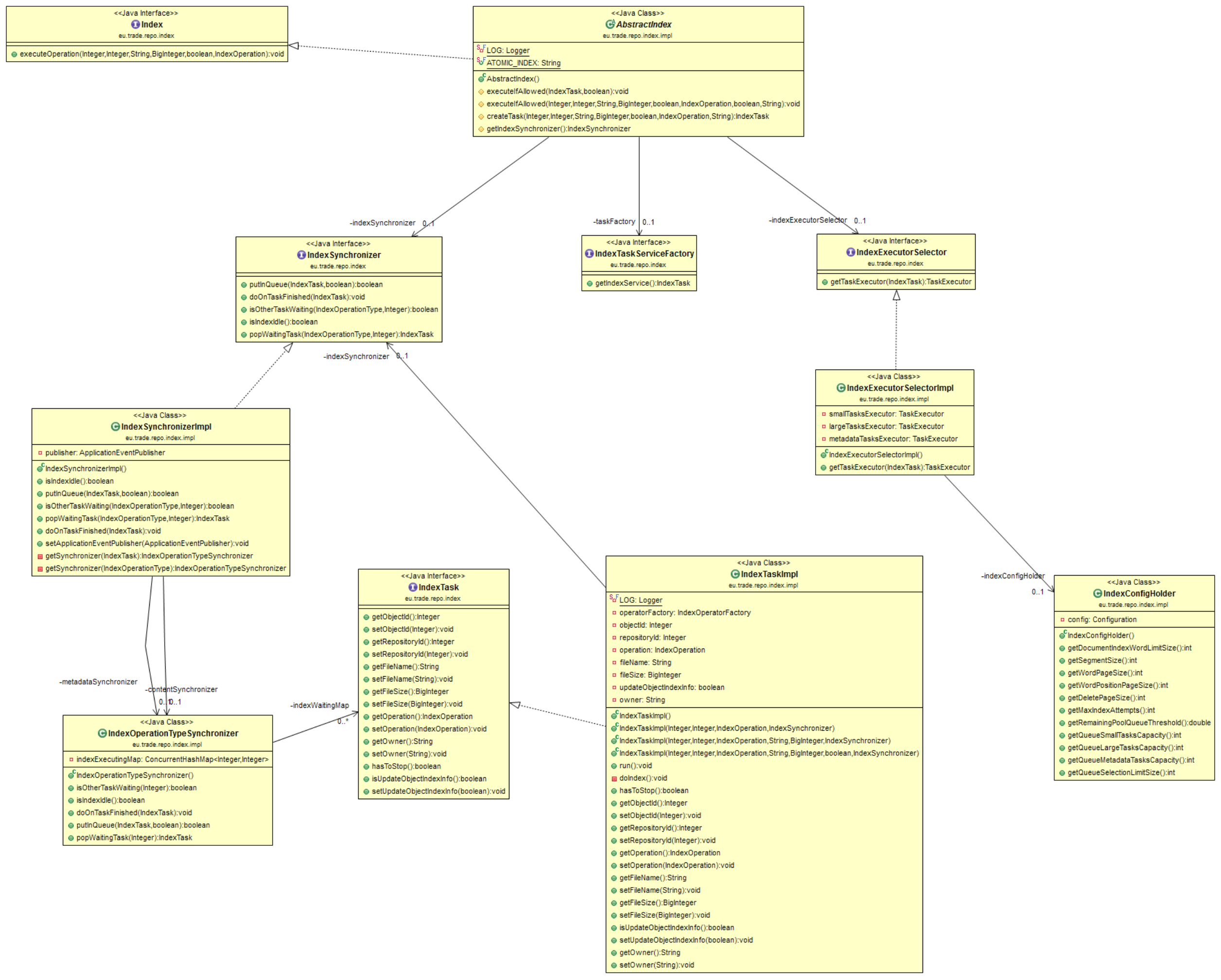

<<Java Class>>
CMISBaseObjectService
eu.trade.repo.service
-indexSynchFactory 0..1
<<Java Interface>>
IndexTransactionSynchronizationFactory
eu.trade.repo.index.txsync
getIndexTransactionSynchronization():IndexTransactionSynchronization
<<Java Interface>>
Index
eu.trade.repo.index
executeOperation(Integer,Integer,String,BigInteger,boolean,IndexOperation):void
-index 0..1
<<Java Class>>
ManagedSchedulerFactoryBean
eu.trade.repo.index.jobs
LOG: Logger
initialized: boolean
indexConfig: Configuration
ManagedSchedulerFactoryBean()
afterPropertiesSet():void
start():void
stop():void
destroy():void
afterPropertiesSetInternal():void
<<Java Class>>
IndexTransactionSynchronization
eu.trade.repo.index.txsync
repositoryId: Integer
objectId: Integer
fileName: String
fileSize: BigInteger
updateObjectIndexInfo: boolean
operation: IndexOperation
operationType: IndexOperationType
IndexTransactionSynchronization()
getRepositoryId():Integer
setRepositoryId(Integer):void
getObjectId():Integer
setObjectId(Integer):void
getFileName():String
setFileName(String):void
getFileSize():BigInteger
setFileSize(BigInteger):void
getOperation():IndexOperation
setOperation(IndexOperation):void
isUpdateObjectIndexInfo():boolean
setUpdateObjectIndexInfo(boolean):void
afterCommit():void
suspend():void
resume():void
flush():void
beforeCommit(boolean):void
beforeCompletion():void
afterCompletion(int):void
hashCode():int
equals(Object):boolean
<<Java Class>>
AbstractIndex
eu.trade.repo.index.impl
LOG: Logger
ATOMIC_INDEX: String
taskFactory: IndexTaskServiceFactory
indexSynchronizer: IndexSynchronizer
indexExecutorSelector: IndexExecutorSelector
AbstractIndex()
executeIfAllowed(IndexTask,boolean):void
executeIfAllowed(Integer,Integer,String,BigInteger,boolean,IndexOperation,boolean,String):void
createTask(Integer,Integer,String,BigInteger,boolean,IndexOperation,String):IndexTask
getIndexSynchronizer():IndexSynchronizer
<<Java Class>>
CleanOrphanIndexesJob
eu.trade.repo.index.jobs
LOG: Logger
indexDelegate: IndexDelegate
CleanOrphanIndexesJob()
execute():void
registerIndexTasksForOrphanIndexList(List<Integer>):void
executeOperation(Integer,Integer,String,BigInteger,boolean,IndexOperation):void
<<Java Class>>
RetryContentIndexesJob
eu.trade.repo.index.jobs
LOG: Logger
indexDelegate: IndexDelegate
executorsHelper: JobIndexExecutorsHelper
indexConfigHolder: IndexConfigHolder
RetryContentIndexesJob()
execute():void
searchAndRegisterDocumentsToIndex(IndexExecutorPool):void
registerIndexTasksForErrorAndUnfinishedIndexList(List<IndexObjectDTO>):void
executeOperation(Integer,Integer,String,BigInteger,boolean,IndexOperation):void
<<Java Class>>
IndexImpl
eu.trade.repo.index.impl
LOG: Logger
IndexImpl()
executeOperation(Integer,Integer,String,BigInteger,boolean,IndexOperation):void
onApplicationEvent(IndexEvent):void
<<Java Class>>
AbstractIndexBackgroundJob
eu.trade.repo.index.jobs
OWNER_JOB_CLEAN_ORPHAN_INDEXES: String
OWNER_JOB_RETRY_CONTENT_INDEX_IN_ERROR: String
OWNER_JOB_RETRY_METADATA_INDEX_IN_ERROR: String
df: DateFormat
owner: String
AbstractIndexBackgroundJob()
getOwner(String):String
<<Java Class>>
RetryMetadataIndexesJob
eu.trade.repo.index.jobs
LOG: Logger
indexDelegate: IndexDelegate
executorsHelper: JobIndexExecutorsHelper
indexConfigHolder: IndexConfigHolder
RetryMetadataIndexesJob()
execute():void
searchAndRegisterMetadataToIndex(IndexExecutorPool):void
registerIndexTasksForErrorAndUnfinishedIndexList(List<IndexObjectDTO>):void
executeOperation(Integer,Integer,String,BigInteger,boolean,IndexOperation):void

4.2. IndexSynchronizer

The IndexSynchronizer is only a store for IndexTasks. Internally, it delegates to two IndexOperationTypeSynchronizer objects: one to store METADATA index operation type tasks and another to store CONTENT tasks.

Each IndexOperationTypeSynchronizer manages two maps:

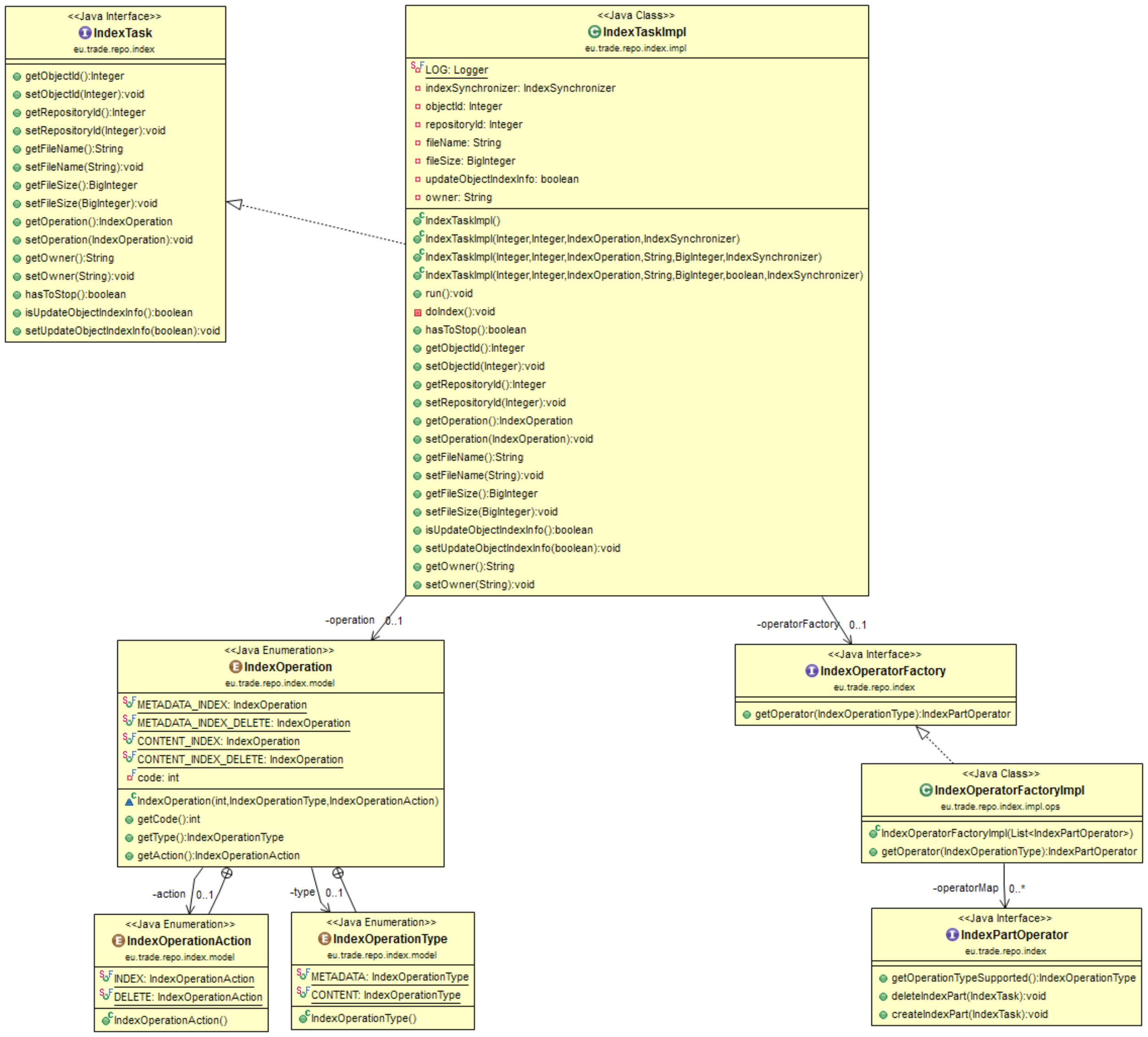
- "executing map": the map that registers the IndexTasks that can be added to the task executor
- "waiting map": it is used to know if there is some other IndexTask (apart from executing one) programmed for a CMIS object so the executing task has to stop. Only the last arrived IndexTask is saved in this map.



<<Java Interface>>
Index
eu.trade.repo.index
executeOperation(Integer,Integer,String,BigInteger,boolean,IndexOperation):void
<<Java Class>>
AbstractIndex
eu.trade.repo.index.impl
LOG: Logger
ATOMIC_INDEX: String
AbstractIndex()
executeIfAllowed(IndexTask,boolean):void
executeIfAllowed(Integer,Integer,String,BigInteger,boolean,IndexOperation,boolean,String):void
createTask(Integer,Integer,String,BigInteger,boolean,IndexOperation,String):IndexTask
getIndexSynchronizer():IndexSynchronizer
-indexSynchronizer 0..1
-taskFactory 0..1
-indexExecutorSelector 0..1
<<Java Interface>>
IndexSynchronizer
eu.trade.repo.index
putInQueue(IndexTask,boolean):boolean
doOnTaskFinished(IndexTask):void
isOtherTaskWaiting(IndexOperationType,Integer):boolean
isIndexIdle():boolean
popWaitingTask(IndexOperationType,Integer):IndexTask
<<Java Interface>>
IndexTaskServiceFactory
eu.trade.repo.index
getIndexService():IndexTask
<<Java Interface>>
IndexExecutorSelector
eu.trade.repo.index
getTaskExecutor(IndexTask):TaskExecutor
-indexSynchronizer 0..1
<<Java Class>>
IndexExecutorSelectorImpl
eu.trade.repo.index.impl
smallTasksExecutor: TaskExecutor
largeTasksExecutor: TaskExecutor
metadataTasksExecutor: TaskExecutor
IndexExecutorSelectorImpl()
getTaskExecutor(IndexTask):TaskExecutor
<<Java Class>>
IndexSynchronizerImpl
eu.trade.repo.index.impl
publisher: ApplicationEventPublisher
IndexSynchronizerImpl()
isIndexIdle():boolean
putInQueue(IndexTask,boolean):boolean
isOtherTaskWaiting(IndexOperationType,Integer):boolean
popWaitingTask(IndexOperationType,Integer):IndexTask
doOnTaskFinished(IndexTask):void
setApplicationEventPublisher(ApplicationEventPublisher):void
getSynchronizer(IndexTask):IndexOperationTypeSynchronizer
getSynchronizer(IndexOperationType):IndexOperationTypeSynchronizer
-metadataSynchronizer
-contentSynchronizer
0..1 0..1
-indexWaitingMap
0..*
<<Java Class>>
IndexOperationTypeSynchronizer
eu.trade.repo.index.impl
indexExecutingMap: ConcurrentHashMap<Integer,Integer>
IndexOperationTypeSynchronizer()
isOtherTaskWaiting(Integer):boolean
isIndexIdle():boolean
doOnTaskFinished(IndexTask):void
putInQueue(IndexTask,boolean):boolean
popWaitingTask(Integer):IndexTask
<<Java Interface>>
IndexTask
eu.trade.repo.index
getObjectId():Integer
setObjectId(Integer):void
getRepositoryId():Integer
setRepositoryId(Integer):void
getFileName():String
setFileName(String):void
getFileSize():BigInteger
setFileSize(BigInteger):void
getOperation():IndexOperation
setOperation(IndexOperation):void
getOwner():String
setOwner(String):void
hasToStop():boolean
isUpdateObjectIndexInfo():boolean
setUpdateObjectIndexInfo(boolean):void
<<Java Class>>
IndexTaskImpl
eu.trade.repo.index.impl
LOG: Logger
operatorFactory: IndexOperatorFactory
objectId: Integer
repositoryId: Integer
operation: IndexOperation
fileName: String
fileSize: BigInteger
updateObjectIndexInfo: boolean
owner: String
IndexTaskImpl()
IndexTaskImpl(Integer,Integer,IndexOperation,IndexSynchronizer)
IndexTaskImpl(Integer,Integer,IndexOperation,String,BigInteger,IndexSynchronizer)
IndexTaskImpl(Integer,Integer,IndexOperation,String,BigInteger,boolean,IndexSynchronizer)
run():void
doIndex():void
hasToStop():boolean
getObjectId():Integer
setObjectId(Integer):void
getRepositoryId():Integer
setRepositoryId(Integer):void
getOperation():IndexOperation
setOperation(IndexOperation):void
getFileName():String
setFileName(String):void
getFileSize():BigInteger
setFileSize(BigInteger):void
isUpdateObjectIndexInfo():boolean
setUpdateObjectIndexInfo(boolean):void
getOwner():String
setOwner(String):void
-indexConfigHolder
0..1
<<Java Class>>
IndexConfigHolder
eu.trade.repo.index.impl
config: Configuration
IndexConfigHolder()
getDocumentIndexWordLimitSize():int
getSegmentSize():int
getWordPageSize():int
getWordPositionPageSize():int
getDeletePageSize():int
getMaxIndexAttempts():int
getRemainingPoolQueueThreshold():double
getQueueSmallTasksCapacity():int
getQueueLargeTasksCapacity():int
getQueueMetadataTasksCapacity():int
getQueueSelectionLimitSize():int

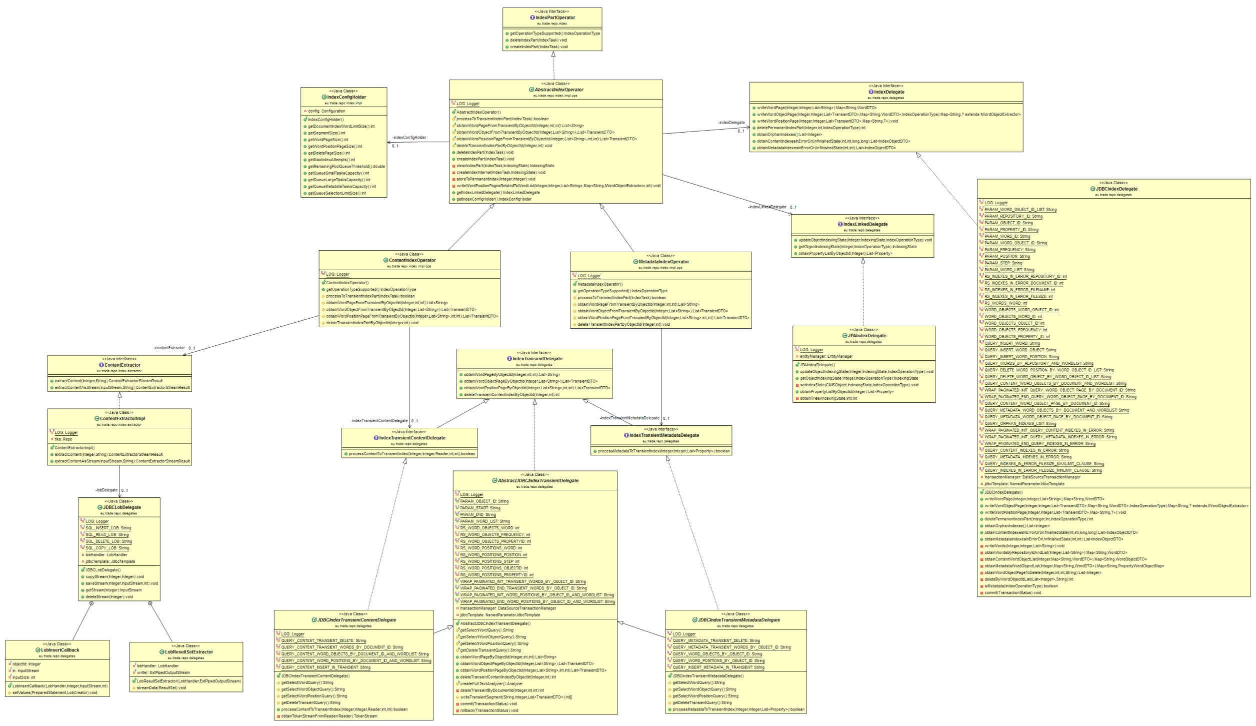
4.3. Index task

The index task is the object that stores information about the CMIS object to be indexed and the index operation to be performed, as well as the thread (Runnable) that executes the indexing process. It is responsible, based on its information, for selecting the propper index operator wich has the knowledge of the indexing process.


<<Java Interface>>
IndexTask
eu.trade.repo.index
getObjectId():Integer
setObjectId(Integer):void
getRepositoryId():Integer
setRepositoryId(Integer):void
getFileName():String
setFileName(String):void
getFileSize():BigInteger
setFileSize(BigInteger):void
getOperation():IndexOperation
setOperation(IndexOperation):void
getOwner():String
setOwner(String):void
hasToStop():boolean
isUpdateObjectIndexInfo():boolean
setUpdateObjectIndexInfo(boolean):void
<<Java Class>>
IndexTaskImpl
eu.trade.repo.index.impl
LOG: Logger
indexSynchronizer: IndexSynchronizer
objectId: Integer
repositoryId: Integer
fileName: String
fileSize: BigInteger
updateObjectIndexInfo: boolean
owner: String
IndexTaskImpl()
IndexTaskImpl(Integer,Integer,IndexOperation,IndexSynchronizer)
IndexTaskImpl(Integer,Integer,IndexOperation,String,BigInteger,IndexSynchronizer)
IndexTaskImpl(Integer,Integer,IndexOperation,String,BigInteger,boolean,IndexSynchronizer)
run():void
doIndex():void
hasToStop():boolean
getObjectId():Integer
setObjectId(Integer):void
getRepositoryId():Integer
setRepositoryId(Integer):void
getOperation():IndexOperation
setOperation(IndexOperation):void
getFileName():String
setFileName(String):void
getFileSize():BigInteger
setFileSize(BigInteger):void
isUpdateObjectIndexInfo():boolean
setUpdateObjectIndexInfo(boolean):void
getOwner():String
setOwner(String):void
-operation
0..1
-operatorFactory
0..1
<<Java Enumeration>>
IndexOperation
eu.trade.repo.index.model
METADATA_INDEX: IndexOperation
METADATA_INDEX_DELETE: IndexOperation
CONTENT_INDEX: IndexOperation
CONTENT_INDEX_DELETE: IndexOperation
code: int
IndexOperation(int,IndexOperationType,IndexOperationAction)
getCode():int
getType():IndexOperationType
getAction():IndexOperationAction
<<Java Interface>>
IndexOperatorFactory
eu.trade.repo.index
getOperator(IndexOperationType):IndexPartOperator
<<Java Class>>
IndexOperatorFactoryImpl
eu.trade.repo.index.impl.ops
IndexOperatorFactoryImpl(List<IndexPartOperator>)
getOperator(IndexOperationType):IndexPartOperator
-operatorMap
0..*
-action
0..1
-type
0..1
<<Java Enumeration>>
IndexOperationAction
eu.trade.repo.index.model
INDEX: IndexOperationAction
DELETE: IndexOperationAction
IndexOperationAction()
<<Java Enumeration>>
IndexOperationType
eu.trade.repo.index.model
METADATA: IndexOperationType
CONTENT: IndexOperationType
IndexOperationType()
<<Java Interface>>
IndexPartOperator
eu.trade.repo.index
getOperationTypeSupported():IndexOperationType
deleteIndexPart(IndexTask):void
createIndexPart(IndexTask):void

4.4. Index Part Operator

Subclasses of this interface have the knowledge necessary to permform index operations (delete or create) according to the operation type supported (metadata or content).



4.5. Text analyzers

Extensions of Lucene analyzers, they are able to interpret a text, read it word by word, and process each word (i.e. normalizing accents, upper/lower cases).

Because of the need of indexing multi-lingual documents, no stop word list is used.

Repo is using two different analyzers:

- FullTextAnalyzer: this is the one used by the index (both content and metadata indexes). It processes contents following the rules below:
  - Word Break rules from the Unicode Text Segmentation algorithm, as specified in Unicode Standard Annex #29. (StandardTokenizer)
  - Converts alphabetic, numeric, and symbolic Unicode characters which are not in the first 127 ASCII characters (the "Basic Latin" Unicode block) into their ASCII equivalents, if one exists. (ASCIIFoldingFilter)
  - Converts upper case letters into lower case. (LowerCaseFilter)
- PropertiesAnalyzer: it is not used by the index. It is used to store normalized values for 'string' cmis:object properties. It's similar to the FullTextAnalyzer except in the way it tokenizes content text. This analyzer emits the entire input as a single token and thus, does not remove any punctuation characters.

4.6. Content extractor

Given a CMIS object identifier, the content's extractor constructs pipes that connect the stream related to that object in database with the Tika's component that is able to interpret the stream as a text.

The whole stream is not loaded into memory but it loads chunks of the stream on demand.

4.7. Transient index

It is an H2 database embedded in the server to store the temporal information resulting from the content extraction process. This database consists of two tables: transient_index and index_transient_metadata.

4.8. Permanent index

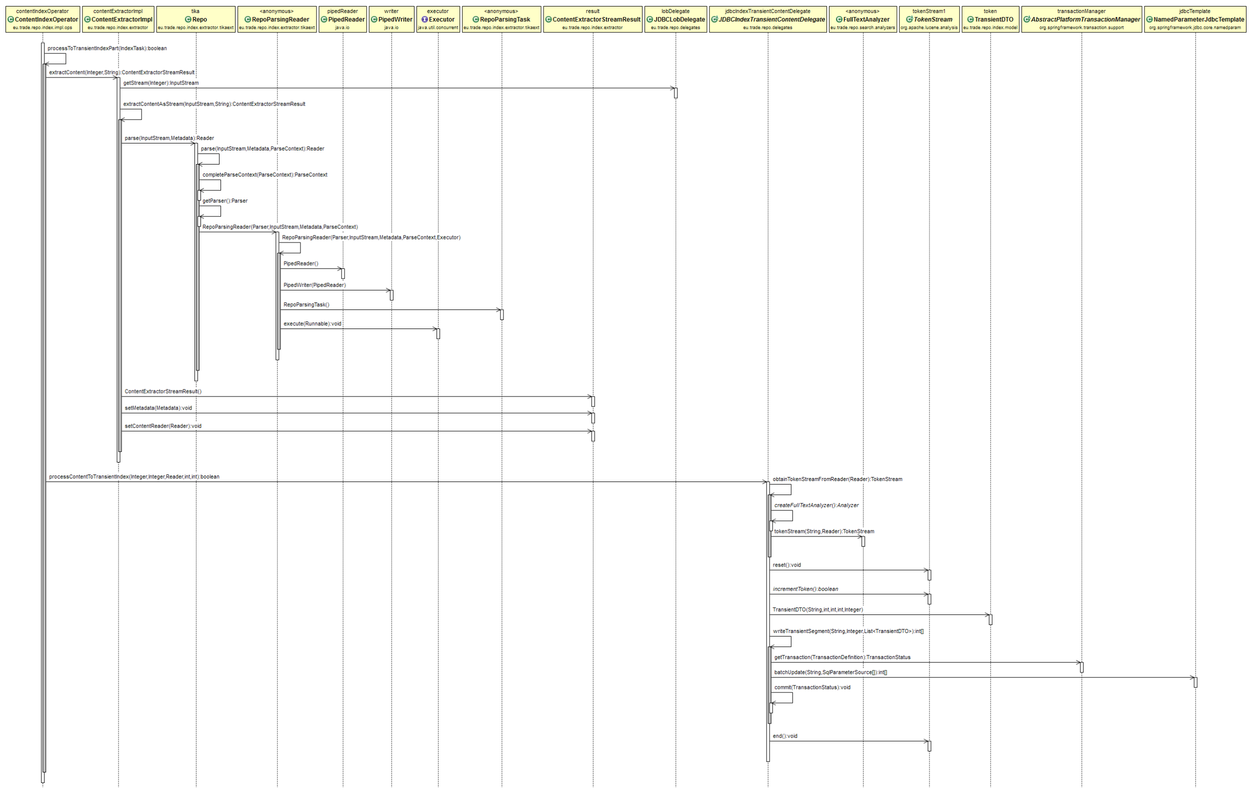
It is the set of the external database tables used to store the reverse index information. It consist of three database tables: Index_word, Index_word_object and Index_word_position.

5. Index processes

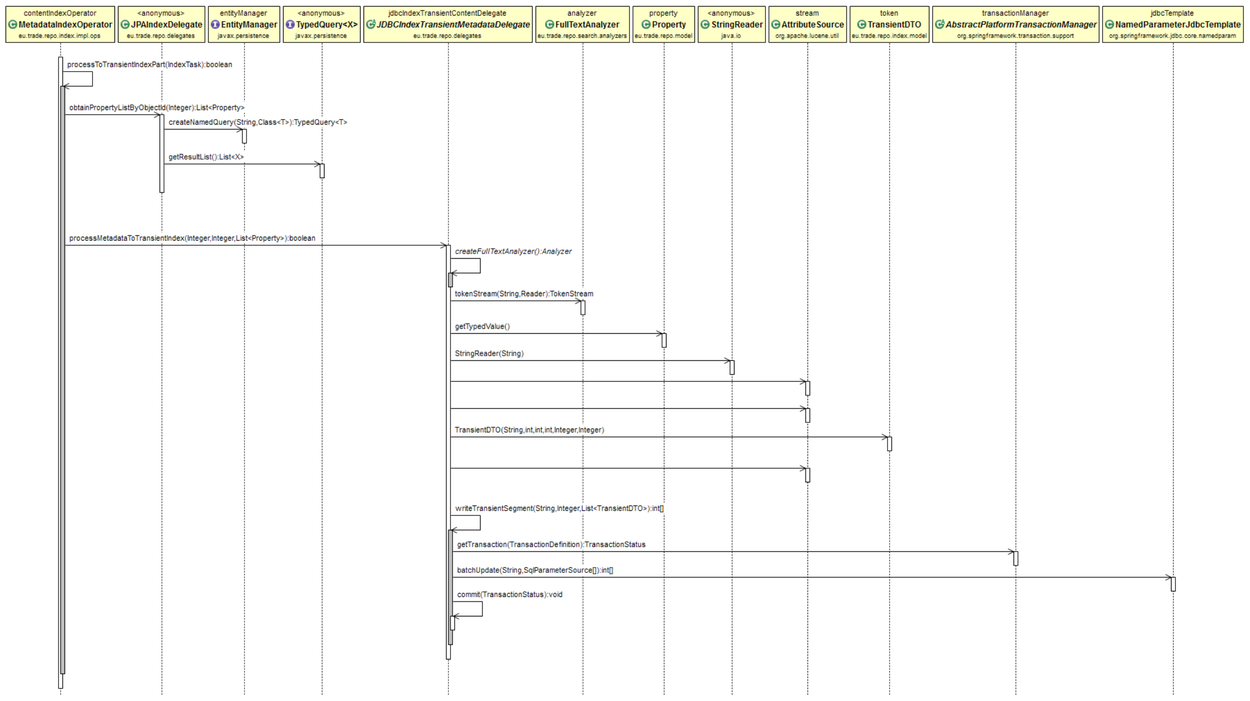
5.1. How words are extracted?

IndexTasks retrieve the information to be indexed from the CMIS database. It retrieves the cmis:object stream for content tasks and the list of 'string' cmis:object properties for metadata tasks. Then the information is processed and saved to the transient index.

5.1.1. Content


contentIndexOperator
ContentIndexOperator
eu.trade.repo.index.impl.ops
contentExtractorImpl
ContentExtractorImpl
eu.trade.repo.index.extractor
tika
Repo
eu.trade.repo.index.extractor.tikaext
<anonymous>
RepoParsingReader
eu.trade.repo.index.extractor.tikaext
pipedReader
PipedReader
java.io
writer
PipedWriter
java.io
executor
Executor
java.util.concurrent
<anonymous>
RepoParsingTask
eu.trade.repo.index.extractor.tikaext
result
ContentExtractorStreamResult
eu.trade.repo.index.extractor
lobDelegate
JDBCLobDelegate
eu.trade.repo.delegates
jdbcIndexTransientContentDelegate
JDBCIndexTransientContentDelegate
eu.trade.repo.delegates
<anonymous>
FullTextAnalyzer
eu.trade.repo.search.analyzers
tokenStream1
TokenStream
org.apache.lucene.analysis
token
TransientDTO
eu.trade.repo.index.model
transactionManager
AbstractPlatformTransactionManager
org.springframework.transaction.support
jdbcTemplate
NamedParameterJdbcTemplate
org.springframework.jdbc.core.namedparam
processToTransientIndexPart(IndexTask):boolean
extractContent(Integer,String):ContentExtractorStreamResult
getStream(Integer):InputStream
extractContentAsStream(InputStream,String):ContentExtractorStreamResult
parse(InputStream,Metadata):Reader
parse(InputStream,Metadata,ParseContext):Reader
completeParseContext(ParseContext):ParseContext
getParser():Parser
RepoParsingReader(Parser,InputStream,Metadata,ParseContext)
RepoParsingReader(Parser,InputStream,Metadata,ParseContext,Executor)
PipedReader()
PipedWriter(PipedReader)
RepoParsingTask()
execute(Runnable):void
ContentExtractorStreamResult()
setMetadata(Metadata):void
setContentReader(Reader):void
processContentToTransientIndex(Integer,Integer,Reader,int,int):boolean
obtainTokenStreamFromReader(Reader):TokenStream
createFullTextAnalyzer():Analyzer
tokenStream(String,Reader):TokenStream
reset():void
incrementToken():boolean
TransientDTO(String,int,int,int,Integer)
writeTransientSegment(String,Integer,List<TransientDTO>):int[]
getTransaction(TransactionDefinition):TransactionStatus
batchUpdate(String,SqlParameterSource[]):int[]
commit(TransactionStatus):void
end():void

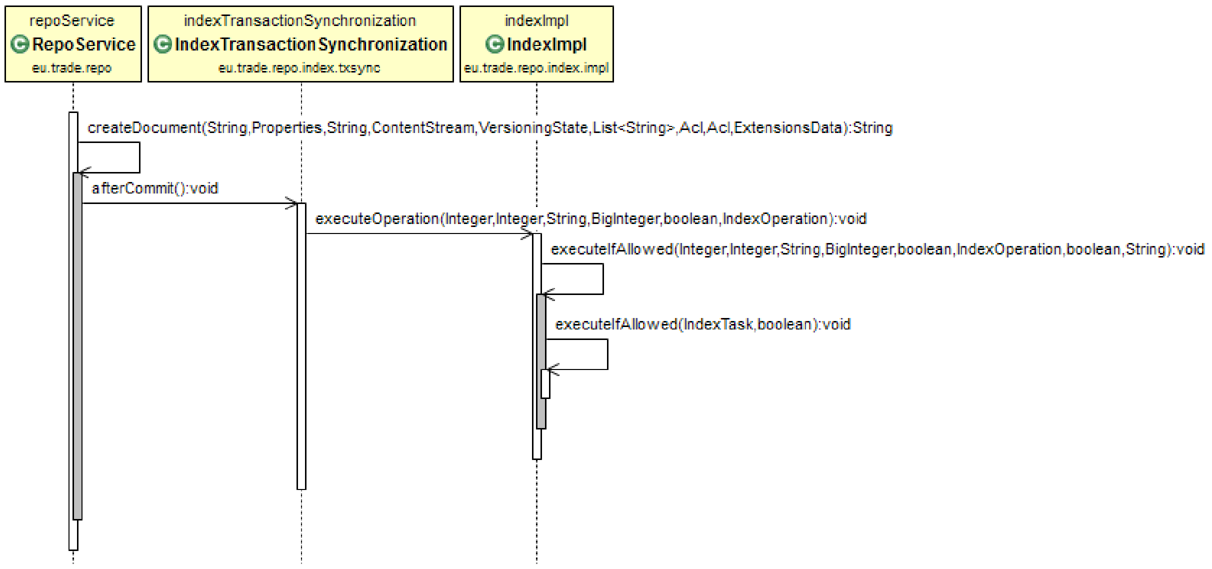
5.1.2. Metadata


contentIndexOperator
MetadataIndexOperator
eu.trade.repo.index.impl.ops
<anonymous>
JPAIndexDelegate
eu.trade.repo.delegates
entityManager
EntityManager
javax.persistence
<anonymous>
TypedQuery<X>
javax.persistence
jdbcIndexTransientContentDelegate
JDBCIndexTransientMetadataDelegate
eu.trade.repo.delegates
analyzer
FullTextAnalyzer
eu.trade.repo.search.analyzers
property
Property
eu.trade.repo.model
<anonymous>
StringReader
java.io
stream
AttributeSource
org.apache.lucene.util
token
TransientDTO
eu.trade.repo.index.model
transactionManager
AbstractPlatformTransactionManager
org.springframework.transaction.support
jdbcTemplate
NamedParameterJdbcTemplate
org.springframework.jdbc.core.namedparam
processToTransientIndexPart(IndexTask):boolean
obtainPropertyListByObjectId(Integer):List<Property>
createNamedQuery(String,Class<T>):TypedQuery<T>
getResultList():List<X>
processMetadataToTransientIndex(Integer,Integer,List<Property>):boolean
createFullTextAnalyzer():Analyzer
tokenStream(String,Reader):TokenStream
getTypedValue()
StringReader(String)
TransientDTO(String,int,int,int,Integer,Integer)
writeTransientSegment(String,Integer,List<TransientDTO>):int[]
getTransaction(TransactionDefinition):TransactionStatus
batchUpdate(String,SqlParameterSource[]):int[]
commit(TransactionStatus):void

5.2. How the index is populated?

The process is as follows:

1. The entry point create an IndexTask, specifying which index operation (index or delete index) it is to be performed and some other data about the cmis object.

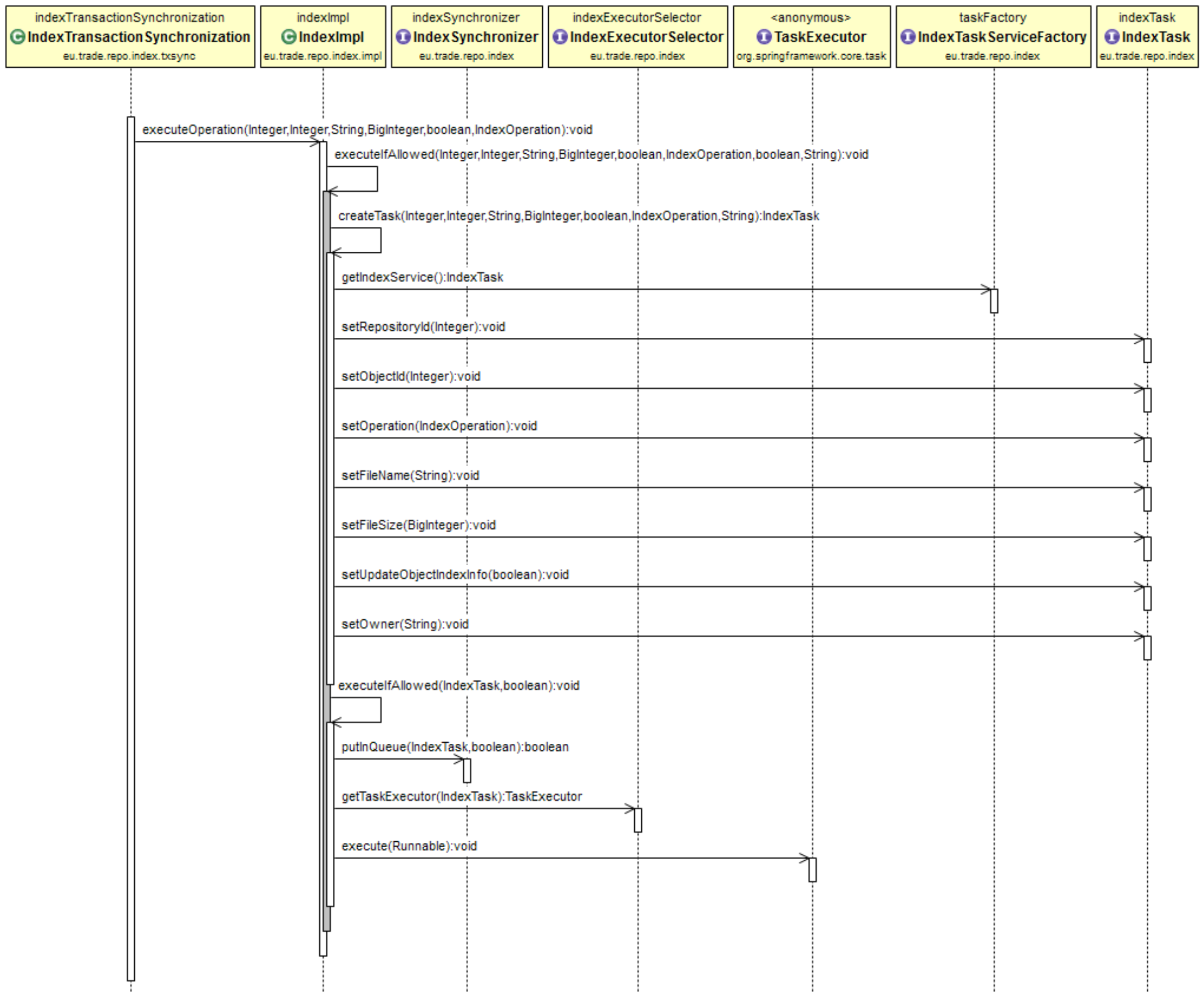

repoService
RepoService
eu.trade.repo
indexTransactionSynchronization
IndexTransactionSynchronization
eu.trade.repo.index.txsync
indexImpl
IndexImpl
eu.trade.repo.index.impl
createDocument(String,Properties,String,ContentStream,VersioningState,List<String>,Acl,Acl,ExtensionsData):String
afterCommit():void
executeOperation(Integer,Integer,String,BigInteger,boolean,IndexOperation):void
executeIfAllowed(Integer,Integer,String,BigInteger,boolean,IndexOperation,boolean,String):void
executeIfAllowed(IndexTask,boolean):void

2. The entry point use the IndexSynchronizer.putInQueue(IndexTask, waiting) method to check if it is possible to add the IndexTask to task executor.

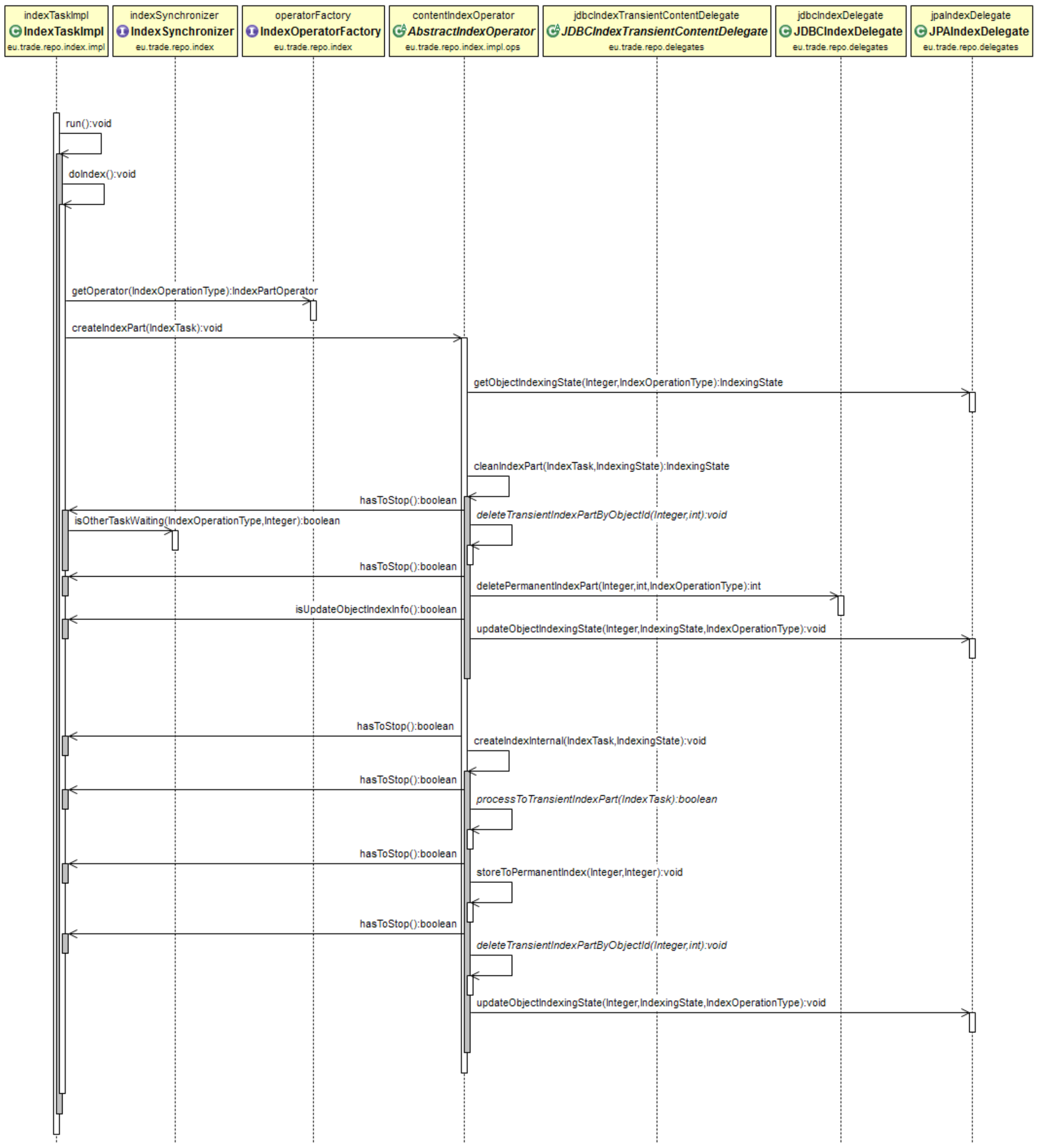
   - If it is possible, it adds the index to the task executor, so the IndexTask.doIndex() will be executed.

- If not, depending on the "waiting" argument, different things could happen:
  - waiting = false: the index synchronizer does nothing with the task and neither the entry point, so the IndexTask is lost. It is the way the background jobs act: if the IndexTask cannot be executed directly, it is ignored. This is to avoid stopping an executing atomic task.
  - waiting = true: the index synchronizer put the IndexTask in the "waiting map".

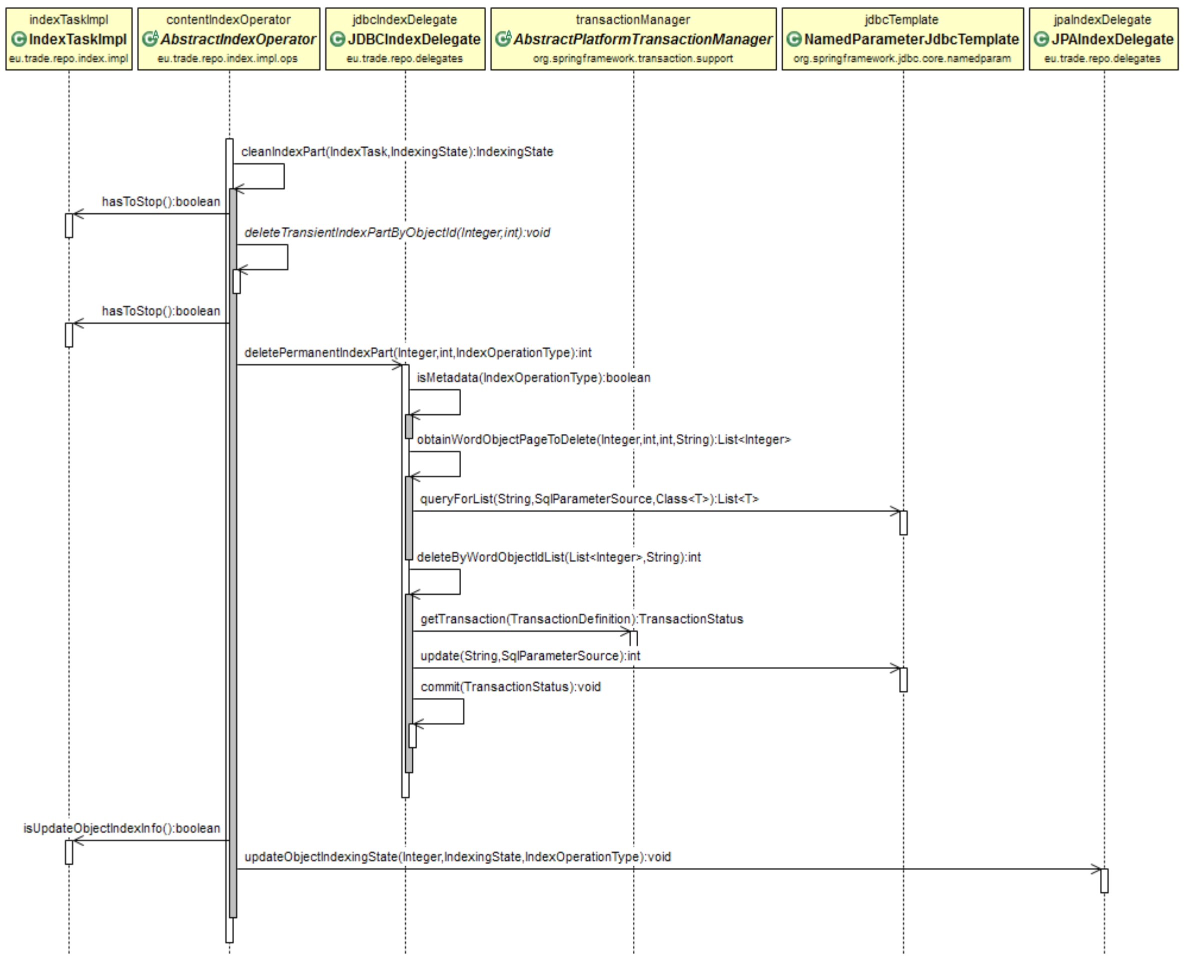
3. If the synchronizer returns that is possible to add the IndexTask to a task executor, then the IndexExecutorSelector is invoked to determine which task executor to use. This is done based on the index operation type and the file size to be indexed if the operation type is CONTENT. This way, there is a task executor for metadata tasks, another for small files and the last for big files to be indexed.


indexTransactionSynchronization
IndexTransactionSynchronization
eu.trade.repo.index.txsync
indexImpl
IndexImpl
eu.trade.repo.index.impl
indexSynchronizer
IndexSynchronizer
eu.trade.repo.index
indexExecutorSelector
IndexExecutorSelector
eu.trade.repo.index
<anonymous>
TaskExecutor
org.springframework.core.task
taskFactory
IndexTaskServiceFactory
eu.trade.repo.index
indexTask
IndexTask
eu.trade.repo.index
executeOperation(Integer,Integer,String,BigInteger,boolean,IndexOperation):void
executeIfAllowed(Integer,Integer,String,BigInteger,boolean,IndexOperation,boolean,String):void
createTask(Integer,Integer,String,BigInteger,boolean,IndexOperation,String):IndexTask
getIndexService():IndexTask
setRepositoryId(Integer):void
setObjectId(Integer):void
setOperation(IndexOperation):void
setFileName(String):void
setFileSize(BigInteger):void
setUpdateObjectIndexInfo(boolean):void
setOwner(String):void
executeIfAllowed(IndexTask,boolean):void
putInQueue(IndexTask,boolean):boolean
getTaskExecutor(IndexTask):TaskExecutor
execute(Runnable):void

4. The IndexTask is executed.
   - Create index part


indexTaskImpl
IndexTaskImpl
eu.trade.repo.index.impl
indexSynchronizer
IndexSynchronizer
eu.trade.repo.index
operatorFactory
IndexOperatorFactory
eu.trade.repo.index
contentIndexOperator
AbstractIndexOperator
eu.trade.repo.index.impl.ops
jdbcIndexTransientContentDelegate
JDBCIndexTransientContentDelegate
eu.trade.repo.delegates
jdbcIndexDelegate
JDBCIndexDelegate
eu.trade.repo.delegates
jpaIndexDelegate
JPAIndexDelegate
eu.trade.repo.delegates
run():void
doIndex():void
getOperator(IndexOperationType):IndexPartOperator
createIndexPart(IndexTask):void
getObjectIndexingState(Integer,IndexOperationType):IndexingState
cleanIndexPart(IndexTask,IndexingState):IndexingState
hasToStop():boolean
deleteTransientIndexPartByObjectId(Integer,int):void
isOtherTaskWaiting(IndexOperationType,Integer):boolean
hasToStop():boolean
deletePermanentIndexPart(Integer,int,IndexOperationType):int
isUpdateObjectIndexInfo():boolean
updateObjectIndexingState(Integer,IndexingState,IndexOperationType):void
hasToStop():boolean
createIndexInternal(IndexTask,IndexingState):void
hasToStop():boolean
processToTransientIndexPart(IndexTask):boolean
hasToStop():boolean
storeToPermanentIndex(Integer,Integer):void
hasToStop():boolean
deleteTransientIndexPartByObjectId(Integer,int):void
updateObjectIndexingState(Integer,IndexingState,IndexOperationType):void

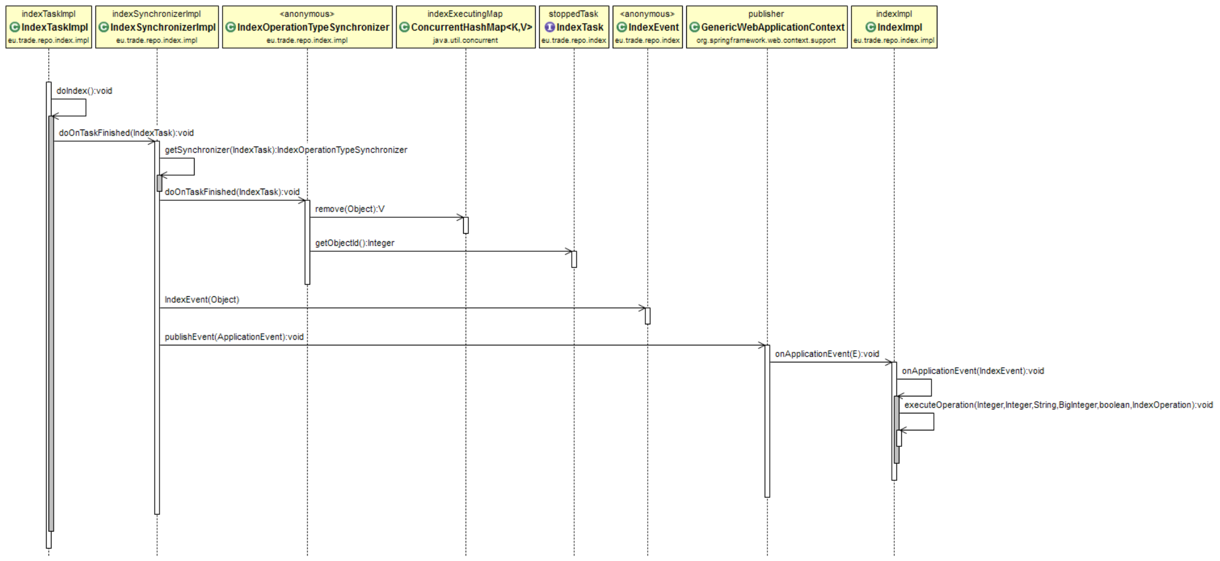
- Delete index part


indexTaskImpl
IndexTaskImpl
eu.trade.repo.index.impl
contentIndexOperator
AbstractIndexOperator
eu.trade.repo.index.impl.ops
jdbcIndexDelegate
JDBCIndexDelegate
eu.trade.repo.delegates
transactionManager
AbstractPlatformTransactionManager
org.springframework.transaction.support
jdbcTemplate
NamedParameterJdbcTemplate
org.springframework.jdbc.core.namedparam
jpaIndexDelegate
JPAIndexDelegate
eu.trade.repo.delegates
cleanIndexPart(IndexTask,IndexingState):IndexingState
hasToStop():boolean
deleteTransientIndexPartByObjectId(Integer,int):void
hasToStop():boolean
deletePermanentIndexPart(Integer,int,IndexOperationType):int
isMetadata(IndexOperationType):boolean
obtainWordObjectPageToDelete(Integer,int,int,String):List<Integer>
queryForList(String,SqlParameterSource,Class<T>):List<T>
deleteByWordObjectIdList(List<Integer>,String):int
getTransaction(TransactionDefinition):TransactionStatus
update(String,SqlParameterSource):int
commit(TransactionStatus):void
isUpdateObjectIndexInfo():boolean
updateObjectIndexingState(Integer,IndexingState,IndexOperationType):void

When the task execution finishes, it invokes the IndexSynchronizer.doOnTaskFinished() method.

5. The IndexSynchronizer removes the IndexTask from the "executing map" and fires and IndexEvent

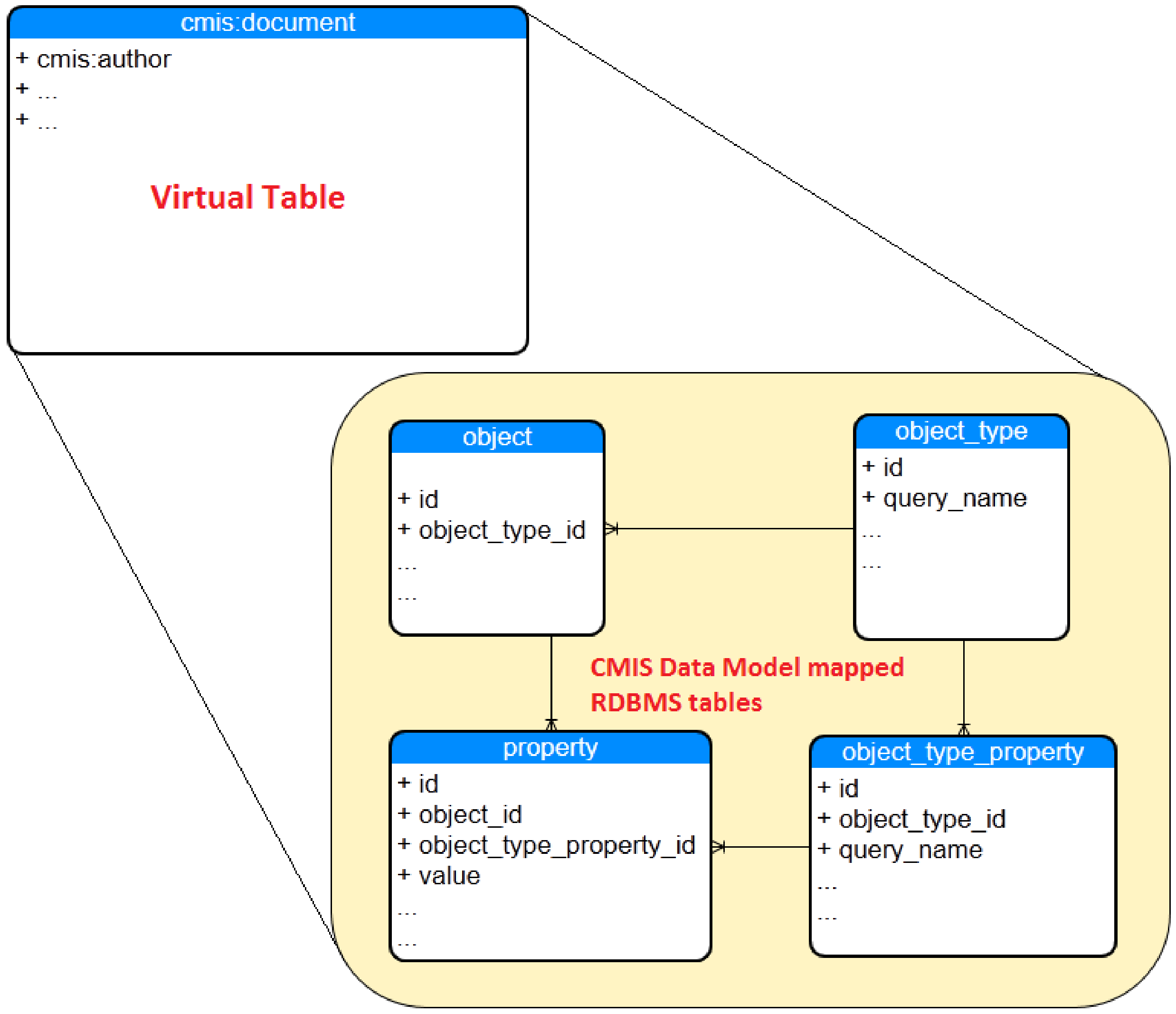
6. The IndexImpl listen to that event, retrieves the waiting task for that cmis object from the IndexSynchronizer and begins the process with this other task, if it exists.


indexTaskImpl
IndexTaskImpl
eu.trade.repo.index.impl
indexSynchronizerImpl
IndexSynchronizerImpl
eu.trade.repo.index.impl
<anonymous>
IndexOperationTypeSynchronizer
eu.trade.repo.index.impl
indexExecutingMap
ConcurrentHashMap<K,V>
java.util.concurrent
stoppedTask
IndexTask
eu.trade.repo.index
<anonymous>
IndexEvent
eu.trade.repo.index
publisher
GenericWebApplicationContext
org.springframework.web.context.support
indexImpl
IndexImpl
eu.trade.repo.index.impl
doIndex():void
doOnTaskFinished(IndexTask):void
getSynchronizer(IndexTask):IndexOperationTypeSynchronizer
doOnTaskFinished(IndexTask):void
remove(Object):V
getObjectId():Integer
IndexEvent(Object)
publishEvent(ApplicationEvent):void
onApplicationEvent(E):void
onApplicationEvent(IndexEvent):void
executeOperation(Integer,Integer,String,BigInteger,boolean,IndexOperation):void

Chapter 4. Query Service

1. Introduction

The CMIS standard implemented by the Repo provides a type-based Query service which allows the querying and discovery of data using a query language with a syntax similar to SQL. In order to achieve this, the query service projects the CMIS Data Model into **virtual tables** and **virtual columns**, such that each Object-Type defined in the Repo is projected into its own virtual table and each Object-Type-Property projected into a Virtual-Column of the table:



Projection of the CMIS Data Model into Virtual Tables

allowing queries like:

```
select * from cmis:document where cmis:author = 'John Doe'
```

where **cmis:document** and **cmis:author** are **VIRTUAL** tables and columns that don't exist as actual tables in the Repo RDBMS.

The projection of Data Model tables into Virtual Tables is achieved at runtime (query-time), by expanding the received CMIS Query (on the Virtual Tables) into a relational query (onto the actual persisted tables representing the Data Model)

e.g.

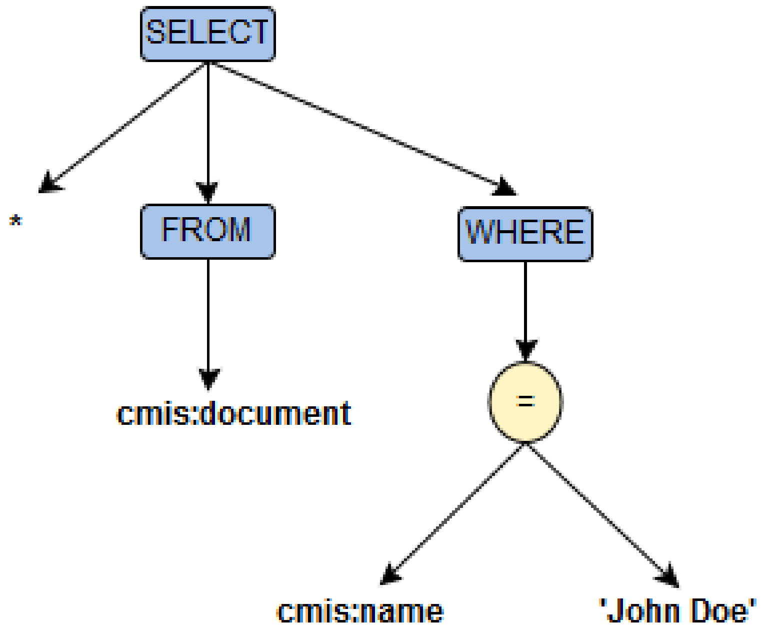
select * from cmis:document ->

select * from object o join object_type ot on o.object_type_id = ot.id WHERE ot.query_name = 'cmis:document'

Transformation of the incoming CMIS query into the equivalent relational query is achieved by reading the incoming query, extracting the relevant, **semantically meaningful words** (from now on, **Tokens**), e.g. **'select'**, **'*'**, **'cmis:document'**, then generating the equivalent relational query statements.

Processing the tokens in the received CMIS Query in order to generate the equivalent relational query is assisted by another step **between extracting the tokens from the query and generating the output**, that of generating an **Abstract Syntax Tree (AST)**.

An AST provides an hierarchical (tree-based) representation of the query, where each node in the tree represents a token in the input:



Abstract Syntax Tree (AST) for query SELECT * FROM cmis:document WHERE cmis:name = 'John Doe'

Generating an AST of the input query, which is then processed to generate the output relational query, provides various advantages:

- The process of parsing the input and verifying that the syntax of the input query is correct, is decoupled from the process of generating the output relational query. This greatly increases the clarity of the code, as the code for reading the input character stream, identifying tokens and verifying that the syntax is correct is separated from the the code which generates the output relational query.

- Generating the output query can be simplified to a process of traversing the AST of the input query, and producing equivalent relational statements for every node (token) encountered in the tree.
- Furthermore, using an AST allows the query service to optimize the query by detecting and removing/modifying non-optimal usage structures in the AST tree structure.

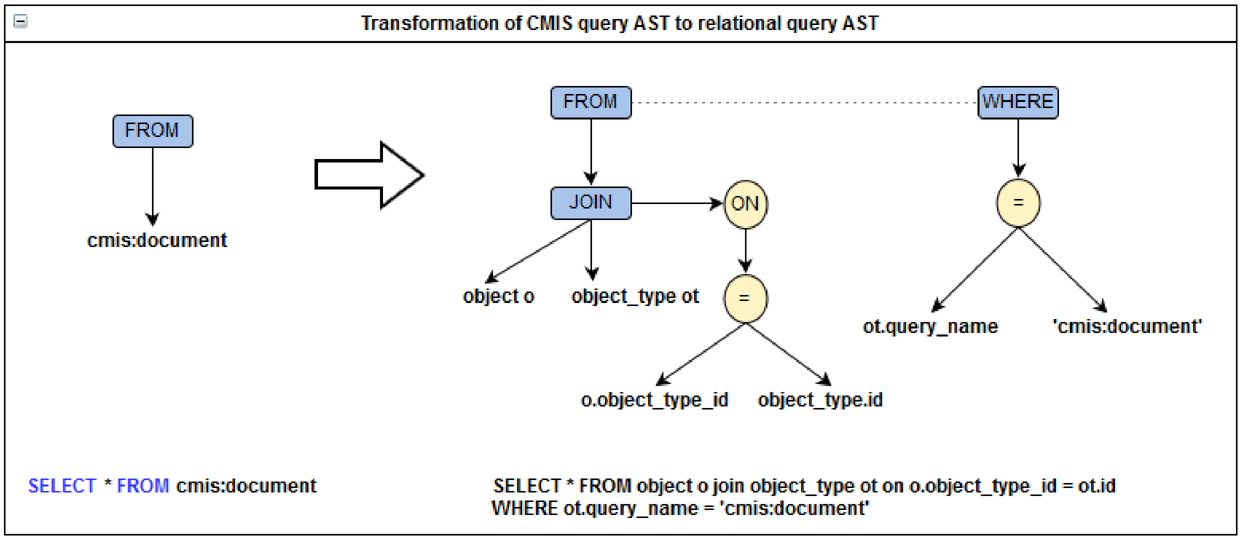
The output relational statement, actually sent to the underlying RDBMS and equivalent to the input CMIS Query can then be generated by traversing (walking) the AST in a recursive, depth-first manner and directly producing equivalent relational statements for every node (token) encountered in the tree.

Furthermore, on traversing each AST node and generating the equivalent relational statement, the generated SQL can be customized to any supported RDBMS, thus allowing the Repo to utilize persistence solutions from various providers. However, this approach is difficult to maintain, as the code generating the relational queries would have to be modified/updated every time a new RDBMS provider was to be supported.

To solve this issue, the Repo Query Service implementation utilizes the JPA Criteria API. The Java Persistence Criteria API enables the definition of dynamic queries through the construction of object-graph-based query definition objects - which provide a tree-like representation of the relational query to be generated - then generates the appropriate relational SQL query for the underlying RDBMS. As multiple providers can be plugged into the JPA implementation, this allows the Repo Query Service to support multiple database systems out of the box without requiring changes to the query generation code.

Therefore, the steps involved in processing a query request are the following:

1. **Tokenize & Lex the input CMIS Query:** Read the input query, extract words according to the CMIS Query Syntax (Lex), then generate Tokens corresponding to each word (each Token is a legal word in the query syntax, with extra information attached to it e.g. position - start/end - in the string, type of token etc).
2. **Parse the tokens, build the CMIS Query AST.**
3. **Walk the CMIS query AST, build equivalent relational query JPA AST.**
4. **Walk the relational JPA AST, generate relational query.**



SELECT * FROM cmis:document

SELECT * FROM object o join object_type ot on o.object_type_id = ot.id
WHERE ot.query_name = 'cmis:document'

In the above steps, Step 4 is completely handled internally by the JPA provider implementation Steps 1 & 2 are handled by ANTLR, while Step 4 is handled internally by the JPA provider implementation

2. Query interfaces/entrypoints

To be updated in future release.

3. Query processing steps

3.1. Step 1: Parse query and generate Abstract Syntax Tree (AST)

Generate tokens (Lexer, Tokenizer), process tokens to generate token tree. ANTLR, lexer/parser generation

3.2. Step 2: Process (walk) AST and generate relational query (JPA)

The AST generated in Step 1 (TBD: add link to previous section) provides a token tree which can then be transformed into a relational query by traversing the AST and generating the equivalent SQL syntax, e.g. 'from cmis:document' -> from object join object_type on object_type.query_name = 'cmis:document'. However, each relational store normally supports a SQL syntax variant for all but the most trivial queries. Therefore, in order to support the widest variety of relational datastores (RDBMSs) and not couple REPO to a particular store provider, another intermediate step is required: rather than transform the AST directly into a database specific SQL query, transform the AST into an AST representing the SQL language in most of it's variants, then generating the final database-specific SQL from the SQL AST. -> JPA

Chapter 5. Policy Service

1. Introduction

Policies were introduced to include custom logic to the server. Following the CMIS specification, the policies are subtypes of `cmis:policy` and should be assigned to objects.

A policy is composed by two parts, the logic and the `cmis:policy` subtype.

The logic of the policy is implemented in Java extending the abstract class `eu.trade.repo.policy.AbstractBasePolicy`. This class implements all the CMIS service methods. For example, if you would like to apply some logic before a document is created, you should overwrite the method `createDocument`.

A `cmis:policy` subtype is needed to link the Java code with a CMIS object, it's also possible to parameterise the policy behaviour with the policy attributes.

The policies could be triggered by:

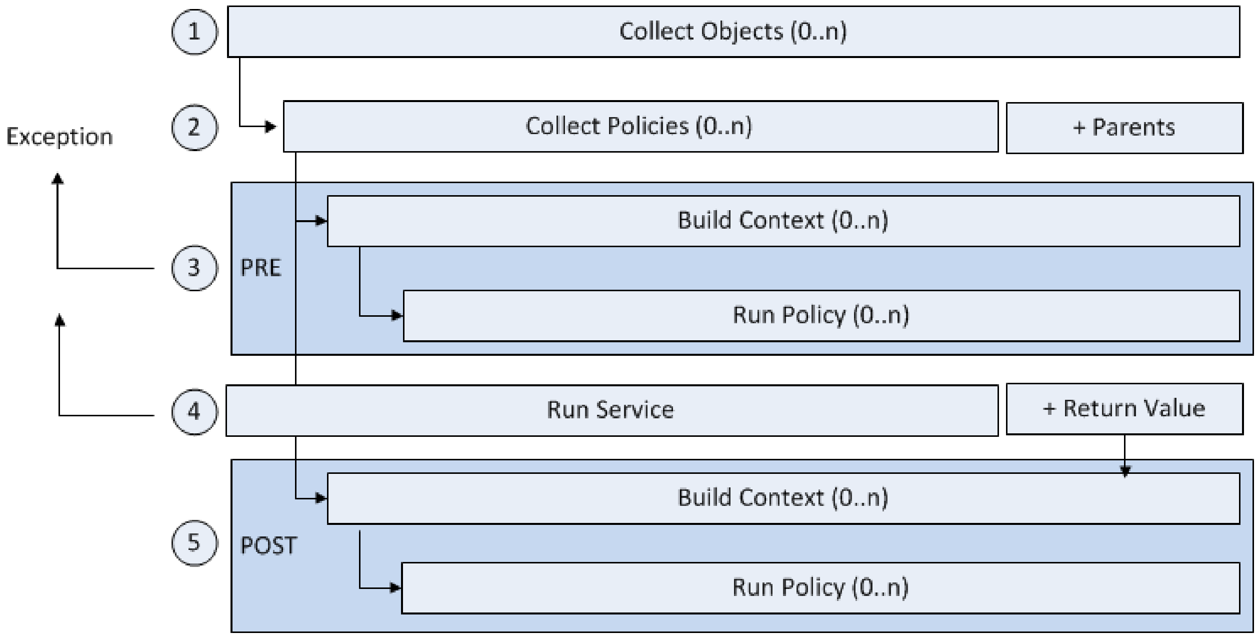
- CMIS service calls
- Server events (startup, shutdown, create or delete repository, change configuration settings) *
- Another policy *
- Time-based *

Some examples of policies could be: quota, notification and initialise.

2. Policy life-cycle

On server startup all the policy types are registered in the system. New policy types cannot be added at runtime but policies of the registered types could be created or modified depending on the security settings.

See below a diagram with the life-cycle of the policies triggered by CMIS services.



1. All CMIS service executions are intercepted and each of the input objects involved are collected. There are services without input objects, services with one single object and services with several objects. For example, the Repository services do not have object parameters, Object services like `createDocument` has one (the parent folder), and `moveObject` has three objects involved (the object to move, the source folder and the target folder).

2. All the applied policies are retrieved for all the objects collected in the previous step. Policies are searched for the current object and for all the parents up to the root folder.

3. For each policy found, a policy context is created and the policy logic is triggered. This step could be used for filtering the CMIS service, and exception could be thrown stopping the execution.

4. The CMIS service is executed. The service call could fail and throw and exception, in this case no more policy logic is executed.

5. For each policy found, a new context is created. The return value of the service call is included to allow modifications. This step could be used for post-processing tasks, like logging or notifications. The logic of the policy after the service call must not interrupt the execution.

Note

The order of the execution of the policies can not be enforced. Avoid implementing policies that require other policies to be executed or any other kind of dependency. Policies logic should be understood as an atomic and independent operation.

Important

Rule 1: the policy triggered will be executed considering the level of access of the creator of the policy object. For example, if the policy object was created by Alice, the scope of the execution is limited to the objects Alice can manage.

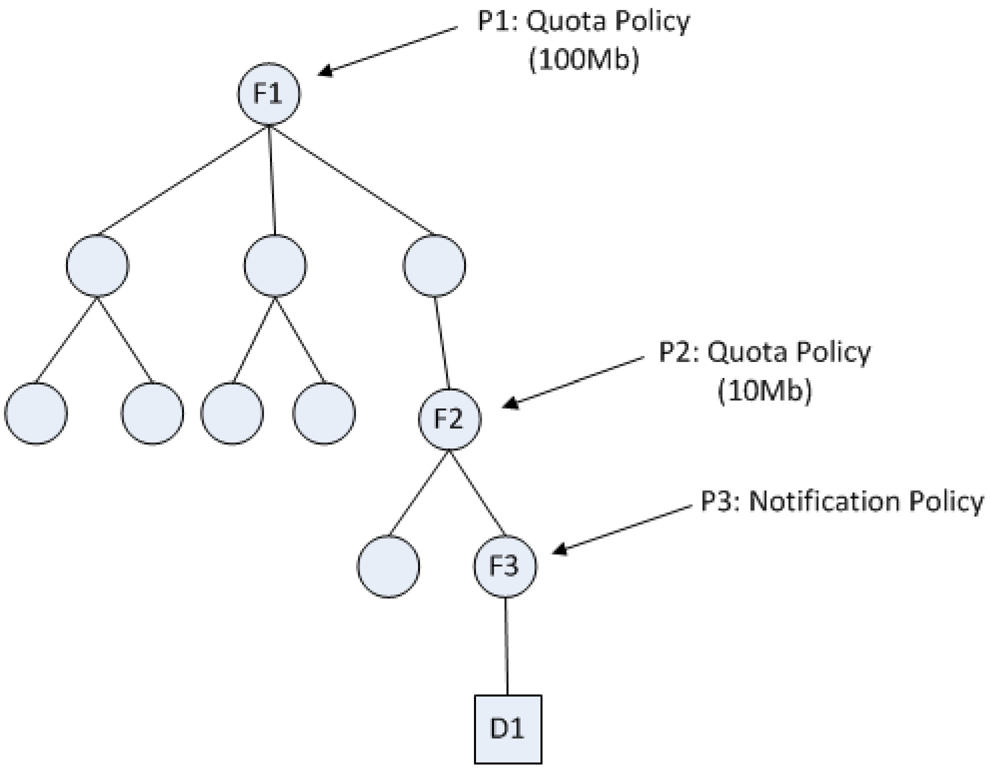
Rule 2: A user service execution could only be affected by a policy that the user can read. For example, if the policy object is created with the ACE "editors" cmis:read, only the users with the principal Id "editors" will be affected by the policy process.

2.1. Policy Context

As described previously, a Policy Context will be available to the Policy logic with all the information needed for the implementation.

Policy Object	Object with the policy information. Often used to parameterise the behaviour of the policy with the policy object properties.
Applied Object	Object associated with the triggered policy, could be the direct object in the parameters list of the service or an indirect object like one of the parent folders. The current logged user may not have access to this object.
CMIS Session	A session with the credentials of the policy object creator. Could be used for interacting with the respository.
Policy State	PRE or POST, used to differenciate if the logic must be executed before and/or after the service call.
Return Value	A reference to the return value of the service call. Only available in the POST step. Could be used to modify the response to the CMIS client.

2.2. Example



In the previous diagram, the circles represents folders, and the squares documents. The lines between these shapes indicates the parent/children relationship.

A CMIS client starts the service call of the method `createDocument`, the document to be created is D1. In the parameters of the call, F3 is indicated as parent folder.

The first step in the policies life-cycle is to collect the parameters of CMIS service calls that references Objects, in this example the only object involved is F3.

```
Objects = { F3 }
```

The next step is to find the policies associated to the previous set of Objects. The policy associated with the object F3 is the Notification policy P3. All the ancestors of F3 are consireded for searching policies. In the example the folders F2 and F1 has policies associated, so the policies to be triggered on the creation of the document D1 are P1, P2 and P3.

```
Policies = { P1, P2, P3 }
```

A context is created for each of the policies:

```
context P1 = {
 Policy Object: P1
 Applied Object: F1
 Policy State: PRE
}

context P2 = {
 Policy Object: P2
 Applied Object: F2
 Policy State: PRE
}

context P3 = {
 Policy Object: P3
 Applied Object: F3
 Policy State: PRE
}
```

All policies are triggered with the correspondent context. The order of the execution is managed by the server. Policies logic must not relay on an specific order.

As the quota policies are applying a filter to the CMIS service, the logic will be triggered before the service, with the PRE state. The notification policy logic, on the contrary, is going to be executed after the service because it will notify the new document created. It is the responsability of the policy logic developer to use the values of the context to encode the proper behaviour.

If during the creation of document D1 one of the quota policies contraints is violated, for example the total space of the folder F2 is bigger than 10Mb or the total space of the folder F1 is bigger than 100Mb, the policy logic will throw an Exception and the execution will be stopped.

If all the policies finish successfully the CMIS service is executed, the `createDocument` method is triggered and the document D1 is created.

The return value of `createDocument` (the Id of the new object created), is added to all the contexts. The state is also updated to POST and then all the policies are executed again.

```
context P1 = {
 Policy Object: P1
 Applied Object: F1
 Policy State: POST
 Return Value: D1
}

context P2 = {
 Policy Object: P2
 Applied Object: F2
 Policy State: POST
 Return Value: D1
}

context P3 = {
 Policy Object: P3
 Applied Object: F3
 Policy State: POST
 Return Value: D1
}
```

This time, the quota policies are not adding any extra logic in the POST state, but the notification policy will use the return value of the service to compose the message to be sent.

3. How to implement a new policy

To be updated in future release.

4. Available policies

To be updated in future release.

Chapter 6. Configuration Parameters

The project is following the standard configuration format described in the deployment process (check the wiki for further information).

All the resources are under `/config`. The files are prefixed with the environment and common files are under `/config/common`. During the deployment the files under `/config` with the prefix of the target enviroment are renamed, removing the prefix and are copied with the common files to the server.

In this chapter you could check the explanation of the entries in the configuration files.

1. Environment files

Configuration files per environment

environment.properties and confidential.properties

In production environments the following keys are divided in two files, one with confidential entries and other with the non-confidential ones. In development and testing environments all the keys are in the file `environment.properties`.

`SERVER_PATH`
: The address of the server.

  For example: `https://localhost:8443`

`ADMIN_TYPE`
: Defines the type of authentication mechanism for the admin module. Currently allowed value is `local`

  `local`: Default Spring security with InMemory authentication provider (admin/admin). No proxy ticket generation. See `repo_admonConfig.xml/admin/local`

`REPO_DB_DRIVER`
: JDBC driver of the main database.

  For example: `oracle.jdbc.OracleDriver`

`REPO_DB_URL`
: URL of the main database.

  For example: `jdbc:oracle:thin:@(DESCRIPTION=(ADDRESS=(PROTOCOL=TCP) (HOST=my-db-server.oranet.net) (PORT=1530)) (CONNECT_DATA=(SERVICE_NAME=repo-service.net)))`

`REPO_DB_USERNAME`
: Username of the main database.

  For example: `username`

`REPO_DB_PASSWORD`
: Password of the main database.

  For example: `******`

`REPO_DB_VALIDATION`
: SQL query to validate the main database connection.

  For example: `SELECT 1 FROM DUAL`

`REPO_JMX_RMI_HOST`
: JMX host

  For example: `localhost`

`REPO_JMX_RMI_PORT`
: Port of the JMX

  For example: `1099`

`REPO_JMX_STATS_COLLECT_ON_START`
: Start collection data of usage on startup.

  For example: `true`

`REPO_JMX_STATS_DEFAULTS_DIR`
: Path to save the usage statistics.

  For example: `/data/applications/repo/confs/stats/`

`NEST_DEV_ENABLED`
: Enable or disable nest-groups-dev authorisation handler.

  For example: `true`

`NEST_TST_ENABLED`
: Enable or disable nest-groups-dev authorisation handler.

  For example: `true`

`TRON_ENABLED`
: Enable or disable tron authorisation handler.

  For example: `true`

`BUILTIN_ENABLED`
: Enable the builtin authentication and authorization handler.

  For example: `true`

`MOCK_AUTHENTICATION_ENABLED`
: Enable or disable the Mock authentication handler.

  For example: `true`

`ECAS_PT_AUTHENTICATION_ENABLED`
: Enable or disable the ECAS proxy ticket authentication handler.

  For example: `true`

`GACA_OPS_AUTHORIZATION_ENABLED`
: Enable or disable the GACA operations authentication handler.

  For example: `true`

`REPO_INDEX_DB_DRIVER`
: JDBC driver for the trasient indexing system.

  For example: `org.h2.Driver`

`REPO_INDEX_DB_FILE`
: Name of the file H2 will save the data.

  Important

  It is not allowed that REPO_INDEX_DB_FILE contains ~ in its value.

  For example: `/data/applications/repo/repo.index`

`REPO_INDEX_DB_URL`
URL of the transient indexing system.

for example: `jdbc:h2:file:${REPO_INDEX_DB_FILE}; DB_CLOSE_DELAY=-1;MODE=ORACLE;TRACE_LEVEL_FILE=1; TRACE_LEVEL_SYSTEM_OUT=0;AUTO_SERVER=TRUE;DB_CLOSE_ON_EXIT=TRUE; MAX_COMPACT_TIME=2000;CACHE_SIZE=32768;MAX_LOG_SIZE=4;LOCK_MODE=0`

`REPO_INDEX_DB_USERNAME`
Username of the transient indexing system.

`REPO_INDEX_DB_PASSWORD`
Password of the transient indexing system.

`REPO_INDEX_DB_VALIDATION`
SQL query to validate the transient indexing database connection.

For example: `SELECT 1`

`REPO_INDEX_H2_SERVER_CACHED_OBJECTS`
Maximum number of objects that H2 can cach per session. If H2 needs to cach more objects than indicated here when dealing with the results of a query, it throws the following exception: org.h2.message.DbException: The object is already closed [90007-164]

Default if not indicated: `512`

`REPO_INDEX_H2_SERVER_RESULT_SET_FETCH_SIZE`
The result set fetch size when using the server mode.

Default if not indicated: `512`

`REPO_INDEX_ENABLED_ATOMIC`
Controls if the indexing process is executed after objects interactions. When true the indexing process adds a job to the queue right after creating a document, updating the stream or delete.

For example: `false`

`REPO_INDEX_ENABLED_BACKGROUND_JOBS`
Controls if the indexing background jobs will be executed.

For example: `false`

`REPO_INDEX_BACKGROUND_JOBS_COMPACT_DATABASE_CRON_EXPRESSION`
Crontab expression to define when the compacting of the indexing database will be performed.

For example: `0 7 * * * ?`

`REPO_INDEX_BACKGROUND_JOBS_CLEAN_ORPHANS_CRON_EXPRESSION`
Crontab expression to define when the delete orphan job will be executed. Index data without a document is cosidered orphan data.

For example: `0 0/15 * * * ?`

`REPO_INDEX_BACKGROUND_JOBS_RETRY_ERRORS_CRON_EXPRESSION`
Crontab defining when the system will execute the indexing of non-indexed documents.

For example: `0 0/5 * * * ?`

`REPO_INDEX_BACKGROUND_JOBS_RETRY_ERRORS_MAX_ATTEMPTS`
: Number of retries during the indexing process.

  For example: `2`

`REPO_INDEX_BACKGROUND_JOBS_RETRY_ERRORS_QUEUE_CAPACITY_THRESHOLD`
: Free remaining capacity below which the index background job is not executed.

  For example: `0.15` indicates that if the background pool has less than 15% of its capacity available the job will not be executed

`REPO_INDEX_THREAD_POOL_SELECTION_LIMIT_SIZE`
: There are two pools for indexing. One for large documents and another one for small ones. This property indicates which is the limit size to consider a document small or large (in bytes).

  For example: `102400` means that documents smaller than 100KB are executed in the pool for small documents

`REPO_INDEX_THREAD_POOL_SMALL_TASKS_QUEUE_CAPACITY`
: Maximum capacity of the queue of the thread pool for small documents. The system will reject new indexing jobs when the queue reach this size. When this is happening, the retry job will take care of this document indexes.

  For example: `300`

`REPO_INDEX_THREAD_POOL_SMALL_TASKS_CORE_SIZE`
: Number of initial indexing threads running in the pool for small documents.

  For example: `2`

`REPO_INDEX_THREAD_POOL_SMALL_TASKS_MAX_SIZE`
: Maximum capacity of the thread pool for small documents.

  For example: `${REPO_INDEX_THREAD_POOL_SMALL_TASKS_CORE_SIZE}`

`REPO_INDEX_THREAD_POOL_SMALL_TASKS_THREAD_PRIORITY`
: Priority for the indexing threads executed by the pool for small documents.

  For example: `3`

`REPO_INDEX_THREAD_POOL_LARGE_TASKS_QUEUE_CAPACITY`
: Maximum capacity of the queue of the thread pool for big documents. The system will reject new indexing jobs when the queue reach this size. When this is happening, the retry job will take care of this document indexes.

  For example: `100`

`REPO_INDEX_THREAD_POOL_LARGE_TASKS_CORE_SIZE`
: Number of initial indexing threads running in the pool for big documents.

  For example: `1`

`REPO_INDEX_THREAD_POOL_LARGE_TASKS_MAX_SIZE`
: Maximum capacity of the thread pool for big documents.

  For example: `${REPO_INDEX_THREAD_POOL_LARGE_TASKS_CORE_SIZE}`

`REPO_INDEX_THREAD_POOL_LARGE_TASKS_THREAD_PRIORITY`
: Priority for the indexing threads executed by the pool for big documents.

For example: `3`

`REPO_INDEX_DB_MAX_ACTIVE`
Maximum active connections of the index database connection pooling. Normally this should match with the number of indexing threads.

For example: `${REPO_INDEX_THREAD_POOL_CORE_SIZE}`

log4j_repo.properties

2. Common files

Common files used in all environments

idx_repo.xml

jmx_access.properties

jmx_password.properties

repo_adminConfig.xml

repo_dbConfig.xml

repo_indexConfig.xml

repo_indexDbConfig.xml

repo_jmxConfig.xml

repo_ormConfig.xml

repo_productConfig.xml

repo_securityConfig.xml

Chapter 7. Deployment Plan

Installation steps

This chapter describes the process of installing the application. The installation process is divided in three parts, actions to be taken before the installation, the intallation of the application and the actions to be taken after the installation.

The pre-installation steps should be done only once and are not covered by the automatic deployment process. This should be executed manually because is altering configuration files of the system.

The installation steps are covered by the standard deployment process. All this part could be execute automatically with the deployment script or jenkins job.

The post-installation steps are administrative interacions with the application running. These operations could be done at any time after the system is started. This covers changing funtional parameters or grant access to users.

1. Pre-installation

1.1. Compile APR and Tomcat native interface

Tomcat is able to be configured using several connectors. By default is using a HTTP connector implemented in Java, to improve the performance, Apache Portable Runtime could be configured.

> Tomcat can use the Apache Portable Runtime to provide superior scalability, performance, and better integration with native server technologies. The Apache Portable Runtime is a highly portable library that is at the heart of Apache HTTP Server 2.x. APR has many uses, including access to advanced IO functionality (such as sendfile, epoll and OpenSSL), OS level functionality (random number generation, system status, etc), and native process handling (shared memory, NT pipes and Unix sockets).
>
> These features allows making Tomcat a general purpose webserver, will enable much better integration with other native web technologies, and overall make Java much more viable as a full fledged webserver platform rather than simply a backend focused technology.
>
> —http://tomcat.apache.org/tomcat-7.0-doc/apr.html

First you need to download the APR package and compile it. Check the Apache APR website to download (http://apr.apache.org/).

```
$ bzip2 -cd apr-1.4.8.tar.bz2 | tar xvf -
$ cd apr-1.4.8
$ export CFLAGS="-m64"
$ ./configure --prefix=/data/applications/repo/test_apr_bin
$ make
$ mkdir /data/applications/repo/test_apr_bin
$ make install
```

And then do the same with the Native runtime. Check the Apache Tomcat website to download (http://tomcat.apache.org/download-native.cgi).

```
$ gzip -cd tomcat-native-1.1.29-src.tar.gz | tar xvf -
$ cd tomcat-native-1.1.29-src/jni/native
$ export CFLAGS="-m64"
$ ./configure --with-apr=/data/applications/repo/test_apr_bin \
 --with-java-home=/usr/jdk/instances/jdk1.7.0_40 \
 --with-ssl=yes \
 --prefix=/data/applications/repo/test_apr_bin
$ make
$ make install
```

Important

The previous instructions are for compiling 64 bits binaries, if you would like to compile for 32bits, don't execute the command `export CFLAGS="-m64"`.

1.2. Configuring JVM parameters

For running properly you need to configure the server to allocate more memory in the permanent generation area and other parameters.

```
export LD_LIBRARY_PATH=/data/applications/repo/test_apr_bin/lib
export JAVA_HOME=/usr/jdk/instances/jdk1.7.0_40
export JAVA_OPTS="-d64 -Xms1024m -Xmx8192m -XX:PermSize=256m \
 -Djava.library.path=/data/applications/repo/test_apr_bin/lib \
 -Djava.awt.headless=true \
 -Djavax.net.ssl.trustStore=/usr/jdk/instances/jdk1.7.0/jre/lib/security/cacerts \
 -Dorg.apache.chemistry.opencmis.stacktrace.disable=true";
export PATH=$JAVA_HOME/bin:$PATH
```

Normally these instructions are saved in the file `$HOME/.profile` to be executed once we log into the system.

2. Installation

2.1. Creating the database

Following the standard structure of the deployment process, the database scripts to create all the needed objects are under `/target/release/db`.

There is an extra file to delete all the database objects. You could use this file in case of uninstall.

2.2. Repository configuration files

Following the standard structure of the deployment process, the configuration files are under `/target/release/config/prod`.

You need to copy these files in the server path `/data/applications/repo/conf`.

A `confidential.properties` file must be manually generated with the sensitive information and saved under `/data/applications/repo/conf`. Check the following template:

Note

Verify that the path `/data/applications/repo/conf` is configured to be under the classpath of Tomcat. To do so, edit the file `/var/apache/tomcat7/conf/catalina.properties`.

You must have an entry with the key `common.loader`, if you haven't the mentioned path, add in the begining and separate with the former value using a coma.

Example 7.1. Apache 7 conf/catalina.properties

```
common.loader=/data/applications/repo/conf,${catalina.base}/lib,
${catalina.base}/lib/*.jar,${catalina.home}/lib,${catalina.home}/lib/
*.jar
```

2.3. Deploy war file

Following the standard structure of the deployment process, the application file is under `/target/release/repo.war`. Copy the war file under `/var/apache/tomcat7/webapps`.

Note

Before starting Tomcat, please delete the folders `/var/apache/tomcat7/work` and `/var/apache/tomcat7/temp`.

3. Post-installation

3.1. Configure the repository instance

By default the repository server is empty, no repositories are created.

To create a repository, enter into the administration panel and select the option "create a new repository".

To enter the repo admin page visit the following link:

http://[url_for_server]:8080/repo/admin

The repo server could host several repositories. Each repository could be configured independently. By default each repository is configured with builtin security handler, review this setting and assign the appropiate handlers.

Check the security chapter for detailed explanation about the security tab.

3.2. ServerAlive page

To verify the application is running properly, a page was put in place to check the availability of the service: http://[url_for_server]:8080/repo/serverAlive.jsp

4. Upgrade from version 1.0.0 to 1.1.0

For upgrading an installation of version 1.0.0 to 1.1.0 check the following points:

- Update the .profile, the JAVA_OPTS value has changed. With SOAP binding, in case of error the stacktrace information is disabled since version 1.1.0.
- Execute DB scripts.
- Rebuild normalized metadata. Metadata is indexed normalized since 1.1.0, existing content MUST be migrated.

4.1. Update .profile - JVM parameters

In version 1.0.0, the SOAP binding was exposing internal details in case of error. Since 1.1.0, the stacktrace in the error message is disabled to avoid exposing implementation details.

To disable the stacktrace add the `org.apache.chemistry.opencmis.stacktrace.disable` variable in the JVM options with any value. See the example below of the .profile content.

```
export LD_LIBRARY_PATH=/data/applications/repo/test_apr_bin/lib
export JAVA_HOME=/usr/jdk/instances/jdk1.7.0_40
export JAVA_OPTS="-d64 -Xms1024m -Xmx8192m -XX:PermSize=256m \
 -Djava.library.path=/data/applications/repo/test_apr_bin/lib \
 -Djava.awt.headless=true \
 -Djavax.net.ssl.trustStore=/usr/jdk/instances/jdk1.7.0/jre/lib/security/cacerts \
 -Dorg.apache.chemistry.opencmis.stacktrace.disable=true";
export PATH=$JAVA_HOME/bin:$PATH
```

4.2. Execute the DB scripts

As part of the standard installation process, several DB scripts are needed to be executed to apply the following changes:

`/releases/1.1.0/db/01_repo_metadataInFullTextSearch.sql`	Metadata is now indexed normalized.
`/releases/1.1.0/db/02_repo_scoreViewUpdate.sql`	Update on the score view.
`/releases/1.1.0/db/03_repo_indexAndScoreViewUpdate.sql`	Changes in the index tables and score view.
`/releases/1.1.0/db/04_repo_changeLogIndex.sql`	New index in change_event to speed up queries in a very active repository.

4.3. Rebuild normalized metadata

Since version 1.1.0 the metadata is indexed normalized. Existing content from version 1.0.0 should be processed to generate the misssing information. To achive this task a tool was created.

1. Go to `/releases/1.1.0/extra/`
2. Modify database connection properties in the file: `config/repo-normalizer.properties`

3. Execute from this directory `java -jar repo-normalizer.jar`

4. Wait until the application finish (message: END normalizer will be printed.)

   Note

   You can ignore this step if you plan to delete the repositories and create a new ones.

5. Upgrade from version 1.1.0 to 1.2.0

For upgrading an installation of version 1.1.0 to 1.2.0 check the following points:

- Execute DB scripts.
- Migrate existing repositories type definitions to CMIS 1.1.
- Migrate existing repositories data to CMIS 1.1.

5.1. Execute the DB scripts

As part of the standard installation process, several DB scripts are needed to be executed to apply the following changes:

`/releases/1.2.0/db/01_repo_idx_property_normalized_value.sql`	Missing index in table property.
`/releases/1.2.0/db/02_repo_secondaryTypes.sql`	New table to store secondary types data.
`/releases/1.2.0/db/03_repo_ancestors_view.sql`	New view with object ancestors used in in_tree() implementation.
`/releases/1.2.0/db/99_repo_generatePWCProperty.sql`	Generates the data of the new property `cmis:isPrivateWorkingCopy`. **Important** If this script is executed before the migration of types definitions, no new rows will be inserted. Execute this script after the types definitions migration.

5.2. Migrate existing repositories type definitions to CMIS 1.1

In CMIS 1.1 were added several new properties and types, this step will upgrade existing repositories.

1. Go to `/releases/1.2.0/migration/`
2. Modify database connection properties in the file: `confidential.properties`
3. Execute from this directory `migrate.bat`
4. Wait until the application finish, a report will me printed with the changes applied

To execute the process Java 7 must be configured in the system.

The script `migrate.bat` was created to be executed under Windows environments, but could be adapted to be executed in other systems like Solaris replacing the path separator `;` by `:`.

See below a typical output for the repository demo:

```
demo
-------------------------------------------------
```

```
> cmis:item [missing]
>>> adding cmis:item
> cmis:secondary [missing]
>>> adding cmis:secondary
> cmis:policy
  .. missing properties [cmis:secondaryObjectTypeIds, cmis:description]
>>> adding property cmis:secondaryObjectTypeIds
>>> adding property cmis:description
  .. cmis:changeToken different
>>> updating property cmis:changeToken
  .. cmis:policyText different
>>> updating property cmis:policyText
> cmis:folder
  .. missing properties [cmis:secondaryObjectTypeIds, cmis:description]
>>> adding property cmis:secondaryObjectTypeIds
>>> adding property cmis:description
  .. cmis:allowedChildObjectTypeIds different
>>> updating property cmis:allowedChildObjectTypeIds
  .. cmis:changeToken different
>>> updating property cmis:changeToken
  .. cmis:parentId different
>>> updating property cmis:parentId
> cmis:relationship
  .. missing properties [cmis:secondaryObjectTypeIds, cmis:description]
>>> adding property cmis:secondaryObjectTypeIds
>>> adding property cmis:description
  .. cmis:targetId different
>>> updating property cmis:targetId
  .. cmis:sourceId different
>>> updating property cmis:sourceId
  .. cmis:changeToken different
>>> updating property cmis:changeToken
> cmis:document
  .. missing properties [cmis:isPrivateWorkingCopy, cmis:secondaryObjectTypeIds, cmis:desc
>>> adding property cmis:isPrivateWorkingCopy
>>> adding property cmis:secondaryObjectTypeIds
>>> adding property cmis:description
  .. cmis:changeToken different
>>> updating property cmis:changeToken
=================================================
demo
-------------------------------------------------
> cmis:policy
> cmis:folder
> cmis:relationship
> cmis:document
> cmis:item
> cmis:secondary
```

5.3. Migrate existing repositories data to CMIS 1.1

New property `cmis:isPrivateWorkingCopy` must be populated with the proper value.

To do so, execute the DB script `/releases/1.2.0/db/99_repo_generatePWCProperty.sql`.

Important

This script generates the data of the new property `cmis:isPrivateWorkingCopy` and must be executed after the previous step (type definition migration).

Chapter 8. Security

1. Introduction

The following sections will provide you with all the relevant information regarding the CMIS 1.0 compliant security model currently implemented by Trade Document Repository. Please note that it is recommended to be familiar with the CMIS 1.0 specification, especially with the following sections under the Domain Model:

- Data Model / Repository / ACL Capabilities
- Data Model / Object Type / Object-Type Attributes
- Data Model / Access Control
- Services / Common Service Elements / Retrieving additional information (ACL, Allowable Actions)
- Services / Common Service Elements / ACLs
- Services / ACL Services

The first section of this document starts explaining the default security configuration for a newly created repository, introducing the key aspects to be considered in order to properly adapt such configuration for the intended use of the repository. In the next section the CMIS services are grouped according to their security restrictions, explaining all the relevant implementation details.

Later, the security handlers are presented, detailing the multiple options available to configure the repository with its custom authentication and authorisation mechanisms. Finally, the last section covers the CMIS ACL model, introducing a complete set of examples that offers a clear idea about how to use it. In addition to that, a detailed description is given about the default ACL strategy offered by TDR.

2. Security configuration by repository

2.1. Default security properties

As stated in the specification a repository can be configured in multiple ways to handle a diversity of security models. In Trade Document Repository, when a new repository is created, the following security configuration options are set by default:

ACL capability	`manage`
ACL Propagation	`propagate`
Permissions	`cmis:read` `cmis:write` (implies `cmis:read`) `cmis:all` (implies `cmis:read` and `cmis:write`)
Permission Mapping	The permission mapping table is set with the default values given by the specification.
Security handlers	Security type: `simple` Authentication: `builtin` Authorisation: `builtin`

In addition to that, the repository's root folder is created using the `cmis:folder` base type. This root folder has the following ACL:

```
<cmis:acl>
 <cmis:permission>
  <cmis:principal>
   <cmis:principalId>cmis:anyone</cmis:principalId>
  </cmis:principal>
  <cmis:permission>cmis:all</cmis:permission>
  <cmis:direct>true</cmis:direct>
 </cmis:permission>
</cmis:acl>
```

2.2. How to configure the repository's security

Once a repository has been created, the administration panel offers the ability to configure all the properties related with the security. Depending on the environment:

https://[url_to_dev_server]:8443/repo/admin
https://[url_to_test_server]:8443/repo/admin
https://[url_to_prod_server]:8443/repo/admin

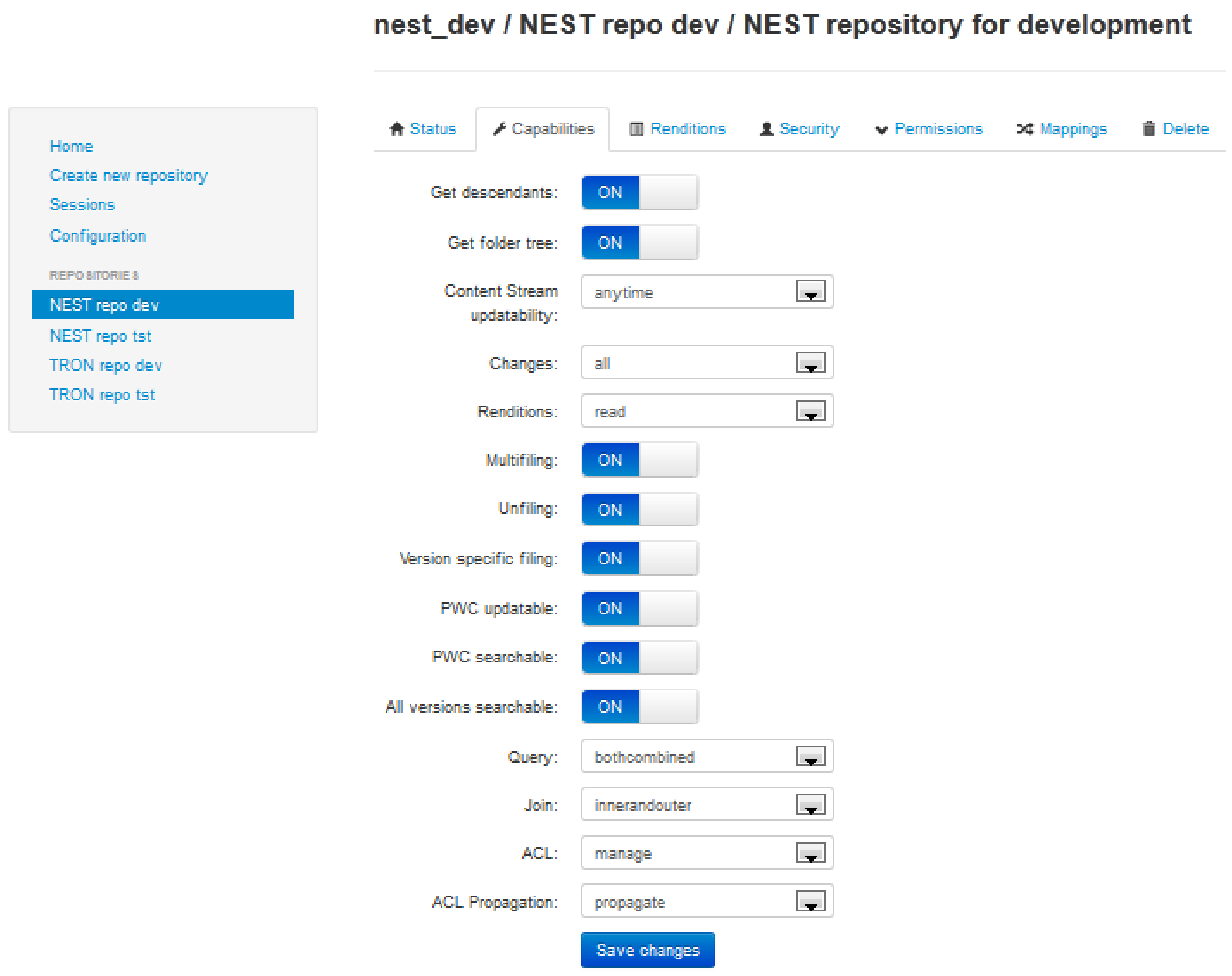
The access to the administration panel is secured with credentials defined in repo_adminConfig.xml. The operations currently defined are:

- `REPO.CreateRepo`
- `REPO.DeleteRepo`
- `REPO.ViewRepoSessions`

- `REPO.ViewRepoSummary`
- `REPO.ChangeRepoCapabilities`
- `REPO.ChangeRepoSecurity`
- `REPO.ChangeRepoMappings`
- `REPO.ChangeRepoPermissions`

2.2.1. Capabilities

In the capabilities tab of a repository, among other relevant capabilities for the repository, the ACL capability and ACL propagation can be changed at any time.



Get descendants:	ON
Get folder tree:	ON
Content Stream updatability:	anytime
Changes:	all
Renditions:	read
Multifiling:	ON
Unfiling:	ON
Version specific filing:	ON
PWC updatable:	ON
PWC searchable:	ON
All versions searchable:	ON
Query:	bothcombined
Join:	innerandouter
ACL:	manage
ACL Propagation:	propagate

a. ACL capability `none`

   This means that the repository does not implements any access control, so any authenticated user can perform any action on any object in this repository. In addition, the ACL services are completely disabled.

b. ACL capability `discover`

   According to the specification this means that the repository implements some kind of access control and that the ACL for an object can be retrieved but not modified.

Implementation Notes

In the case of TDR, the only access control system is derived from the object's ACL. Therefore, if the `discover` ACL capability is enforced for the repository in some point after its creation, this would imply the following:

- If the ACL propagation is `propagate`, then the new objects created under a folder will inherit the pre-existent folder's ACL.

  In the case this change was performed right after the creation of the repository, then every `filed` object would be created having the same ACL as the root folder, i.e.: `cmis:anyone` - `cmis:all`.

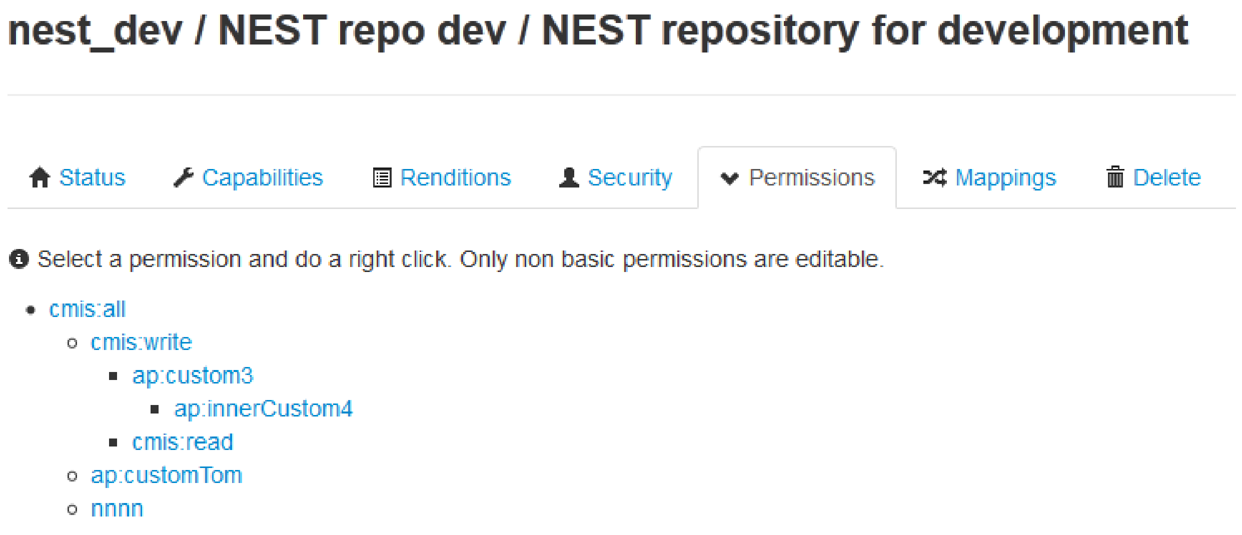
  The difference with ACL Capability none in that case would be just that an `unfiled acl controllable` object couldn't be created since there is no option to set its ACL.

- If the ACL propagation is `objectonly`, then new `acl controllable` objects cannot be created.

For more information about ACL capabilities and propagation, see the ACL section.

2.2.2. Permissions

In the permissions tab of a repository, the set of permissions can be modified. The TDR implementation imposes that the CMIS basic set of permissions (`cmis:read`, `cmis:write` and `cmis:all`) MUST be present and with the same tree structure, being `cmis:all` the only root node. However, this basic set can be extended adding new custom permissions. Remember, a parent permission node means that the permission implies all the descendant permissions.


nest_dev / NEST repo dev / NEST repository for development

Select a permission and do a right click. Only non basic permissions are editable.

- cmis:all
  - cmis:write
    - ap:custom3
      - ap:innerCustom4
    - cmis:read
  - ap:customTom
  - nnnn

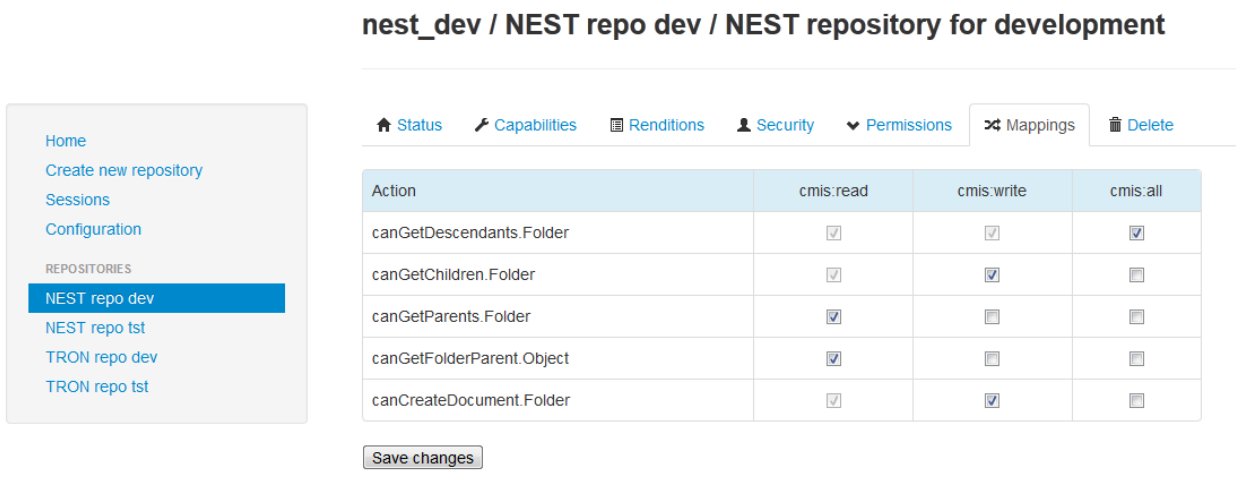
Alternative permission tree

Note

Since the presence of the basic CMIS permissions is mandatory, the supported permissions capability of the repository would be `basic` or `both`, but never `repository`. See the definition of the `getRepositoryInfo` service for more information.

2.2.3. Permission Mappings

In the mappings tab of a repository, an administrator can modify the set of minimum permissions needed to perform the set of actions defined by CMIS.



Action	cmis:read	cmis:write	cmis:all
canGetDescendants.Folder	✓	✓	✓
canGetChildren.Folder	✓	✓	☐
canGetParents.Folder	✓	☐	☐
canGetFolderParent.Object	✓	☐	☐
canCreateDocument.Folder	✓	✓	☐

Please note that the set of permissions related with an action key cannot be empty and, in the case of multiple permissions for the same key, please remember that a user only needs to satisfy one of them in order to be able to perform the related action.

See the CMIS Services Security section to learn how the action keys are used to restrict the use of the CMIS services.

2.2.4. Root Folder's ACL

The root folder’s ACL is the last component to take into account when configuring the repository’s security. As it was stated, during the repository creation the root folder’s ACL is set to `cmis:anyone` - `cmis:all`, meaning that any authenticated user can perform any action on the root folder. This default ACL can be modified in order to restrict the access to the repository using any CMIS client.

Example 8.1.

```
applyAcl(repoId, rootFolderId, add[admin-cmis:all], remove[cmis:anyone-
cmis:all], ...)
```

3. CMIS services security

3.1. Non secured

The following two repository services are non-secured, i.e. any user can access to them.

```
getRepositoryInfos
getRepositoryInfo
```

3.2. Authenticated users only

The rest of the repository services require the user has been authenticated.

```
getTypeChildren
getTypeDefinition
getTypeDescendants
```

Additionally, the discovery services only require the user to be authenticated. However, in this case the service response will be determined according to the user authorization.

```
query
getContentChanges
```

Note

The query service returns only the objects that the user has access to their properties (`CAN_GET_PROPERTIES`). Additionally, regarding to the full text search, then expressions using the clause `contains` only will return true for the objects the user has access to their content stream (`CAN_GET_CONTENT_STREAM`).

3.3. Authorized users only

Following is the detailed list of the current supported services that are restricted according to the user authorization, i.e. the set of user's allowable actions determined by both, the object's ACL and the repository's Permission Mapping. As a general example,

Example 8.2.

Given an empty folder X ...

- With the following's ACL
  `[user1 / cmis:write], [cmis:user2 / cmis:all]`
- Having the repository's Permission Mappings
  `[canDelete.Object / cmis:all], [canCreateDocuemnt.Folder / cmis:write]`
- Having the repository's Permissions tree
  `[cmis:all > cmis:write > cmis:read]`

Then:

- The user1 cannot delete the folder X, because `cmis:write` do not satisfies `cmis:all`.
- The user2 can delete the folder X.
- Both, the user1 and user2, can create a document inside the folder X, because `cmis:write` and `cmis:all` satisfies the minimum permission `cmis:write`.

3.3.1. Navigation Services

`getChildren`
Action: `CAN_GET_CHILDREN`

`getFolderParent`
Action: `CAN_GET_FOLDER_PARENT`

`getObjectParents`
Action: `CAN_GET_OBJECT_PARENTS`

`getCheckedOutDocs`
Not yet implemented

`getDescendants`
Action: `CAN_GET_DESCENDANTS`

`getFolderTree`
Action: `CAN_GET_DESCENDANTS`

Note

The `getDescendants` and the `getFolderTree` do not include the contents of the subfolders where the user has no access to (`CAN_GET_DESCENDANTS`), and also they do not include the documents the user has no access to its properties (`CAN_GET_PROPERTIES`).

3.3.2. Object Services

`createDocument`
Action: `CAN_CREATE_DOCUMENT`

`createDocumentFromSource`
Actions: `CAN_CREATE_DOCUMENT`, `CAN_GET_PROPERTIES` and `CAN_GET_CONTENT_STREAM`

`createFolder`
Action: `CAN_CREATE_FOLDER`

`createPolicy`
Action: `CAN_CREATE_DOCUMENT`

`createRelationship`
Action: `CAN_CREATE_RELATIONSHIP`

`getAllowableActions`
Action: `CAN_GET_PROPERTIES`

`getObject`
Action: `CAN_GET_PROPERTIES`

`getObjectByPath`
Action: `CAN_GET_PROPERTIES`

`getProperties`
Action: `CAN_GET_PROPERTIES`

`getRenditions`
Action: `CAN_GET_PROPERTIES`

`getContentStream`
: Action: `CAN_GET_CONTENT_STREAM`

`updateProperties`
: Action: `CAN_UPDATE_PROPERTIES`

`moveObject`
: Action: `CAN_MOVE_OBJECT`

`setContentStream`
: Action: `CAN_SET_CONTENT_STREAM`

`deleteContentStream`
: Action: `CAN_DELETE_CONTENT_STREAM`

`deleteObject`
: Action: `CAN_DELETE_OBJECT`

`deleteTree`
: Action: `CAN_DELETE_TREE`

Notes

- The `createDocument` service is restricted with the specified action when the document is filed under a folder. If no parent folder is specified, then the document will be created only if the user is authenticated and the `Unfiling` capability is enabled for the repository.
- The authorization for the `createDocumentFromSource` service requires to meet the following conditions:
  - The `CAN_CREATE_DOCUMENT` allowable action for the parent folder. In the case of an unfiled document, the same restriction is applied as in the previous service.
  - The `CAN_GET_PROPERTIES` allowable action for the source document.
  - In the case the source document has a content stream, then the `CAN_GET_CONTENT_STREAM` allowable action for the source document.
- The `createPolicy` service is restricted intentionally with the action `CAN_CREATE_DOCUMENT`, with the same restrictions regarding an unfiled policy.
- The `moveObject` service may imply to change the ACLs of the moved document, and in the case of moving a folder, the moved subtree. To see a complete description of the restrictions and consequences of moving an object, please go to ACL section.
- In the case of deleting a folder, it will only be deleted in the case of an empty folder. In other case, the `deleteTree` service should be used.

3.3.3. Filing Services

Not yet implemented.

3.3.4. Versioning Services

`checkOut`
: Action: `CAN_CHECK_OUT`

`cancelCheckOut`
: Action: `CAN_CANCEL_CHECK_OUT`

`checkIn`
: Action: `CAN_CHECK_IN`

`getObjectOfLatestVersion`
: Action: `CAN_GET_PROPERTIES`

`getPropertiesOfLatestVersion`
: Action: `CAN_GET_PROPERTIES`

`getAllVersions`
: Action: `CAN_GET_ALL_VERSIONS`

Notes

- For the functions `getObjectOfLatestVersion` and `getPropertiesOfLatestVersion` the action restriction is applied to the last version.
- For the function `getAllVersions` The action restriction is applied to the whole version series.

3.3.5. Relationship Services

Not yet implemented.

3.3.6. Policy Services

Not yet implemented.

3.3.7. ACL Services

`getAcl`
: Action: `CAN_GET_ACL`

`applyAcl`
: Action: `CAN_APPLY_ACL`

4. Security handlers

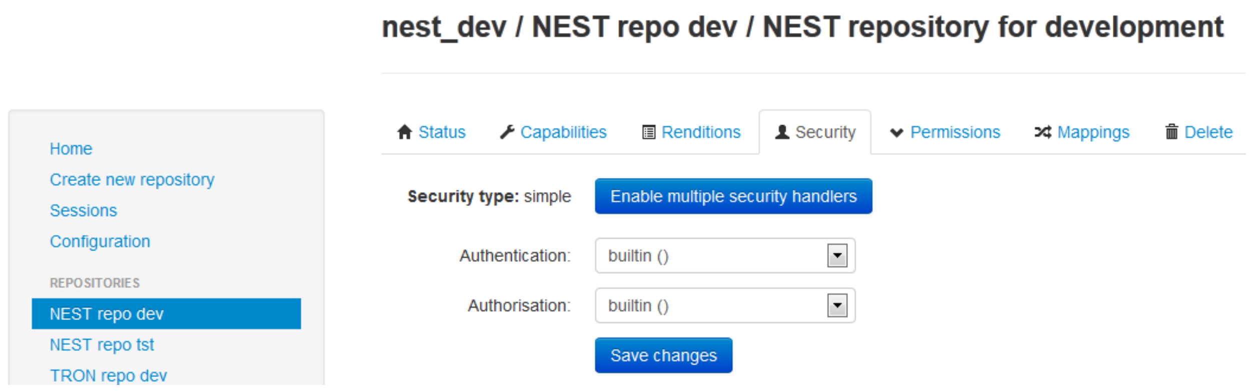
TDR defines two groups of security handlers: the authentication handlers, responsible for authenticating the users that tries to connect to a repository: and authorisation handlers, responsible for determining the principal ids related with the authenticated user and for resolving if the authenticated user is an `admin` user for the repository.

4.1. Admin users

If an authenticated user is authorised as an `admin` user, then no authorisation restrictions are applied. In other words, an `admin` user can perform any CMIS action in the same way like it has a `cmis:all` permission granted for every object in the repository.

4.2. Security Handlers by repository

The complete set of security handlers (at server level) are defined by the server's configuration and cannot be modified in runtime. However, the set of security handlers associated to a specific repository can be modified using the administration panel.



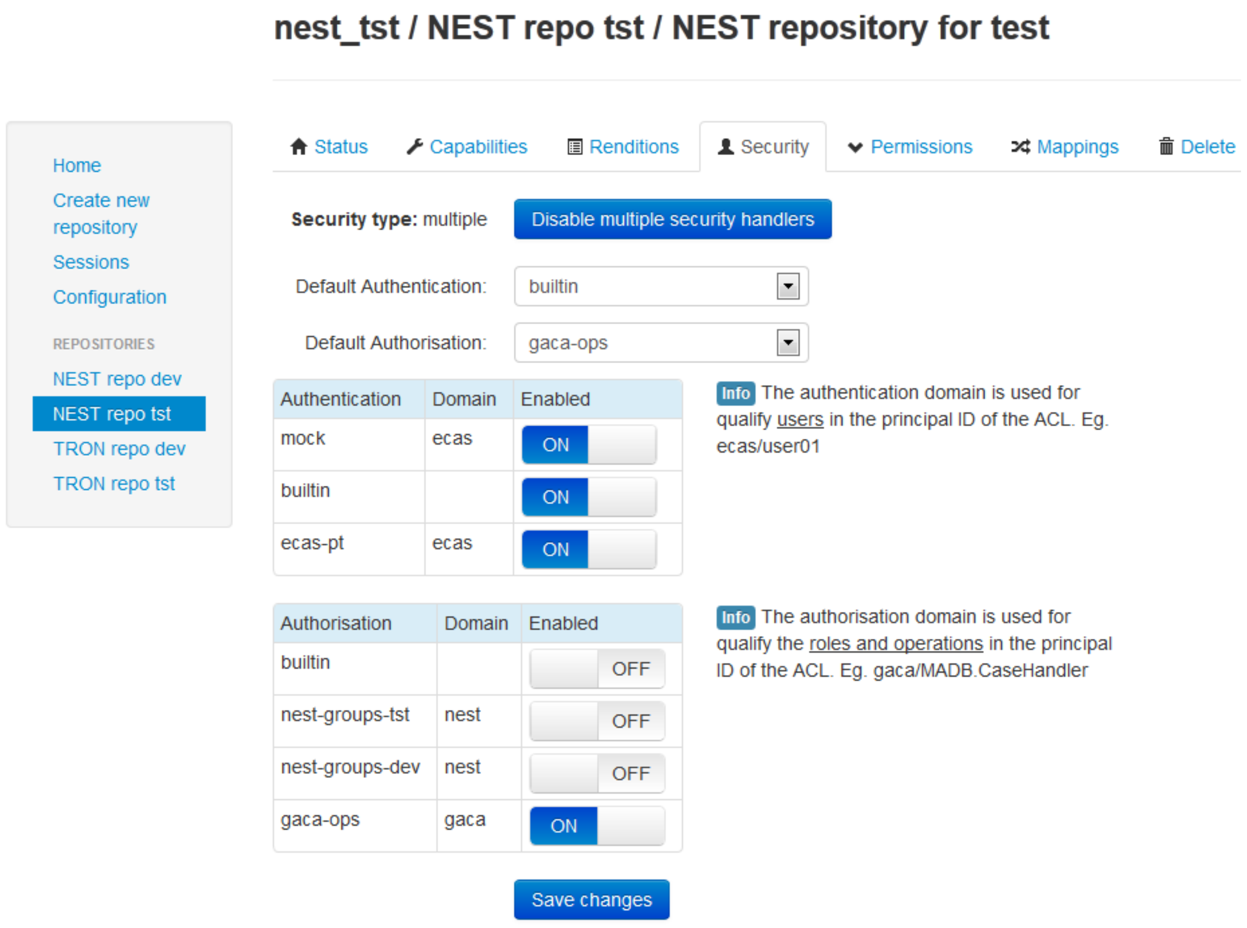
Security type: simple

Authentication: builtin ()

Authorisation: builtin ()

4.3. Simple vs. Multiple

Two security types are available for a repository in TDR. The simple security type, which implies the use of only one authentication handler and only one authorisation handler in order to access to the repository; and the multiple security type, that enables the use of several authentication and authorisation handlers.



nest_tst / NEST repo tst / NEST repository for test

Security type: multiple

Default Authentication: builtin

Default Authorisation: gaca-ops

Authentication	Domain	Enabled
mock	ecas	ON
builtin		ON
ecas-pt	ecas	ON

Info The authentication domain is used for qualify users in the principal ID of the ACL. Eg. ecas/user01

Authorisation	Domain	Enabled
builtin		OFF
nest-groups-tst	nest	OFF
nest-groups-dev	nest	OFF
gaca-ops	gaca	ON

Info The authorisation domain is used for qualify the roles and operations in the principal ID of the ACL. Eg. gaca/MADB.CaseHandler

Please note that in order to change the repository security type, none specific entry must exists in the repository's ACLs. I.e. the reference to the default principal Ids (`cmis:anyone` and `cmis:anonymous`) are allowed, but any other principal Id cannot be automatically translated from one type of security to the other. This "manual" change would imply a database migration script for both the current ACLs and the repository's security type.

4.4. How the prefixes work for login and ACLs

4.4.1. Simple Security Type

If the security type of a repository is simple, then the user can access directly to the repository using its username and password, also the principal Ids specified in the ACL entries MUST NOT be prefixed.

For example, given a new repository with the builtin handlers and a user with username `user1` and password `pwd1` which has the role `regularUser`, it can connect to the repository using the following credentials:

- Default access to the unique handlers.

Username:	`user1`
Password:	`pwd1`

- Specific access to the unique handlers.

Username:	`builtin/builtin/user1`

Password: `pwd1`

And finally, an ACL entry like `[regularUser – cmis:write]` is applicable to this user.

You can see other examples in this ACL section.

4.4.2. Multiple Security Type

In the other hand, if the security type of a repository is multiple then, in every connection to the repository, the user can choose the authentication and authorisation handlers to be used. In order to do so, the user needs to prefix its username with the names of the desired authentication and authorisation handlers. If no prefix is provided, then the default authentication and default authorisation handler will be used.

Regarding to the principal Ids used in the ACLs for this type of security, every principal Id MUST be prefixed with the domain of the related security handler. This will enable to work at the same time with principal Ids defined by multiple security handlers.

For example, given a repository as the one you can see in the previous picture:

- Default authentication handler: `builtin`.
- Default authorisation handler: `Open as admin`.
- Available authentication handlers: `mock`, `builtin`.

Please note that the prefixes used for the login are the `names` of the security handlers while the prefixes for the ACL entries are the `domains` of the security handlers.

In order to see a complete example of the login and ACL entries using the multiple security type, please go to this ACL section.

4.5. Supported Authentication Handlers

Following is the list of authentication handler types supported by TDR.

4.5.1. Builtin

Using the server configuration files, this handler defines a fixed set of users, specifying the username and password.

This handler can be used to configure the access to the repository for an application that always uses the repository in its own name, not on behalf of a final user. In this way, is similar to the connection to a database where a read-write connection is commonly used whereas a read only connection can also be configured.

Several builtin handlers can be configured providing separate users for different repositories. However, currently the only builtin handler configured is for testing purposes.

4.5.2. Mock

Intended only for test purposes this authentication handler does not perform any authentication allowing the access to every user.

4.6. Supported Authorisation Handlers

Following is the list of authentication handler types supported by TDR.

4.6.1. Builtin

Using the server configuration files, this handler defines a fixed set of roles for a fixed set of users, specifying also whether the user is an admin user.

This handler can be used to configure the access to a subset of authenticated users. For example, it can authorise a subgroup of authenticated users to access to a certain repository with `cmis:read` permission.

Several builtin handlers can be configured providing separate groups of roles for different repositories. However, currently the only builtin handler configured is for testing purposes.

4.6.2. DbAuthorizationHandler

This authorisation handler use a database as the source for the user principal Ids. The configuration of these handlers specifies the database connection properties as well as the query to retrieve the principal ids for the user, `authoritiesByUsernameQuery`. This query must return a set of strings based on only one not-named String parameter.

Example 8.3.

SELECT ROLE FROM ROLES_VIEW WHERE USER = ?

Please note that the question mark can appear multiple times but the same value (the username) will be used for all of them. Also, as in the example, it is a good practice to define a view in the source database instead of accessing to concrete tables.

Finally, please note that this authorisation handler does not provide any admin user.

4.6.3. AdminDbAuthorizationHandler

This is an extension of the previous `DbAuthorizationHandler`, which also defines a query to find out if the user is an admin user, `adminByUsernameQuery`. In this case, if the query returns a non-empty result set, the user is considered as an `admin` user.

Example 8.4.

SELECT ROLE FROM ADMIN_ROLES_VIEW WHERE USER = ?

4.6.4. AdminFixedIdsDbAuthorizationHandler

This is an extension of the previous `DbAuthorizationHandler`, which also defines a set of fixed admin principal ids. This way, if one of the principal Ids of the user provided by the `authoritiesByUsernameQuery` matches at least one of the fixed admin principal ids, the user is considered as an `admin` user.

4.6.5. AdminFixedIdsDbAuthorizationHandler

This authorisation handler use a LDAP server as the source for the user principal Ids. The configuration of these handlers specifies the LDAP connection properties as well as the parameters for quering the user groups and, optionally, the parameter to determine if the user is an admin user. The following is the list of parameters to configure this handler:

domain
: The authorization handler domain. Mandatory

java.naming.factory.initial
: The `javax.naming.Context.INITIAL_CONTEXT_FACTORY` property. Optional, default to `com.sun.jndi.ldap.LdapCtxFactory`.

java.naming.provider.url
: The `javax.naming.Context.PROVIDER_URL` property. Mandatory.

java.naming.security.authentication
: The `javax.naming.Context.SECURITY_AUTHENTICATION` property. Optional, default to false.

java.naming.security.principal
: The `javax.naming.Context.SECURITY_PRINCIPAL` property. Mandatory.

java.naming.security.credentials
: The `javax.naming.Context.SECURITY_CREADENTIALS` property. Mandatory.

groupBaseDn
: Base context for the group search. Mandatory.

groupFilterExpr
: Filter expression for the group search. Note that, as maximum, only one parameter should appear in the filter (`{0}`), the user login name. Mandatory.

groupAttribute
: The attribute in the results that contains the value for the user's groups. Mandatory.

groupRecursive
: Whether the parent groups should be resolved or only the directly assign groups. Optional, default to false.

adminUsers
: Whether the handler should resolve admin users. If not, then `isAdmin()` returns always false.

adminBaseDn
: Base context for the admin user search. Mandatory only when adminUsers is true.

adminFilterExpr
: Filter expression for the admin search. Note that, as maximum, only one parameter should appear in the filter (`{0}`), the user login name. Mandatory only when adminUsers is true.

searchTimeLimit
: Time limit for any search. Optional, default to 15 seconds = `15000`.

Example 8.5.

```
<authorizationHandler>
 <name>Name</name>
 <enable>true</enabled>
 <description>Description</description>
 <class>eu.trade.repo.security.impl.LdapAuthorizationHandler</class>
 <properties>
  <domain>test</domain>
  <java>
   <naming>
    <provider>
     <url>ldaps://host:port</url>
    </provider>
    <security>
     <principal>readOnlyUserName</principal>
     <credentials>readOnlyUserPassword</credentials>
    </security>
   </naming>
  </java>
  <groupBaseDn>ou=groups,dc=company,dc=com</groupBaseDn>
  <groupFilterExpr>uniqueMember=uid={0},ou=people,dc=company,dc=com</groupFilterExpr>
  <groupAttribute>entryDN</groupAttribute>
 </properties>
</authorizationHandler>
```

5. ACL

5.1. Introduction

This section covers all the relevant aspects about the access control implementation in TDR. After clearly define the ACL entity in CMIS and explaining how the default CMIS principal ids are supported, several examples are described showing how the repository's security configuration and the ACLs interact in the resolution of the allowable actions over a concrete object.

Finally, the current implementation for the access control is described in detail.

5.2. ACL definition

CMIS defines an object's ACL (access control list) as a set of ACEs (access control entry), where an ACE is defined by the following fields:

- Principal Id: The ACE grants some permissions to the principal identified by this string.
- Permissions: The set of permissions (among the repository's set of permissions) granted to the principal.
- isDirect: Whether this ACE has been applied directly to the object or is inherited from some parent's ACL.

In TDR, the CMIS ACL is translated internally to a flat representation where one ACE with N permissions is transformed in N ACEs with only one permission, as in the following example:

Example 8.6.

This ACE:

```
[ user – {cmis:read, cmis:write} – true ]
```

Is transformed to:

```
[ user – cmis:read – true ]
[ user – cmis:write – true ]
```

Finally, given two ACEs for the same object, they will be equal if and only if the three fields are equal. V.g. the three following ACEs are all different:

```
[ user – cmis:read – true ]
[ user – cmis:read – false ]
[ user2 – cmis:read – false ]
```

5.2.1. Add and remove

From the previous definition, add ACEs to an ACL and remove ACEs from an ACL can be seen as basic set operations:

```
{} + [u–p–t] = {[u–p-t]}
{[u–p-t]} + [u–p-t] = {[u–p-t]}
{[u–p-t]} + [u–p-f] = {[u–p-t], [u–p-f]}
{[u–p-t], [u–p-f]} - [u–p-f] = {[u–p-t]}
{[u–p-t]} - [u–p-f] = {[u–p-t]}
{[u–p-t]} - [u–p-t] = {}
```

```
{} + {[u–p–t]} = {[u–p-t]}
{[u–p-t]} + {[u–p–t]} = {[u–p-t]}
{[u–p-t], [u-p-f]} + {[u2–p-f], [u–p-t]} = {[u–p-t], [u–p-f], [u2–p-f]}
{[u–p-t], u-p-f} - {[u2–p-f], [u-p-t]} = {[u–p-f]}
{[u–p-t]} – {[u–p-f]} = {[u–p-t]}
{[u–p-t]} – {[u–p-t]} = {}
```

5.2.2. Inherit

Additionally, an object only can inherit an ACE if that ACE is present in one of its parents as a direct or inherited ACE.

5.3. Default CMIS principal ids

The following default principal Ids are defined by CMIS:

- `cmis:anyone`

  Any authenticated user will have this among its set of principal Ids. Therefore, it can be used in the ACL to represent a permission granted to any user.

- `cmis:anonymous`

  Currently none of the implemented authentication mechanism provides with anonymous authentication.

- `cmis:user`

  According to CMIS, the repository MAY support the use of `cmis:user` as a reference to the current logged user. Currently this is not supported by TDR and the client MUST specify the current user username.

5.4. Example of Use

For the detailed description of the scenarios used in the examples go to Annex 1.

5.4.1. Simple Security

5.4.1.1. Scenario 1

- Repository: repository01
- Security Handlers: securityHandlers01
- Objects: objects01

The following assertions applied:

- `user1` can access with username: `user1` and password: `pwd1`
- `user1` can access with username: `builtin/builtin/user1` and password: `pwd1`
- `user2` can access with username: `user2` and password: `pwd2`
- `user` can access with username: `builtin/builtin/user2` and password: `pwd2`
- `user3` can access with username: `user3` and password: `pwd3`

- `user3` can access with username: `builtin/builtin/user3` and password: `pwd3`
- `user1` has the following principal Ids: `user1, cmis:anyone, reader`.
- `user2` has the following principal Ids: `user2, cmis:anyone, writer`.
- `user3` has the following principal Ids: `user3, cmis:anyone, reader, owner`.
- All the users can read `object01`.
- `user1` cannot access `object02`. *Note that `test-roles/reader` cannot be applied to `user1` because in simple security the principal Ids are not prefixed.*
- The `user2` and `user3` can do anything with `object01` and with `object02` (write, modify, delete, apply ACL, etc.).

5.4.1.2. Scenario 2

- Repository: repository02
- Security Handlers: securityHandlers01
- Objects: objects01

The following assertions applied:

- `user1` can access with username: `user1` and password: `pwd1`
- `user1` can access with username: `builtin/builtin/user1` and password: `pwd1`
- `user2` can access with username: `user2` and password: `pwd2`
- `user` can access with username: `builtin/builtin/user2` and password: `pwd2`
- `user3` can access with username: `user3` and password: `pwd3`
- `user3` can access with username: `builtin/builtin/user3` and password: `pwd3`
- `user1` has the following principal Ids: `user1, cmis:anyone, reader`.
- `user2` has the following principal Ids: `user2, cmis:anyone, writer`.
- `user3` has the following principal Ids: `user3, cmis:anyone, reader, owner`.
- All the users can read `object01`.
- `user1` cannot access `object02`.
- `user2` can do anything with `object01` (write, modify, etc.) but delete or apply ACL. *Note that in the permission mappings for this scenario, `delete` and `applyAcl` actions require the `cmis:all` permission.*
- The `user2` can do anything with `object02` (write, modify, delete, apply ACL, etc.).
- The `user3` can do anything with `object01` (write, modify, delete, apply ACL, etc.).
- `user3` can do anything with `object02` (write, modify, etc.) but delete or apply ACL. *Note that in the permission mappings for this scenario, `delete` and `applyAcl` actions require the `cmis:all` permission.*

5.4.2. Multiple Security

5.4.2.1. Scenario 1

- Repository: repository02
- Security Handlers: securityHandlers01
- Objects: objects01

The following assertions applied:

- `user1` can access with username: `user1` and password: `pwd1`
- `user1` can access with username: `builtin/builtin/user1` and password: `pwd1`
- `user2` can access with username: `user2` and password: `pwd2`
- `user` can access with username: `builtin/builtin/user2` and password: `pwd2`
- `user3` can access with username: `user3` and password: `pwd3`
- `user3` can access with username: `builtin/builtin/user3` and password: `pwd3`
- `user1` has the following principal Ids: `test-users/user1, cmis:anyone, test-roles/reader`.
- `user2` has the following principal Ids: `test-users/user2, cmis:anyone, test-roles/writer`.
- `user3` has the following principal Ids: `test-users/user3, cmis:anyone, test-roles/reader, test-roles/owner`.
- All the users can read `object01`.
- `user1` and `user3` can read `object02`. *Note that `test-roles/reader` can be applied to `user1` and `user3` because in multiple security the principal Ids are prefixed.*
- None of the users can modify `object01` or `object02`. *Note that `user2`, `reader`, `writer` or `owner` cannot be applied to any of the users.*

5.4.3. Admin users

5.4.3.1. Scenario 1

- Repository: repository03
- Security Handlers: securityHandlers01
- Objects: objects01
- The users access prefixing its username with `builtin/admin/`.

The following assertions applied:

- `user1` can access with username: `builtin/admin/user1` and password: `pwd1`
- `user` can access with username: `builtin/admin/user2` and password: `pwd2`
- `user3` can access with username: `builtin/admin/user3` and password: `pwd3`

- `user1` has the following principal Ids: `test-users/user1, cmis:anyone`.
- `user2` has the following principal Ids: `test-users/user2, cmis:anyone`.
- `user3` has the following principal Ids: `test-users/user3, cmis:anyone`.
- `user1` is an admin user.
- All the users can read `object01`.
- `user1` can do anything with `object02` (write, modify, delete, apply ACL, etc.). *Note that user1 is an admin user*.
- None of the other users can modify `object01` or `object02`. *Note that `user2, reader, writer` or `owner` cannot be applied to any of the users.*

Chapter 9. Configuration

Capabilities detailed explanation

List of capabilities and effect on the repository.

Set of capabilities tested.

To be updated in future release.

Chapter 10. Versioning

Decisions taken in the versioning service:

If a version series has a PWC, only the PWC could be updated (properties and stream). If a version series doesn't have a PWC only the document flagged as latest version could be updated (properties and stream). A document of the version series could be deleted at any time.

Document the validations executed when a document in the version series is deleted. When deleting the latest document, the previous one becomes the latest, in this case could exist another document in the folder with the same name.

Chapter 11. Discovery

Current funtionality available in the discovery service.

Relaxed implementation of the CMIS standard: eg. several contains() and the posibility to use single value operations in multiple properties.

Join is not available

Chapter 12. Extensions

List of extra capabilities that are not covered by CMIS

1. Extension Data

1.1. Index Status

In order to provide information about the indexing status for the document objects the following extension data is added under the TDR's namespace `{http://[namespace_url]/repo}`:

- indexing
  - state Enum:

    `NONE` (Indexation pending)
    `INDEXED` (Indexed)
    `ERROR` (Indexation error)
    `NON_INDEXABLE` (Not indexable)
    `PARTIALLY_INDEXED` (Partially indexed. Word limit reached)
  - tries Integer

Chapter 13. Administration panel manual

To be updated in future release.