Version 0.5

Content Management Interoperability Services

Part I – Introduction, General Concepts, Data Model, and Services

© EMC Corporation, IBM Corporation, Microsoft Corporation. All Rights Reserved.
8/28/2008
Folder Object ........................................................................................................................................... 23
Relationship Object .................................................................................................................................. 26
Policy Object ............................................................................................................................................. 27
Object Type .............................................................................................................................................. 28
Base Object Types .................................................................................................................................... 34
Enumerations ............................................................................................................................................. 38
Property Filter .......................................................................................................................................... 39
Versioning .................................................................................................................................................. 39
Query .......................................................................................................................................................... 42
Relational View .......................................................................................................................................... 43
CMIS SQL .................................................................................................................................................. 44
Services ....................................................................................................................................................... 51
Common Service Elements ....................................................................................................................... 51
Common exceptions .................................................................................................................................... 51
Other exceptions ......................................................................................................................................... 51
Property Filters ......................................................................................................................................... 52
Repository Services .................................................................................................................................... 52
getRepositories ........................................................................................................................................ 52
getRepositoryInfo ...................................................................................................................................... 53
getTypes ..................................................................................................................................................... 54
getTypeDefinition ..................................................................................................................................... 54
Navigation Services .................................................................................................................................... 55
getDescendants ......................................................................................................................................... 55
getChildren ............................................................................................................................................... 56
getFolderParent ......................................................................................................................................... 57
getObjectParents ....................................................................................................................................... 58
getCheckedoutDocuments ....................................................................................................................... 58
<table>
<thead>
<tr>
<th>Service Type</th>
<th>Functions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Object Services</td>
<td>createDocument, createFolder, createRelationship, createPolicy, getContentStream, updateProperties, moveObject, deleteObject, deleteTree, setContentStream, deleteContentStream</td>
</tr>
<tr>
<td>Multi-filing Services</td>
<td>addObjectToFolder, removeObjectFromFolder</td>
</tr>
<tr>
<td>Discovery Services</td>
<td>query</td>
</tr>
<tr>
<td>Versioning Services</td>
<td>checkOut, cancelCheckOut, checkIn, getPropertiesOfLatestVersion, getAllVersions, deleteAllVersions</td>
</tr>
<tr>
<td>Relationships Services</td>
<td></td>
</tr>
</tbody>
</table>
(c) 2006-2008 EMC Corporation (EMC), International Business Machines Corporation (IBM), and Microsoft Corporation. All rights reserved.

Upon adoption as a standard, permission to copy and display the "Content Management Interoperability Services" Specification, in any medium without fee or royalty is hereby granted, provided that you include the following on ALL copies of the "Content Management Interoperability Services" Specification, or portions thereof, that you make:

1. A link or URL to the "Content Management Interoperability Services" Specification at this location: <tbd>.

2. The copyright notice as shown in the "Content Management Interoperability Services" Specification.

Upon adoption as a standard, EMC Corporation (EMC), International Business Machines Corporation (IBM), and Microsoft Corporation (collectively, the "Authors") each agree to grant you a royalty-free license, under reasonable, non-discriminatory terms and conditions to their respective patents to the extent that the Authors that they deem necessary to implement the "Content Management Interoperability Services" Specification.

THE "CONTENT MANAGEMENT INTEROPERABILITY SERVICES" SPECIFICATION IS PROVIDED "AS IS," AND THE AUTHORS MAKE NO REPRESENTATIONS OR WARRANTIES, EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, NON-INFRINGEMENT, OR TITLE; THAT THE CONTENTS OF THE "CONTENT MANAGEMENT INTEROPERABILITY SERVICES" SPECIFICATION ARE SUITABLE FOR ANY PURPOSE; NOR THAT THE IMPLEMENTATION OF SUCH CONTENTS WILL NOT INFRINGE ANY THIRD PARTY PATENTS, COPYRIGHTS, TRADEMARKS OR OTHER RIGHTS.

THE AUTHORS WILL NOT BE LIABLE FOR ANY DIRECT, INDIRECT, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF OR RELATING TO ANY USE OR DISTRIBUTION OF THE "CONTENT MANAGEMENT INTEROPERABILITY SERVICES" SPECIFICATION.

The name and trademarks of the Authors MAY NOT be used in any manner, including advertising or publicity pertaining to the "Content Management Interoperability Services" Specification or its contents without specific, written prior permission. Title to copyright in the "Content Management Interoperability Services" Specification will at all times remain with the Authors.
INTRODUCTION

The Content Management Interoperability Services (CMIS) standard will define a domain model and set of bindings, such as Web Service and REST/Atom that can be used by applications to work with one or more Content Management repositories/systems.

The CMIS interface is designed to be layered on top of existing Content Management systems and their existing programmatic interfaces. It is not intended to prescribe how specific features should be implemented within those CM systems, nor to exhaustively expose all of the CM system’s capabilities through the CMIS interfaces. Rather, it is intended to define a generic/universal set of capabilities provided by a CM system and a set of services for working with those capabilities.

NOTATION (RFC 2119)

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in BCP 14, [RFC2119], as scoped to those conformance targets.

VISION / MISSION

According to Forrester and other major consultants, most companies have multiple content management systems from different vendors. It is common to see large companies running IBM Content Manager, IBM FileNet P8, EMC Documentum in the data center and Microsoft SharePoint on departmental servers. Historically content management systems were purchased for specific application uses and this led to islands of incompatible systems. The lack of a standard interface to content management systems made it difficult to integrate content from multiple repositories into a single application such as a portal, CRM system, or office desktop. It also made it difficult for ISVs and integrators to interface to multiple content management systems consistently.

The objective of the CMIS standard is to define a common content management web services interface that can be implemented by content repositories and enable interoperability across repositories. These capabilities and interfaces will not match every existing content management system and MAY require some level of change to existing products, at least in terms of conforming existing interfaces to those defined here. However, it is an explicit goal that CMIS will NOT require major product changes or significant data model changes like other standards such as JSR 170 have required.
DESIGN OBJECTIVES

This section will outline the goals and non-goals (i.e. goals that are NOT in scope for the initial version of this standard).

GOALS

ALLOW APPLICATIONS TO BE BUILT THAT CAN TARGET ONE OR MORE ECM REPOSITORIES UNIFORMLY FOR BASIC CONTENT SERVICES

The CMIS standard will expose core/common ECM repository capabilities in an intentionally generic way. These will allow for applications to be constructed that can work with content residing in one or more ECM repositories, without having to understand implementation differences between the individual repositories or worrying about interface inconsistencies between the repositories.

PROVIDE A SET OF BASIC SERVICES ENABLING RICHER ECM APPLICATIONS AND USE CASES

While most/all of the capabilities that will be exposed via CMIS generally fall into the “core/basic” functions of an ECM repository, the goal of this standard is to ensure that ECM applications can be built on top of the CMIS interfaces that enable richer/business critical applications and use cases, like Business Process Management and Electronic Discovery. Because those application use cases have been under consideration through the CMIS design process, CMIS will enable ECM applications to focus on solving business logic problems at the application-level without worrying about the implementations of specific ECM repositories — and the capabilities that have been included in CMIS are intended to support the use cases listed in the “Use Cases” section.

ALLOW FOR LOOSE COUPLING BETWEEN AN ECM APPLICATION AND A REPOSITORY

By providing a services-oriented architecture for interacting with an ECM repository, ECM applications can use CMIS to be loosely-coupled to individual repositories, rather than more tightly integrated. This will make it simpler for (a) applications to use CMIS interfaces “a la carte” rather than having to having to invoke the full-set of CMIS interfaces, and (b) allow applications to be built in a Services-Oriented-Architecture.

NON-GOALS

The following are explicit non-goals of CMIS.

EXPOSE ALL (OR SUBSTANTIALLY ALL) THE CAPABILITIES OF AN INDIVIDUAL ECM REPOSITORY

It is certainly the case that most ECM repositories in existence today offer more capabilities and features than those that are defined in CMIS.

The CMIS standard is intentionally taking a “least common denominator” approach to ensure that the set of capabilities it covers can be supported by all major ECM systems — which explicitly means that those
unique/differentiated features offered only by one or a few modern ECM systems, or those whose implementation in each ECM system varies widely, will not be covered as part of this standard. (For example, administration and management are not included in CMIS 1.0 for this reason.)

It is by no means the intent of this standard to specify ALL of the capabilities that a customer MAY want in an ECM repository. In future versions of the standard, we MAY consider additional capabilities for inclusion in CMIS as they become more universally-implemented. However it is likely to always be the case that individual ECM systems will want to offer capabilities beyond those defined in CMIS.

BE THE “PRIMARY” MECHANISM FOR INTERACTING WITH AN ECM REPOSITORY

As described in the previous non-goal – most ECM systems will offer features and capabilities that are not covered by CMIS, and applications and customers will certainly want to take advantage of these capabilities in many scenarios.

As such, CMIS fully recognizes that most ECM repositories will continue to expose proprietary interfaces that allow for interacting with all aspects of a particular ECM system, and that in many use cases it will be appropriate for customers to want applications that are at least in part tightly-coupled to those proprietary interfaces. (For example, for administration of a particular repository, or for applications that only need to interact with a single repository.)

It is the intent of CMIS to provide a generic “secondary” set of universal interfaces for an ECM repository to expose, to enable interoperable applications to be built – even if those applications are not using every capability of each individual repository.

TARGET VS. NON-TARGET USE CASES

To ensure that CMIS meets the previously-stated goal of allowing for the creation of rich ECM applications using this standard, here are the list of representative use cases that are accounted for in the CMIS capabilities, as well as some that were NOT targeted for CMIS v1.

These are categorized into the following groups:

1) **“Core ECM” use cases that are directly supported by the CMIS interfaces**: These are the fundamental ECM use cases that are common to most/all ECM applications today.

2) **ECM applications & use cases that can be built on top of the CMIS interfaces**: These are examples of applications that can use the basic capabilities exposed via the CMIS interfaces to provide higher-level solutions to customer problems.

3) **Out-of-scope use cases for CMIS 1.0**: These are use cases/applications that, although in many cases are important for ECM scenarios, have been deemed “out-of-scope” for the initial version of the CMIS standard – due to either particularly unique or complex requirements and/or because there has been less of a widespread need for achieving cross-repository interoperability for those use cases.
   - And while many of the requirements for these use cases are already covered in the in-scope use cases, where there are additional requirements to either enable or optimize for a use case in this group we have deferred those requirements to a future version of the CMIS standard.
CORE ECM USE CASES

COLLABORATIVE CONTENT CREATION

A set of users wish to work collaboratively to create one or more documents or web pages. The set of users involved in the collaboration is constrained and users must authenticate to the system so that their actions can be recorded. Multiple users will edit the same content, generally one at a time, sometimes directed by a structured business process. As content is edited older versions of the content MAY be stored and are available for access. References to content to be edited are emailed between the parties collaborating.

PORTALS

A portal is used to bring together different types of enterprise information for employees, customers, customer services agents, and other users. It provides an integrated, often personalized, user interface that improves productivity by integrating disparate information sources on the desktop. One of the most common uses of portals is to integrate employee HR information such as health benefits and forms, travel expense reporting and documentation, education and training, and anything else an employee MAY need. The information integrated into a portal often includes both database data as well as documents.

MASHUPS

A mashup is a web-site or application that seamlessly combines content from multiple sources into a new application. Within enterprises it is often an ad-hoc application quickly implemented to solve a specific business problem. Mashups require simple, easy to use Web Service interfaces for accessing & manipulating content that can be discovered and composed with the data from other sources. Many ECM repositories contain very valuable metadata information and this will often be used by mashups directly.
ECM APPLICATIONS & USE CASES THAT CAN BE BUILT ON TOP OF THE CMIS INTERFACES

WORKFLOW & BUSINESS PROCESS MANAGEMENT

Many content-centric applications are constructed and executed via formal business process management (BPM). Some of the steps in the BP are “manual” steps where a user is asked to perform some actions on or with some content and other steps are “automatic” where the BP invokes another computer program to execute some steps on or with some content. Steps MAY include branching (one activity is followed by two or more activities to be executed in parallel), decision points (choosing one of many possible next steps, generally based on the state of some content) and collection points (where a step waits for two or more feeding paths to complete). Interaction points with the content repository occur both during business process design time and during BP execution.

ARCHIVAL APPLICATIONS

Many business applications produce high volumes of documents per day, often in a print format such as PDF or PostScript. These documents need to be captured from other business systems, indexed, and stored into archival repository such that they can be easily retrieved by end users as needed for business functions.

COMPOUND / VIRTUAL DOCUMENTS

Content repositories are often used by publishing and collaborative applications that enable multiple users to concurrently develop documents. Examples of these documents are maintenance manuals, engineering documents, investment reports, product manuals and documentation and healthcare and life sciences publications. Another key requirement is to enable content, such as an image, chart, or block of text to be repurposed across multiple documents. From a user’s perspective, the end product is a single document but from the document management system’s perspective, it is comprised of multiple parts that have relationships. When the document is printed or distributed, it is usually packaged into a single PDF or print file. However, when the compound document is published or edited, the multi-part structure is preserved so the entire file does not have to be downloaded at once.

ELECTRONIC LEGAL DISCOVERY

Legal discovery is the process of finding evidence for a civil or criminal legal case. Specific information needs to be located, secured, and analyzed with the intent of using it as evidence – and it is vital that electronic discovery tools can use CMIS interfaces to locate potentially relevant content.

Note: Legal discovery is a broad area that puts many requirements on an archive or repository. The scope of this use case is the “searching” for information that can be used in a legal discovery process. This use case does not cover how legal documents are stored or how those documents are secured via legal hold.

OUT-OF-SCOPE USE CASES FOR CMIS 1.0
Content Management repositories are the storage location of many of an organization's most important business records. In order to appropriately manage those records, a company's legal or Records Management department needs to be able to:

1) Define retention schedules that determine how long each type of record in the organization (regardless of what repository it's living in) will be kept / when it will be dispositioned.
2) Ensure that content in repositories is classified appropriately so that those retention schedules can be enforced automatically & consistently.
3) Have a mechanism for dealing with litigation hold orders – when a litigation in the organization arises, the legal department needs to be able to locate all items that are potentially relevant to that litigation, and override its normal disposition appropriately.

Especially in organizations with heterogeneous content management systems, records managers will need a single centralized Records Management (RM) application to let them carry out these functions.

**Digital Asset Management**

One type of content often stored in ECM systems are "digital assets" – this term generally refers to content objects that change very infrequently, are used/incorporated often into other content objects (E.g. a corporate logo). Additionally, these assets are often have large byte streams, and the system usually stores multiple renditions of the asset for use in different contexts. (E.g. the ECM system may store 3 different renditions of the corporate logo; a large uncompressed image file for use in print advertising, a small vector graphic for use in electronic presentations, a highly-compressed image file for use in web pages.)

Organizations generally want one (or at least a small number of) applications that are used to browse/find/use those digital assets. These applications need to present a uniform interface for browsing or searching for digital assets (regardless of source repository), presentation logic that shows only appropriate renditions of items given a user's context (e.g. a search result should return only 1 result “asset”, not 8 results that are all renditions of an asset), and allow consumer applications to retrieve renditions in a stream-able/fault-tolerant way.

**Web Content Management**

One type of application on top of an ECM repository, is Web Content Management. Supporting this use case is out of scope for the initial deliverable.

**Subscription / Notification Services**

A user may wish to either be notified in some manner when content is changed or to receive feeds of updated/new content. ECM applications may allow users to register for the system to perform specific fulfillment actions (e.g. send notification, send RSS/ATOM feed) when a specific event (e.g. checkin, checkout...) occurs to one or more documents or folders. The user can define terms of the notification/feed such as criteria for notification (e.g. expiration date, document lifecycle state). The actions defined for the notification/feed are constrained by the user's permissions.
HOW DOES CMIS RELATE TO OTHER STANDARDS EFFORTS?

Since CMIS is certainly not the first attempt to standardize some of the core elements of ECM repositories, it is worth briefly describing how it differs from those other standardization efforts.

(This section does NOT attempt to summarize recap what those particular standards efforts do, other than how they differ from CMIS – for more information about each standard, see the reference links below.)

Content Repository for Java – JSR 170/283


The approach and goals of CMIS differ from those of JSR 170/283 in the following important ways:

1) **JSR is a Java-language only standard.**

2) **The JSR standard requires a particular type of implementation for ECM repositories:** Whereas CMIS restricts itself to specifying only generic/universal concepts for ECM constructs like Documents and Object Types that could be layered on most existing ECM implementation, the JSR standard requires a highly-specific & feature-completion implementation of a repository. This structure may not be appropriate for many types of applications, or efficiently layered on existing ECM repositories.

Atom Publishing Protocol -- APP


The scope of APP is targeted towards supporting Wikis and Blogs and in particular the ATOM format. CMIS focuses on providing a fairly broad set of core enterprise content management services for interacting with an ECM repository. APP is focused strictly on an HTTP-based protocol for editing particular kinds of web resources (such as Weblogs, Wikis, and online journals). Editing individual content objects is certainly within the scope of CMIS – and in this regard we will look to leverage existing APP conventions/mechanisms for overlapping areas of functionality where appropriate in our protocol bindings.

HTTP Extensions for Distributed Authoring -- WebDAV


WebDAV is an extension to the HTTP/1.1 protocol to allow clients to perform remote web content operations. (In this regard, it's scope is somewhat similar to APP's).

Many of the initial goals of WebDAV were fundamentally similar to CMIS' (for example, providing a way to read & write documents and their properties, and provide a versioning model for interacting with content over HTTP.)

However, the DAV standard is lacking in a few key areas that are critical for the use cases listed above, including:
The ability for a repository to expose a heterogeneous set of Object Types within a single repository or folder.

The ability for a user to issue a “query” to retrieve & interact with a collection of objects, regardless of their logical storage location.

Relationships between objects other than their logical storage hierarchy.

Furthermore, the WebDAV standard also includes several features that are not commonly implemented in modern/mainstream ECM repositories, such as shared and exclusive locking of an object (section 6.1).

Finally, the WebDAV standard is very tightly integrated to the HTTP protocol – meaning that mapping it onto dissimilar protocols in the future would be a significant challenge.

As a result, the authors of the CMIS specification decided that rather than design CMIS as an extension to the WebDAV protocol, CMIS would instead be created as a new protocol that could then be bound to multiple protocols, and that wouldn’t need to worry about backwards compatibility with the WebDAV protocol.
GENERAL CONCEPTS

OVERVIEW OF CMIS 1.0

While the next few sections will properly define all of the details of CMIS 1.0, the following primer may be helpful in terms of understanding the basic concepts underlying a repository as defined by CMIS 1.0:

1. A Repository is an abstraction that is defined independent of the underlying storage mechanisms and the standard will define interfaces to the content repository abstraction.

2. CMIS defines four base “types” of objects that exist within a Repository:
   a. Documents represent individual content objects in the repository. A Document may or may not include one content-stream.
   b. Folders represent organizational containers in which Documents (or other folders) can be stored.
   c. Relationships represent loose relationships between exactly 2 objects (documents or folders) in the Repository.
   d. Policies represent administrative policies that may be applied to objects.

3. The Repository can define additional Object Types for any of the above-listed type of objects. An Object Type specifies the schema of Properties that are allowed or required for the object.

4. CMIS exposes services for:
   a. Discovering Object Type definitions and other Repository information (including which optional capabilities are supported by a particular Repository.)
   b. Creating, Reading, Updating, and Deleting objects.
   c. Filing Documents into zero, one, or more Folders — if the repository supports the optional multi-filing capability (see #6).
   d. Navigating and traversing the hierarchy of folders in the Repository.
   e. Creating versions of Documents and accessing a Document’s version history.
   f. Querying a repository to retrieve one or more objects matching user-specified search criteria, including full-text search queries.

5. Document objects can be versioned. (Folder, relationship, and policy objects are not versioned.)
   a. All methods for referring/retrieving a Document can specify whether they refer to a specific version of a Document, or should always retrieve the latest version.

6. A CMIS Repository has the option of supporting multi-filing of Documents into zero, one, or more than one folder concurrently.
   a. Folders can never be multi-filed.
   b. The Repository’s level of support for multi-filing will be exposed to applications through the Repository service.

GENERAL DESIGN TENETS FOR CMIS
The following points summarize general design concepts used by CMIS.

**Protocol bindings (REST & SOAP)**

It is the intent of the CMIS standard to support eventual binding to multiple protocols – specifically Simple Object Access Protocol (SOAP) and Representational State Transfer (REST/Atom) based on Atom/APP for CMIS 1.0.

As such, this standard document will first define a domain model which includes data model and service model, and then define specific bindings of those services to both the REST/Atom and SOAP protocols. The core functionality will be maintained across all bindings. However, the form of the services might be different in each binding.

A conformant implementation MUST implement the Web Service binding and MUST implement the REST binding.

Additionally, future versions of this standard may add additional bindings.

**Extensibility**

Conformant implementations MUST provide the CMIS services with the same semantics as specified. However, implementations may provide additional implementation-specific information through the services where appropriate. This can be done via the Repository Info, additional properties on objects or via implementation-specific namespaces on the resulting messages.

**AREAS THAT ARE OUT-OF-SCOPE FOR CMIS 1.0**

The following issues are explicitly excluded from consideration from this version of CMIS. Subsequent versions of the standard may choose to incorporate these areas.

**Administration / Configuration**

CMIS explicitly does not provide any services to allow an ECM application to administer or configure an ECM repository (for example by editing Object Types or modifying other configuration state.)

It is expected that CMIS 1.0-compliant repositories will continue to expose these types of operations as applicable via their proprietary interfaces.

**Security (authentication & authorization)**

CMIS explicitly does not specify any specific security settings, as described in the two sections below:

**AUTHENTICATION**

It is left up to each ECM repository, the protocols used by applications to interact with the repository via CMIS services, and the applications using those protocols, to determine how the identities of users will be determined and verified.

The CMIS services all assume that the repository is able to identify users through those protocols.
**Authorization**

CMIS provides no mechanism for an application to specify what operations a particular user should be able to perform on a given Object in a Repository, via the CMIS interfaces or otherwise.

It is left up to the Repository to determine based on the identity of the user invoking a particular CMIS service (however that authentication of identity occurs), whether or not the method invocation should succeed or fail.

CMIS will expose a single service to allow an application to determine what CMIS-defined operations can be performed in the current (user’s) context for a particular object. Again, it is left up to the Repository to determine which actions should be returned as “allowed” based on its internal authorization model.

**Localization**

CMIS provides no mechanism for an application to explicitly specify a particular locale identifier as part of a CMIS service invocation – It is left up to each ECM repository, the protocols used by applications to interact with the repository via CMIS services, and the applications using those protocols, to determine: (a) how the user’s locale will be determined by the repository, and (b) how the repository will respond to the request based on that locale information.

CMIS Repositories will not directly expose multiple localized representations of a single object.

Where appropriate, the bindings may expose xml:lang attribute if known to aid in localization.
CMIS provides an interface for an application to access a **Repository**. To do so, CMIS specifies a core data model that defines the *persistent* information entities that are managed by the repository, and specifies a set of basic services that an application can use to access and manipulate these entities. In accordance with the CMIS objectives, this data model does not cover *all* the concepts that a full-function ECM repository typically supports. Specifically, transient entities (such as programming interface objects), administrative entities (such as user profiles), and extended concepts (such as compound or virtual document, work flow and business process, event and subscription) are not included.

However, when an application connects to a CMIS service endpoint, the same endpoint may provide access to more than one CMIS repositories. (How an application obtains a CMIS service endpoint is outside the scope of CMIS. How the application connects to the endpoint is a part of the protocol that the application uses.) An application may use the CMIS “Get Repositories” service (getRepositories) to obtain a list of repositories that are available at that endpoint. For each available repository, a **Repository Name**, a **Repository Identity**, and an URI are returned. The Repository Identity uniquely identifies an available repository at this service endpoint. Both the repository name and the repository identity are opaque to CMIS. Aside from the “Get Repositories” service, all other CMIS services are single-repository-scoped, and require a Repository Identity as an input parameter. In other words, except for the “Get Repositories” service, multi-repository and inter-repository operations are not supported by CMIS.

Commercial ECM repositories vary in their designs. Moreover, some repositories are designed for a specific application domain and may not provide certain features that are not needed for their targeted domain. Thus, a repository implementation may not necessarily be able to support all CMIS features. A few CMIS features are therefore “optional” for a repository to be compliant. A repository’s support for each of these optional features is discoverable using the CMIS “Get Repository Information” service (getRepositoryInfo). The following is the list of these optional features:

- Ability to file a document (or other fileable object that is not a folder) in more than one folder (“Multi-filing”)
- Ability to leave a document (or other fileable object that is not a folder) not filed in any folder (“Unfiling”)
- Ability to file a particular version (i.e., not all versions) of a document in a folder (“Version-specific filing”)
- Ability to update the “Private Working Copy” of a checked-out document (“PWC updatable”)
- Ability to include non-latest versions of document in query search scope; otherwise only the latest version of each document is searchable (“All versions searchable”)
- Ability to include the “Private Working Copy” of checked-out documents in query search scope; otherwise PWC’s are not searchable (“PWC searchable”)
- Ability to provide query (enumCapabilityQuery):
  1. none (no query support)
  2. metadataonly (only metadata)
3. fulltextonly (only full text query)
4. both (both fulltext and metadata capabilities provided)
- Join support level in query: This is an enumeration with the following allowable values:
  1. No join support
  2. Support inner join
  3. Support inner join and outer join
- Full-text search support level in query: This is an enumeration with the following allowable values:
  1. No full-text support (i.e., a text-search-predicate is not allowed in the WHERE clause of a query)
  2. Full-text only (i.e., only text-search-predicate is allowed in the WHERE clause of a query)
  3. Full-text + structured/typed search on Properties (i.e., all search predicates are supported in the WHERE clause of a query)

In addition, the “Get Repository Information” service MAY return a list of other repositories that are related to the current repository. This list SHOULD be a subset of the list of repositories that are returned by the “Get Repositories” service. For each of these repositories, a Repository Name, a Repository Identity, an URI, and a Repository Relationship Name are returned. The Relationship Name is opaque to CMIS. Suggested/example values for the “Relationship Name” include:
- “Parent”, “Child”, “Sibling”: For use by systems that support a logical hierarchy of repositories.
- “Replica”, “Archive”: For use by systems that support archival and replication.

Furthermore, the “Get Repository Information” service also returns implementation information such as vendor name, product name, product version, version of CMIS that it supports, the root folder ID, and repository-specific information.

Besides “Get Repository Information”, an application may also get object-type definitions using the “Get Types” and “Get Type Definition” services (getTypes, getTypeDefinition).

All these concepts are described in the sections below.

**OBJECT**

The primary entities managed by CMIS are modeled as typed **Objects**. There are four base types of objects: **Document Objects**, **Folder Objects**, **Relationship Objects**, and **Policy Objects**.

- A *document object* represents a standalone information asset. Document objects are the elementary entities managed by a CMIS repository.
- A *folder object* represents a logical container for a collection of “fileable” objects, which include folder objects and document objects. Folder objects are used to organize fileable objects. Whether or not an object is fileable is specified in its object-type definition.
- A *relationship object* represents an instance of directional relationship between two independent objects (see below).
- A *policy* object represents an administrative policy, which may be “applied” to one or more “controllable” objects. Whether or not an object is controllable is specified in its object-type definition. The support for policy objects is optional, and may be discovered via the “Get Types” service.
Document objects, folder objects, and policy objects are independent objects, in the sense that each object can persist independently as a standalone object in the repository. A relationship object, on the other hand, represents an explicit, application-maintained, instance of relationship between two independent objects. Therefore, relationship objects are semantically dependent objects. Additional object types MAY be created by a repository as subtypes of these base types. CMIS services are provided for the discovery of object types that are defined in a repository. However, object-type management services, such as the creation, modification, and deletion of an object type, are outside the scope of CMIS.

Every CMIS object has an opaque and immutable Object Identity (ID), which is assigned by the repository when the object is created in the repository. An ID uniquely identifies an object within a repository regardless of the type of the object. It is strongly recommended, but not mandated, that IDs be made “permanent”. That is, they remain unchanged during the lifespan of the identified objects, and they are never reused or reassigned after the objects are deleted from the repository. Besides ID, a repository MAY assign a Uniform Resource Identifier (URI), as defined by RFC3986, to an object. URI allows an application to access an object as a web resource using web protocols and tools. However, such accesses beyond the protocol bindings specified by CMIS are outside the scope of CMIS. The permanency of a URI is repository-specific. For example, after a certain period of time or when a certain event occurs, an URI MAY expire (i.e., become invalid) or MAY reference a different resource. Validity of URI is outside the scope of CMIS.

Every CMIS object has a set of named, but not explicitly ordered, Properties. (However, Repositories SHOULD always return object properties in a consistent order.) Within an object, each property is uniquely identified by its name. In addition, a document object MAY have a Content-stream, which may be used to hold a raw digital asset such as an image or a word-processing document. A repository can specify, in each document object type definition, whether or not document objects of that type are allowed to have a content-stream. If allowed, a document object MAY have zero or one content-stream. A content-stream is not named. Instead, it has a repository-assigned URI. Property and content-stream are not shared between objects.

**Property**

A property holds typed data value(s). Each property may be single-valued or multi-valued. A single-valued property contains a single data value, whereas a multi-valued property contains an ordered list of data values of the same type. The ordering SHOULD be preserved by the repository. If a value is not provided for a property, the property is in a “value not set” state. There is no “null” value for a property. Through protocol binding, a property is either not set, or is set to a particular value or a list of values. A multi-valued property is either set or not set in its entirety. An individual value of a multi-valued property cannot be in an individual “value not set” state and hold a position in the list of values. An empty list of values is not allowed.

Property is typed. Property type defines the data type of the data value(s) held by the property. CMIS specifies the following Property Types. They include the following data types defined by “XML Schema Part 2: Datatypes Second Edition” (W3C Recommendation, 28 October 2004, http://www.w3.org/TR/xmlschema-2/):

- String (xsd:string)
- Decimal (xsd:decimal)
- Integer (xsd:integer)
- Boolean (xsd:boolean)
- DateTime (xsd:dateTime)
- URI (xsd:anyURI)
(Note: Precision support for numeric types remains to be addressed. One approach is to keep the abstract data model free of numeric precision, and let each binding to specify the appropriate precision support.)

In addition, the following property types are also specified by CMIS:

- ID (xsd:string)
- XML (xs:any)
- HTML (xs:any)

An ID property holds a system-generated, read-only identity, such as an Object ID, an Object Type ID, a Repository ID, or a Version Series ID. (Specifically, the ID property type is NOT defined by xsd:id.) The lexical representation of an ID is an opaque string. As such, an ID is not interpretable syntactically, can not be collated with other IDs, and can only be used in its entirety as a single atomic value. When used in a query predicate, an ID can only participate in an “equal” or a “not equal” comparison with a string literal or with another ID. While all CMIS identities share the same property type, they do not necessarily share the same address space. (Thus, a repository may use separate address spaces for different kinds of identity, and may perform different validity checks.)

Unless explicitly specified, referential integrity constraint of an ID property is not implied. Therefore, storing the ID of one object in another object does not constrain the behavior of either object. A repository may, however, support referential constraint underneath CMIS if the effect on CMIS services remains consistent with an allowable behavior of the CMIS model. For example, a repository may throw a runtime exception when a CMIS service call violates an underlying referential constraint maintained by the repository. A description may be returned to the application to describe the cause of exception and possibly to suggest a remedial action. The format and content of such description is outside the scope of CMIS.

IDs in CMIS, for documents, usually refer to the specific document version. They can also refer to the version series if obtained from the versionSeriesId property.

An XML property holds a valid fragment of Extensible Markup Language (XML) content.

An HTML property holds a valid fragment of Hypertext Markup Language (HTML) content.

Besides name, type, and whether single- or multi-valued, a property has other defining attributes. They are described in the “Object Type” section.

CMIS provides ID-based CRUD (Create, Retrieve, Update, Delete) services for objects. The “Create” services (createDocument, createFolder, createRelationship, createPolicy) create an object and return an ID. The “Retrieve” service (getProperties) returns the properties of an object, and optionally the operations (CMIS services) that the user is allowed to perform on the object. A filter MAY be supplied to specify which properties are to be returned. For an ID property, the property MAY appear to a user as “value not set” if returning the value would violate an implementation-enforced constraint, such as a security constraint on disclosing the identity of an entity that the user does not have sufficient privilege to know its existence. An opaque, repository-specific, system-generated Change Token MAY be returned by the retrieve service, allowing the repository to perform “optimistic locking” if the retrieved properties are then updated by the user. (A repository may, for example, use an object’s last-modification timestamp as its change token if it has sufficient precision.) The “Update” service (updateProperties) updates the properties of an object. For CMIS Version 1.0, a multi-valued property can only be updated by replacing the entire list of values. Read-only properties are not updatable by application using this service. They are either maintained or computed by the repository. When updating an object, an application MUST
include the change token if it was provided by the repository when the object was retrieved. The repository MUST
then check the change token to determine whether or not the object has been updated since the application
retrieved the object. If it has, then the update SHOULD fail. The “Delete” service (deleteObject) deletes an object.
For a versioned Document object, only the specific version of the document is deleted. (See the “Versioning”
section.)

In addition to these CRUD services, a “Get Allowable Actions” service (getAllowableActions) is provided to allow a
user to discover the CMIS operations he/she can perform on a particular object at that point in time. A repository’s
response not only reflects the behavior allowed by CMIS, but may also reflect constraints imposed by the
underlying implementation or by any management policy that is in effect at the time (such as locking, access
control, legal hold, and lifecycle constraints). The modeling of management policy is outside the scope of CMIS.

When a resource-based protocol is used (such as REST), the Object and ObjectCollection entities are used to
represent respectively an object and a set of objects, and the AllowableAction and AllowableActionCollection
entities are used to represent respectively an allowable action and a set of allowable actions.

Document Object and Content Stream

Document objects are the elementary information entities managed by the repository. A document object MAY
have a content-stream. Furthermore, a document object is versionable (see the “Versioning” section), fileable (see
the “Folder Object” section), and queryable (see the “Query” section). In addition, a document object MAY be
controllable (see the “Policy Object” section). Whether an object is versionable, fileable, queryable, controllable, or
allowed to have a content-stream is specified in its object-type definition. Note: When a document is versioned,
each version of the document is a separate document object. Thus, for document objects, an object ID actually
identifies a specific version of a document. (See the "Versioning" section.)

A content-stream is a binary string. Its maximum length is repository-specific. Each content-stream has a MIME
Media Type, as defined by RFC2045 and RFC2046 and registered with IANA
(http://www.iana.org/assignments/media-types/). A content-stream’s attributes are handled as properties of the
content-stream’s containing document object. There is no MIME-type-specific attribute for a content-stream. A
content-stream is unnamed. But it has a repository-assigned URI which is distinct from the URI (if there is one)
assigned to the document object that contains the content-stream. The permanency of this URI is repository-
specific.

CMIS provides basic CRUD services for content-stream, using the ID of a content-stream’s containing document
object for identification. The “Set Content-Stream” service (setContentStream) either creates a new content-
stream for a document object or replaces an existing content-stream. The “Get Content-Stream” service
(getContentStream) retrieves a content-stream. The “Delete Content-Stream” service (deleteContentStream)
deletes a content-stream from a document object. In addition, the “Check-in” service MAY also take a content-
stream as an optional input (see the “Versioning” section). These are the only services that operate on content-
stream. The “Get Properties” and “Query” services, for example, do not return a content-stream. “Set Content-
Stream” and “Delete Content-Stream” services are considered modifications to a content-stream’s containing
document object, and will therefore change the object’s LastModificationDate property upon successful
completion.

When a resource-based protocol is used, the ContentStream entity is used to represent a content-stream.
Folder Object

Compared to a document object, a folder object does not have a content-stream and is not versionable, although it is queryable and may be controllable. Specifically, a folder object serves as the anchor for an explicit collection of fileable objects, although this collection may be empty. The anchor object itself cannot be a member of its own collection. As the anchor, the folder object has an implicit hierarchical relationship with each object in its collection, with the anchor folder object being the Parent object and each object in the collection being a Child object. This implicit relationship has a specific containment semantics which is maintained by the repository with implicit referential integrity. (That is, conceptually there will never be a dangling parent-relationship or a dangling child-relationship. Furthermore, object A is a parent of object B if and only if object B is a child of object A.) This system-maintained implicit relationship is distinct from an explicit relationship which is instantiated by an application-maintained Relationship Object. (See the “Relationship Object” section.)

A fileable object, therefore, is one that may be “filed” into a folder. That is, it may be a child object of a folder object. Folder objects must be fileable. Document objects should be fileable, and must be fileable if “unfiling” is not supported (see below). Policy objects may be fileable, and relationship objects are not fileable. A child object that is a folder object can itself be the parent object of other fileable objects. Through recursion, the child objects of a given folder object, their child objects, and grandchild objects, etc., are called Descendant objects of the given folder object. Similarly, the parent object of a fileable object, and recursively the parent of the parent, etc., are called Ancestor objects of that fileable object. A folder object together with all its descendant objects are collectively called a Tree rooted at that folder object.

Since document objects are versionable, a document object’s membership in a folder collection may be version-specific or version-independent. That is, the folder membership may be restricted to that particular version of the document or may apply to all versions of the document. Whether or not a repository supports version-specific filing is discoverable via the “Get Repository Information” service (getRepositoryInfo). When the child objects of a folder are retrieved, a specific version of a document is returned if the repository supports version-specific filing, and the latest version is returned if the repository does not support version-specific filing. Likewise, this version sensitivity in child-binding also affects the behavior of parent retrieval for a document object, as well as the scope of the IN_FOLDER() and IN_TREE() function calls in a query. For non-versionable fileable objects, their membership in a folder does not have version sensitivity.

CMIS imposes the following two constraints on folder objects:

- Every folder object, except for one which is called the Root Folder, must have one and only one parent folder. The Root Folder does not have a parent.
- A cycle in folder containment relationships is not allowed. That is, a folder object cannot have itself as one of its descendant objects.

With these two constraints, the folder objects in a CMIS repository necessarily form a strict hierarchy, with the Root Folder being the root of the hierarchy. However, there is no similar constraint imposed on other fileable objects. Therefore, a non-folder fileable object may have any number of parent folders, including none. A fileable object that has multiple parent folders is said to be multi-filed. A fileable object that has no parent folder is said to be unfiled. Multi-filing and unfiling are optional features that are discoverable through the “Get Repository Information” service. On the other hand, by definition, a non-folder object does not have any descendant object. Thus, a Folder Graph that consists of all fileable objects as nodes, and all the implicit folder containment relationships as directed edges from parent to child, is a directed acyclic graph, possibly with some disconnected
(orphan) nodes. It follows that the tree rooted at any given folder object is also a directed acyclic graph, although a non-folder object in the tree MAY have ancestors that are not ancestors of the rooted folder.

A folder collection’s membership MAY be restricted by object type. Each folder object has a multi-valued AllowedChildObjectTypes property, which specifies that only objects of these types are allowed to be its children. (These allowed object types SHOULD all be fileable.) If this property is “not set”, then objects of any fileable type are allowed. If a repository does not support type constraint on folder membership, it may define this folder property as read-only, and keep the property value “not set” for all folder objects.

Because of these filing constraints, when a new folder object is created, an existing folder object MUST be specified as its parent. However, when a non-folder object is created, specifying a parent folder is optional, and is not allowed if the object is not fileable. In both cases, when a parent folder is specified, the newly created object MUST be one of the allowed child types for the given parent folder. On the other hand, when a fileable object is deleted, it is removed from any folder collection in which the object is a member. In other words, when an object is deleted, all implicit parent-child relationships with the deleted object as a child cease to exist. If the to-be-deleted object is a folder with at least one child object, then the deletion MUST fail.

Folder objects are handled using the basic CRUD services for objects. In addition, services are provided for maintaining and traversing the folder graph. The “Add Object to Folder” service (addObjectToFolder) may be used to add a non-folder fileable object to a folder’s collection, regardless of whether or not the object is already a child of another folder. If multi-filing is not supported, an attempt to add a filed object to another folder will fail. The “Remove Object from Folder” service (removeObjectFromFolder) may be used to remove a non-folder object from a folder’s collection without deleting the object and without changing its membership in other folder collections. If unfiling is not supported, an attempt to remove an object from its sole containing folder will fail. A folder object can not be added to or removed from a folder collection due to the single-parent constraint. Instead, the “Move Object” service (moveObject) may be used to change the parent of a child object. The acyclicity constraint of the folder hierarchy implies that a folder can not be moved to one of its descendant folders. The “Delete Tree” service (deleteTree) may be used to delete the tree rooted at a particular folder object. As options, the non-folder objects in this tree may be retained and disconnected from this tree (i.e., they survive as members of folders outside this tree, or as orphans if unfiling is supported); or they may be deleted unless they are members of folders outside this tree (i.e., no orphan is created); or they may be unconditionally deleted (including their membership in any folder outside this tree). For navigation through the folder graph, the “Get Children”, “Get Descendants”, “Get Folder Parent”, and “Get Object Parents” services (getChildren, getDescendants, getFolderParent, getObjectParents) may be used. “Get Children” supports paging, which is useful for scrolling through a large folder. In comparison, “Get Descendants” allows an application to traverse a folder sub-hierarchy to a certain depth. However, the tree-traversal scheme (e.g., depth-first versus breadth-first) is repository-specific and there is no paging support. When a child document object is traversed, depending on the version sensitivity of the child binding (which is repository-specific), either a specific version or the latest version of the child object is returned. “Get Folder Parent” returns the parent of a folder, and optionally all its ancestors. “Get Object Parents” returns the parent(s) of a non-folder fileable object, but not other ancestors.

The Root Folder is a special folder such that it can not be created, deleted, or moved using CMIS services. Otherwise, it behaves like any other folder object.
Folders also have a path which consists of '/' for root folder plus the first sub-folder’s name with '/' as separator between folder names. An example would be, /FolderA/FolderB. Please note the root folder’s name is not specified and it is represented by the first '/'.

Documents in CMIS do not have paths.
A relationship object is semantically a dependent object. Compared to other objects, a relationship object does not have a content-stream, and is not versionable, not queryable, and not fileable, although it MAY be controllable. Specifically, a **Relationship Object** instantiates an explicit, binary, directional, non-invasive, and typed relationship between a **Source Object** and a **Target Object**. The source object and the target object MUST be an independent object, such as a document object, a folder object, or a policy object, but not a relationship object. Whether a policy object is allowed to be the source or target object of a relationship object is repository-specific. The relationship instantiated by a relationship object is explicit since it is explicitly represented by an object and is explicitly managed by application. This relationship is non-invasive in the sense that creating or removing this relationship does not modify either the source or the target object. That is, it does not require an update capability (or permission) on either object; does not affect the versioning state of either object; and does not change their “Last Modification Date”. The source object and the target object of a relationship MAY be the same object. In that case, it is a relationship from an object to itself. For example, a French translation (relationship) of a document written in French can be the document itself. Explicit relationships may be used to create an arbitrary relationship graph among independent objects. Such a relationship graph is only structural in nature. No behavior or inheritance rule is attached to an explicit relationship, nor is there any behavior or constraint propagated/inherited between the source object and the target object through a relationship object.

The notion of a source object and a target object of a relationship is used solely to indicate the direction of the relationship. No semantics or implementation bias is implied by this terminology. The binding of a relationship object to a source document object or to a target document object may be either version-specific or version-independent. This version sensitivity is repository-specific, and is largely transparent to CMIS. An independent object MAY participate in any number of explicit relationships, as the source object for some and as the target object for others. Multiple relationships between the same pair of source and target objects are allowed. Referential integrity, either between the source object and the target object, or between the relationship object and the source or target object, is not specified. Therefore, creating an explicit relationship between two objects does not impose a constraint on any of the three objects, and removing a relationship or deleting either the source or the target object is not restricted by such a constraint. If the source or the target object of a relationship is
deleted, whether or not the relationship object is automatically deleted is repository-specific. A repository may support referential integrity constraint underneath CMIS, either between the source object and the target object of a relationship, or between a relationship object and its source or target object, provided that the resulting effect on CMIS services is consistent with an allowable behavior of CMIS. For example, a repository may throw a generic runtime exception when an underlying referential constraint is violated, and may return a description of the cause of the exception. However such a description is outside the scope of CMIS.

Like all CMIS objects, relationship objects are typed. Typing relationships allows an application to group and label relationships using type name to support different application semantics; allows application-specific properties be defined and attached to relationship instances; and allows an application to selectively traverse relationships from a given independent object using relationship type name. Additionally, a relationship object type MAY specify type constraints on the source object and/or on the target object for relationship objects of that type. If no such constraint is specified, then an independent object of any type may be the source or the target of a relationship object of that type.

Relationship objects are handled using the basic CRUD services for objects. When a relationship object is created, the source object and the target object MUST already exist. When a relationship object is retrieved, its source object or target object may have been deleted since referential integrity may not be maintained by a repository. That is, a relationship object’s SourceID and/or TargetID properties may still reference objects that have been deleted. When a relationship object is deleted, the source and the target objects are left untouched. In addition to object CRUD services, a “Get Relationships” service (getRelationships) may be used to return a set of relationship objects in which a given independent object is identified as the source or the target object, according to the binding semantics maintained by the repository (i.e., either a version-specific or a version-independent binding as described above). This traversal may be selectively restricted to relationships of a given type and/or in a particular direction (i.e., where the given object is the source, the target, or either). From the returned relationship objects, the corresponding “related objects” (i.e., the target of a source, or the source of a target) may then be retrieved. A user should be aware that, in the absence of referential constraint, some “related objects” may have been deleted.

### Policy Object

A policy object represents an administrative policy that can be enforced by a repository, such as an Access Control List (ACL) or a retention management policy. CMIS 1.0 does not specify what kinds of administrative policies that are specifically supported, nor attempts to model administrative policy of any particular kind. Only a base object type is specified for policy objects. Each policy object holds the text of an administrative policy as a repository-specific string, which is opaque to CMIS and which may be used to support policies of various kinds. (For CMIS 1.0, the use case is primarily access control.) A repository may create subtypes of this base type to support different kinds of administrative policies more specifically. The support for policy objects is optional. If a repository does not support policy objects, the policy base object type SHOULD NOT be returned by a “Get Types” service call.

Aside from allowing an application to create and maintain policy objects, CMIS allows an application to “apply” a policy to an object, and to remove an applied policy from an object. An object to which a policy may be applied is called a controllable object. A policy MAY be applied to multiple controllable objects. Conversely, a repository MAY allow multiple policies applied to a controllable object. (A repository may, for example, impose constraints such as only one policy of each kind can be applied to an object.) Whether or not an object is controllable is specified by the object’s type definition. Applying a policy to an object is to place the object under the control of that policy (while the object may also be under the control of other policies at the same time), and removing an applied policy
from one of its controlled objects is to remove the corresponding control from that object. This control may change the state of the object, may impose certain constraints on service calls operating on this object, or may cause certain management actions to take place. The effect of this control, when this effect takes place, and how this control interacts with other controls, are repository-specific. Only directly/explicitly applied policies are covered by CMIS 1.0. Indirectly applying policy to an object, e.g. through inheritance, is outside the scope of CMIS 1.0.

A policy object does not have a content-stream and is not versionable. It may be fileable, queryable, or controllable. Policy objects are handled using the basic CRUD services for objects. If a policy is updated, the change may alter the corresponding control on objects that the policy is currently applied to. If a controlled object is deleted, all the policies applied to that object, if there is any, are removed from that object. A policy object that is currently applied to one or more controllable objects can not be deleted. That is, there is an implicit referential constraint from a controlled object to its controlling policy object(s). Besides the basic CRUD services, the “Apply Policy” (applyPolicy) and the “Remove Policy” (removePolicy) services may be used to apply a policy object to a controllable object and respectively to remove an applied policy from one of its controlled objects. In addition, the “Get Applied Policies” (getAppliedPolicies) service may be used to obtain the policy objects that are currently applied to a controllable object.

**Object Type**

An **Object Type** defines a fixed set of properties that all objects of that type have, which constitutes a “flat” (non-hierarchical) schema for an object’s properties. This schema is used by a repository to validate objects and enforce constraints, and is also used by a user to compose type-based (structured) queries. CMIS objects are strongly typed. Incidental properties that are not prescribed in an object’s type definition (such as “residual properties” in JCR, or “dead properties” in WebDAV) are not modeled. If a property not specified in an object’s type definition is supplied by application, an exception SHOULD be thrown.

Each object type is uniquely identified within a repository by a system-assigned and immutable **Object Type Identity**, which is of type ID. While a repository MAY define additional object types, it MAY NOT extend or alter the behavior or semantics of a CMIS service. A repository may attach constraints to an object type underneath CMIS, provided that the effect visible through the CMIS interface is consistent with the allowable behavior of CMIS.

**Type Hierarchy** and **Type Inheritance** are supported by CMIS in the following manner:

- **A Root Type** does not have a parent type. A non-root type has one and only one parent type. An object type’s **Parent Type** is a part of the object type definition.

- An object type definition includes a set of object type attributes and a set of property definitions. There is no type inheritance for object type attributes.

- **A Subtype** SHOULD inherit all the property definitions that are specified for its parent type. In addition, it MAY have its own property definitions. If a property is NOT inherited by a subtype, for any reason, the exhibited behavior is as if the value of this property is “not set” for all objects of this subtype. The properties of a CMIS base type MUST be inherited by its descendant types.

- The search scope of a query on a given object type is automatically expanded to include all the **Descendant Types** of the given object type. Only the properties of the given object type, including inherited ones, MAY be used in the query. If a property of its parent type is not inherited by this type, the property still appears as a column in the corresponding virtual table in the relational view, but this column
will contain a SQL NULL value for all objects of this type. Properties defined for its descendant types MAY NOT be used in the query, and CAN NOT be returned by the query. (See the “Query” section.)

- **CMIS defines four root types:**
  - Document object type
  - Folder object type
  - Relationship object type
  - Policy object type

  Each MAY be an abstract type. The support for the Policy object type is optional.

- **Additional object types** MAY be defined as subtypes, or descendant types, of these four root types. No additional root types MAY be defined, and no additional services MAY be introduced.

- **Objects of the Document object type or its descendant types are called “document objects” in this specification.**
  - Objects of the Folder object type or its descendant types are called “folder objects” in this specification.
  - Objects of the Relationship object type or its descendant types are called “relationship objects” in this specification.
  - Objects of the Policy object type or its descendant types are called “policy objects” in this specification.

- **CMIS defines a service interface that offers a set of operations. This set of operations is fixed, and MUST NOT be altered or extended.**

A repository that does not support type inheritance natively may define additional object types as immediate subtypes of these three root types. In that case, the type hierarchy is essentially flat. For a query that searches a root type specifically, the repository has to explicitly search all the immediate subtypes of the root type as well.

An **Object Type Definition** contains the following attributes of an object type:

- **ID ObjectType.Id**
  - This attribute uniquely identifies this object type in the repository.
  - This attribute must be unique for all types in the repository

- **String ObjectTypeQueryName**
  - This attribute is used as a table name in a SQL query. It MAY be in mixed case, but MUST uniquely identify this object type within the repository case-insensitively, and MUST conform to the syntax rules for SQL identifiers.
  - This attribute must be unique for all types in the repository

- **String ObjectTypeDisplayName**
  - This attribute is used for presentation by application.

- **ID ParentTypeId**
  - This is the parent type’s ObjectTypeID. It is not set for a root type.

- **String RootTypeQueryName**
  - This is the root type’s ObjectTypeQueryName. Its value can only be Document, Folder, Relationship, or Policy, respectively identifying whether objects of this type are document objects, folder objects, relationship objects, or policy objects.

- **String Description**
  - This is an application’s description of this object type, such as the nature of content, or its intended use.

- **Boolean Creatable**
  - This attribute indicates whether new objects of this type can be created. If an object type is not creatable,
the repository MAY contain objects of this type already, but it is not possible to create new objects of this
type.

- **Boolean Queryable**
  This attribute indicates whether or not this object type is queryable. A non-queryable object type is not visible through the relational view that is used for query, and can not appear in the FROM clause of a query statement. If this object type is non-queryable and its super-type is queried, whether or not objects of this type are included in the search scope is repository-specific. Document and folder object types SHOULD be queryable. Relationship object types are not queryable. Policy object types MAY be queryable.

- **Boolean Controllable**
  This attribute indicates whether or not objects of this type are controllable. Policy objects can only be applied to controllable objects. All object types MAY be controllable.

- **Boolean includeInSuperTypeQuery**
  This attribute controls whether this type and its subtypes appear in a query of this type’s super type. For example, if Invoice is a sub-type of Document, if this is true on Invoice then for a query on Document type, instances of Invoice will be returned if they match. If this attribute is false, no instances of Invoice will be returned even if they match the query.

For Document Types:

- **Boolean Fileable**
  This attribute indicates whether or not objects of this type are fileable. A fileable object is allowed to be a child object of a folder. Folder object types MUST be fileable. Document object types SHOULD be fileable, and MUST be fileable if unfileing is not supported. Relationship object types are not fileable. Policy object types MAY be fileable.

- **Boolean Versionable**
  This attribute indicates whether or not objects of this type are versionable. Only document object types MAY be versionable. It MUST be False for other object types.

- **Enum ContentStreamAllowed**
  This attribute indicates whether content-stream is “not allowed”, “allowed”, or “required” for objects of this type. Only document object types MAY allow or require content-stream. It MUST be “not allowed” for other object types. If content-stream is required, a content-stream MUST be provided when an object is created. The content-stream may be replaced later but can not be deleted.

For Relationship Types:

- **ID AllowedSourceTypes**
  This is a multi-valued attribute for relationship object types, containing a list of object type IDs. It indicates that the source object of a relationship object of this type MUST be of one of these listed object types. If this attribute is “not set”, then the source object can be of any type. This attribute MUST be “not set” if the type being defined is not a relationship object type.

- **ID AllowedTargetTypes**
  This is almost the same as **AllowedSourceTypes** except that the constraint is on the target object of a relationship object instead of on the source object.

All string attributes can be localized by a specific binding client application. CMIS services will return a single value.

All implementations must expose all type attributes on a type even if the function of a particular attribute is not supported. In that case, the implementation must return false, or not-set.

The following table summarizes the allowable type-attribute setting for the four kinds of CMIS objects.
<table>
<thead>
<tr>
<th></th>
<th>Independent obj</th>
<th>Independent obj</th>
<th>Dependent object</th>
<th>Independent obj</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Can have content stream</td>
<td>Can have fileable objects as children</td>
<td>Link 2 independent objects together</td>
<td>Can be “applied” to controllable objects</td>
</tr>
<tr>
<td><strong>Fileable</strong></td>
<td>Should be TRUE</td>
<td>TRUE</td>
<td>FALSE</td>
<td>May be TRUE</td>
</tr>
<tr>
<td><strong>Queryable</strong></td>
<td>Should be TRUE</td>
<td>Should be TRUE</td>
<td>FALSE</td>
<td>May be TRUE</td>
</tr>
<tr>
<td><strong>Controllable</strong></td>
<td>May be TRUE</td>
<td>May be TRUE</td>
<td>May be TRUE</td>
<td>May be TRUE</td>
</tr>
<tr>
<td><strong>Versionable</strong></td>
<td>Should be TRUE</td>
<td>FALSE</td>
<td>FALSE</td>
<td>FALSE</td>
</tr>
<tr>
<td><strong>ContentStreamAllowed</strong></td>
<td>any setting</td>
<td>“not allowed”</td>
<td>“not allowed”</td>
<td>“not allowed”</td>
</tr>
<tr>
<td><strong>AllowedSourceTypes</strong></td>
<td>Not set</td>
<td>Not set</td>
<td>May be set</td>
<td>Not set</td>
</tr>
<tr>
<td><strong>AllowedTargetTypes</strong></td>
<td>Not set</td>
<td>Not set</td>
<td>May be set</td>
<td>Not set</td>
</tr>
</tbody>
</table>

Besides these object type attributes, an object type definition SHOULD contain inherited property definitions and zero or more additional property definitions. All the properties of an object, including inherited properties, are retrievable through the “get” services, and MAY appear in the SELECT clause of a query. However, object properties are logical. It is up to a repository whether a property value is physically stored as a part of an object, or is computed at runtime when the value is requested by an application.

A property definition specifies the following attributes of a property:

- **String PropertyName**
  This attribute identifies this property among all properties of this object type, including inherited properties. This attribute is also used as a column name in a SQL query. It MAY be in mixed case, but MUST uniquely identify this property case-insensitively, and MUST conform to the syntax rules for SQL identifiers.

- **ID PropertyId**
  This attribute contains a system-assigned ID which uniquely identifies this property.

- **String DisplayName**
  This attribute is used for presentation by application.

- **String Description**
  This is an optional attribute containing a description of the property.

- **Boolean IsInherited**
  This attribute indicates whether the property is inherited from the parent object type or it is explicitly defined for this object type.

- **Enum PropertyType**
  This attribute indicates the type of this property. It MUST be one of the allowed property types. (See the “Property” section.)

- **Enum Cardinality**
  This attribute indicates whether the property is “single-valued” or “multi-valued”. Repositories SHOULD preserve the ordering of values in a multi-valued property. That is, the order in
which the values of a multi-valued property are returned in "read" operations SHOULD be the same as the order in which they were supplied during previous "write" operation.

- `<PropertyType array> Choices
  This attribute is optional and is only applicable to application-maintained properties. It specifies what property values are allowed. If choices are not specified, there is no constraint on the data value. Each choice is specified in the form of `<choice name [, value [, index]] <[,choice+]>`. The name is used for presentation purpose. The value will be stored in the property when selected. The index provides guidance for ordering of names when presented.

  Choices can be hierarchically presented. In this case, the index orders siblings in the choice hierarchy; if a choice has no siblings then the index may be omitted; if a choice has siblings then the index is required on all siblings. Please see the bindings for the representation of choice lists.

  If a choice does not specify a value, the name is displayed on the client but cannot be selected. This is useful for hierarchical choice lists where only the leaf entries can be chosen.

- Boolean `OpenChoice`
  This attribute is only applicable to properties that provide a value for the "Choices" attribute. If FALSE, then the data value for the property MUST be one of the values specified in the "Choices" attribute. If TRUE, then values other than those included in the "Choices" attribute may be used for the property.

- Boolean `Required`
  For non-read-only properties:
  The value of a required property MUST always be set. If a value is not provided by application, then the default value is used. If no default value is defined, then this constraint is violated.

  For read-only properties:
  The value of a required system property MUST always be set by a CMIS repository or computed by the repository when it is requested by an application. A not required system property may be left in the "value not set" state.

- `<PropertyType> DefaultValue`
  This attribute is optional and is only applicable to application-maintained properties. The property will have this value if a value is not provided by application. Without a default value, the property value will be left in a "value not set" state until a value is provided by application.

- Enum `Updatability`
  This attribute can be "read-only", "read + write", or "read + write when checked out". A "read-only" property is a system property that is either maintained or computed by the repository. An application can not alter the property value directly using the `updateProperties()` service, and often can not explicitly set the property value when the object is created. In some cases, an application may indirectly cause a change in the property value as a result of calling a special-purpose service. For example, the `ParentID` property of a folder object is maintained by repository. An application CAN NOT alter its value using the `updateProperties()` service, but MAY use the `moveObject()` service to cause a change in the value of the `ParentID` property. A "read + write" property is one that is updatable using the `updateProperties()` service. A "read + write when checked out" property is updatable when the update is made using a "private working copy" object ID (see the "Versioning" section). That is, the update is either made on a "private working copy" object or made using a "check in" service.

- Boolean `Queryable`
  This attribute defines whether or not the property can appear in the WHERE clause of a SQL SELECT statement. Only the properties of a queryable object type, both inherited and specifically defined properties, may be queryable. (Note: "Queryable" has a different meaning for object type and for property. The former pertains to the FROM clause and the latter pertains to the WHERE clause.)
• **Boolean Orderable**
  This attribute defines whether or not the property can appear in the ORDER BY clause of a SQL SELECT statement. Only single-valued properties of a queryable object type may be orderable.

For Decimal Types:
• **Integer precision (enumDecimalPrecision)**
  This is the precision in bits supported for this property (32 or 64 currently)

For Integer Types:
• **Integer MinValue**
  This is the MinValue supported for this property
• **Integer Max Value**
  This is the MaxValue supported for this property

For String Types:
• **Integer MaximumLength**
  This is the maximum length of each data value. It is for variable-length property types only.

For XML Types:
• **URI SchemaURI**
  This attribute is for properties of type “XML” only. It provides the URI location of an XML schema to which the property value MUST conform.
• **String Encoding**
  This attribute is for properties of type “XML” only. It specifies the encoding used for the property value (e.g. UTF-8, etc.).

The four root types mentioned above are CMIS base object types. Additional object types MAY be defined by a repository as descendant types of these four base object types. However, object-type management services are outside the scope of CMIS. This implies that, except for these base object types, a repository has a complete control over the definition of all other object types, such as whether an application is allowed to define new object types, or imposing restrictions on new object types. A CMIS application may use the “Get Types” service to list the object types that are defined in a repository, and use the “Get Type Definition” service to obtain the type definition of a given object type. When a resource-based protocol is used, the **TypeDefin** and **TypeDefinColl** entities are used to represent respectively an object type definition and a set of object type definitions.
Base Object Types

CMIS specifies four root types, representing the four base object types: Document object type, Folder object type, Relationship object type, and Policy object type. They are defined in this section.

The following property attributes have the same specification for every property defined for these four base object types. For simplicity, they are specified here and not repeatedly in each property definition.

- **PropertyId**: Repository-specific
- **DisplayName**: Repository-specific
- **Description**: Repository-specific
- **IsInherited**: FALSE
- **MaximumLength**: Repository-specific for variable-length property type; not applicable otherwise
- **Schema URI**: Not applicable (XML type is not used by these four object types)
- **Encoding**: Not applicable (XML type is not used by these four object types)

The following table defines a set of properties that are common to all four base object types.

<table>
<thead>
<tr>
<th>Property Name</th>
<th>Property Type</th>
<th>Card</th>
<th>Choice</th>
<th>Open Choice</th>
<th>Req’d</th>
<th>Default Value</th>
<th>Upd</th>
<th>Qry</th>
<th>Ord</th>
</tr>
</thead>
<tbody>
<tr>
<td>ObjectId</td>
<td>ID</td>
<td>S</td>
<td>na</td>
<td>na</td>
<td>T</td>
<td>na</td>
<td>RO</td>
<td>RS*</td>
<td>RS*</td>
</tr>
<tr>
<td>Uri</td>
<td>URI</td>
<td>S</td>
<td>na</td>
<td>na</td>
<td>RS</td>
<td>na</td>
<td>RO</td>
<td>RS*</td>
<td>RS*</td>
</tr>
<tr>
<td>ObjectTypeId</td>
<td>ID</td>
<td>S</td>
<td>na</td>
<td>na</td>
<td>T</td>
<td>na</td>
<td>RO</td>
<td>RS*</td>
<td>RS*</td>
</tr>
<tr>
<td>CreatedBy</td>
<td>String</td>
<td>S</td>
<td>na</td>
<td>na</td>
<td>T**</td>
<td>na</td>
<td>RO</td>
<td>T*</td>
<td>T*</td>
</tr>
<tr>
<td>CreationDate</td>
<td>DateTime</td>
<td>S</td>
<td>na</td>
<td>na</td>
<td>T**</td>
<td>na</td>
<td>RO</td>
<td>T*</td>
<td>T*</td>
</tr>
<tr>
<td>LastModifiedBy</td>
<td>String</td>
<td>S</td>
<td>na</td>
<td>na</td>
<td>T**</td>
<td>na</td>
<td>RO</td>
<td>T*</td>
<td>T*</td>
</tr>
<tr>
<td>LastModificationDate</td>
<td>DateTime</td>
<td>S</td>
<td>na</td>
<td>na</td>
<td>T**</td>
<td>na</td>
<td>RO</td>
<td>T*</td>
<td>T*</td>
</tr>
<tr>
<td>ChangeToken</td>
<td>String</td>
<td>S</td>
<td>na</td>
<td>na</td>
<td>RS</td>
<td>na</td>
<td>RO</td>
<td>RS*</td>
<td>RS*</td>
</tr>
</tbody>
</table>

*RS*: Repository-specific

*na*: Not applicable

*: “F” for Relationship object type, “RS” for Policy object type

**: “RS” for Relationship object type

Below are the definitions for the four CMIS base object types.

(1) The definition of the **Document** object type contains the following information:
- **ObjectTypeID**: <Repository-specific>
- **ObjectTypeQueryName**: Document
- **ObjectTypeDisplayName**: <Repository-specific>
- **ParentTypeID**: <Not set>
- **RootTypeQueryName**: Document
- **Description**: <Repository-specific>
- **Creatable**: <Repository-specific>
- **Fileable**: TRUE
- **Queryable**: TRUE
- **Controllable**: <Repository-specific>
- **Versionable**: <Repository-specific>
- **ContentStreamAllowed**: <Repository-specific>
- **AllowedSourceTypes**: <Not set>
- **AllowedTargetTypes**: <Not set>
- **Property definitions**: The above common properties plus the following:

<table>
<thead>
<tr>
<th>Property Name</th>
<th>Property Type</th>
<th>Card</th>
<th>Choice</th>
<th>Open Choice</th>
<th>Req'd</th>
<th>Dfult Value</th>
<th>Upd</th>
<th>Qry</th>
<th>Ord</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td>String</td>
<td>S</td>
<td>RS</td>
<td>RS</td>
<td>RS</td>
<td>RS</td>
<td>RS</td>
<td>RS</td>
<td>RS</td>
</tr>
<tr>
<td>IsImmutable</td>
<td>Boolean</td>
<td>S</td>
<td>na</td>
<td>na</td>
<td>RS</td>
<td>na</td>
<td>RO</td>
<td>RS</td>
<td>RS</td>
</tr>
<tr>
<td>IsLatestVersion</td>
<td>Boolean</td>
<td>S</td>
<td>na</td>
<td>na</td>
<td>T</td>
<td>na</td>
<td>RO</td>
<td>RS</td>
<td>RS</td>
</tr>
<tr>
<td>IsMajorVersion</td>
<td>Boolean</td>
<td>S</td>
<td>na</td>
<td>na</td>
<td>RS</td>
<td>na</td>
<td>RO</td>
<td>RS</td>
<td>RS</td>
</tr>
<tr>
<td>IsLatestMajorVersion</td>
<td>Boolean</td>
<td>S</td>
<td>na</td>
<td>na</td>
<td>RS</td>
<td>na</td>
<td>RO</td>
<td>RS</td>
<td>RS</td>
</tr>
<tr>
<td>VersionLabel</td>
<td>String</td>
<td>RS</td>
<td>na</td>
<td>na</td>
<td>T</td>
<td>na</td>
<td>RO</td>
<td>RS</td>
<td>RS</td>
</tr>
<tr>
<td>VersionSeriesId</td>
<td>ID</td>
<td>S</td>
<td>na</td>
<td>na</td>
<td>T</td>
<td>na</td>
<td>RO</td>
<td>RS</td>
<td>RS</td>
</tr>
<tr>
<td>IsVersionSeriesCheckedOut</td>
<td>Boolean</td>
<td>S</td>
<td>na</td>
<td>na</td>
<td>T</td>
<td>na</td>
<td>RO</td>
<td>RS</td>
<td>RS</td>
</tr>
<tr>
<td>VersionSeriesCheckedOutBy</td>
<td>String</td>
<td>S</td>
<td>na</td>
<td>na</td>
<td>F</td>
<td>na</td>
<td>RO</td>
<td>RS</td>
<td>RS</td>
</tr>
<tr>
<td>VersionSeriesCheckedOutId</td>
<td>ID</td>
<td>S</td>
<td>na</td>
<td>na</td>
<td>F</td>
<td>na</td>
<td>RO</td>
<td>RS</td>
<td>RS</td>
</tr>
<tr>
<td>CheckinComment</td>
<td>String</td>
<td>S</td>
<td>na</td>
<td>na</td>
<td>F</td>
<td>na</td>
<td>RO</td>
<td>RS</td>
<td>RS</td>
</tr>
<tr>
<td>ContentStreamAllowed**</td>
<td>String</td>
<td>S</td>
<td>na</td>
<td>na</td>
<td>T</td>
<td>na</td>
<td>RO</td>
<td>RS</td>
<td>RS</td>
</tr>
<tr>
<td>ContentStreamLength***</td>
<td>Integer</td>
<td>S</td>
<td>na</td>
<td>na</td>
<td>F*</td>
<td>na</td>
<td>RO</td>
<td>RS</td>
<td>RS</td>
</tr>
</tbody>
</table>
RS: Repository-specific

na: Not applicable

*: This property MUST be “not set” if the object does not have a content-stream. It is required if the object has a content-stream.

**: This is an Enum: "not allowed", "allowed", or "required" (like the ContentStreamAllowed attribute of an Object Type).

***: ContentStreamLength MUST be the length of the content stream in bytes.

(2) The definition of the **Folder** object type contains the following information:

- **ObjectTypeID**: <Repository-specific>
- **ObjectTypeQueryName**: Folder
- **ObjectTypeDisplayName**: <Repository-specific>
- **ParentTypeID**: <Not set>
- **RootTypeQueryName**: Folder
- **Description**: <Repository-specific>
- **Creatable**: <Repository-specific>
- **Fileable**: TRUE
- **Queryable**: TRUE
- **Controllable**: <Repository-specific>
- **Versionable**: FALSE
- **ContentStreamAllowed**: "not allowed"
- **AllowedSourceTypes**: <Not set>
- **AllowedTargetTypes**: <Not set>
- **Property definitions**: The above common properties plus the following:

<table>
<thead>
<tr>
<th>Property Name</th>
<th>Property Type</th>
<th>Card</th>
<th>Choice</th>
<th>Open Choice</th>
<th>Req’d</th>
<th>Default Value</th>
<th>Upd</th>
<th>Qry</th>
<th>Ord</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name*</td>
<td>String</td>
<td>S</td>
<td>RS</td>
<td>RS</td>
<td>RS</td>
<td>none</td>
<td>RS</td>
<td>RS</td>
<td>RS</td>
</tr>
<tr>
<td>ParentId*</td>
<td>ID</td>
<td>S</td>
<td>na</td>
<td>na</td>
<td>T</td>
<td>na</td>
<td>RO</td>
<td>RS</td>
<td>RS</td>
</tr>
<tr>
<td>AllowedChildObjectTypes</td>
<td>ID</td>
<td>M</td>
<td>na</td>
<td>na</td>
<td>F**</td>
<td>na</td>
<td>RS</td>
<td>RS</td>
<td>RS</td>
</tr>
</tbody>
</table>

RS: Repository-specific

na: Not applicable

*: For the root folder, Name="CMIS_Root_Folder" and ParentID contains the ID of the root
(3) The definition of the **Relationship** object type contains the following information:

- **ObjectTypeID**: <Repository-specific>
- **ObjectTypeDisplayName**: <Repository-specific>
- **ParentTypeID**: <Not set>
- **ChildSubtypeList**:
  - **ChildSubtypeID**: <Not set>
- **RootTypeQueryName**: Relationship
- **Description**: "not allowed"
- **Creatable**: <Repository-specific>
- **Queryable**: FALSE
- **Creatable**: <Repository-specific>
- **Fileable**: FALSE
- **Queryable**: FALSE
- **Controllable**: <Repository-specific>
- **Versionable**: FALSE
- **ContentStreamAllowed**: "not allowed"
- **AllowedTargetTypes**: <Not set>
- **Property definitions**: The above common properties plus the following:

<table>
<thead>
<tr>
<th>Property Name</th>
<th>Property Type</th>
<th>Card</th>
<th>Choice</th>
<th>Open Choice</th>
<th>Req'd</th>
<th>Default Value</th>
<th>Upd</th>
<th>Qry</th>
<th>Ord</th>
</tr>
</thead>
<tbody>
<tr>
<td>SourceId</td>
<td>ID</td>
<td>S</td>
<td>RS</td>
<td>RS</td>
<td>T</td>
<td>na</td>
<td>RS</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>TargetId</td>
<td>ID</td>
<td>S</td>
<td>RS</td>
<td>RS</td>
<td>T</td>
<td>na</td>
<td>RS</td>
<td>F</td>
<td>F</td>
</tr>
</tbody>
</table>

RS: Repository-specific

na: Not applicable

(4) The definition of the **Policy** object type contains the following information:

- **ObjectTypeID**: <Repository-specific>
- **ObjectTypeQueryName**: Policy
- **ObjectTypeDisplayName**: <Repository-specific>
- **ParentTypeID**: <Not set>
- **RootTypeQueryName**: Policy
- **Description**: <Repository-specific>
- **Creatable**: <Repository-specific>
- **Fileable**: <Repository-specific>
- **Queryable**: <Repository-specific>
- **Controllable**: <Repository-specific>
- **Versionable**: FALSE
- **ContentStreamAllowed**: "not allowed"
- **AllowedSourceTypes**: <Not set>
- **AllowedTargetTypes**: <Not set>
- **Property definitions**: The above common properties plus the following:
The support for policy objects is optional. If a repository does not support policy objects, the “Get Types” service SHOULD NOT return the Policy object type.

### Enumerations

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>enumDecimalPrecision</td>
<td>Integer</td>
<td>32, 64</td>
</tr>
<tr>
<td>enumContentStreamAllowed</td>
<td>String</td>
<td>notallowed, allowed, required</td>
</tr>
<tr>
<td>enumCardinality</td>
<td>String</td>
<td>single, multi</td>
</tr>
<tr>
<td>enumUpdateability</td>
<td>String</td>
<td>readonly, readwrite, whencheckedout</td>
</tr>
<tr>
<td>enumPropertyType</td>
<td>String</td>
<td>Boolean, id, integer, datetime, decimal, html, uri, xml</td>
</tr>
<tr>
<td>enumCollectionType</td>
<td>String</td>
<td>root-children, root-descendants, unfiled, checkedout, types-children, types-descendants, query</td>
</tr>
<tr>
<td>enumObjectType</td>
<td>String</td>
<td>document, folder, relationship, policy</td>
</tr>
<tr>
<td>enumCapabilityQuery</td>
<td>String</td>
<td>none, metadataonly, fulltextonly, both</td>
</tr>
<tr>
<td>enumCapabilityJoin</td>
<td>String</td>
<td>nojoin, inneronly, innerandouter</td>
</tr>
<tr>
<td>Enum Name</td>
<td>Type</td>
<td>Values</td>
</tr>
<tr>
<td>-----------------------------------------------</td>
<td>---------</td>
<td>----------------------------------------------------------------------</td>
</tr>
<tr>
<td>EnumCapabilityFullText</td>
<td>String</td>
<td>none, fulltextonly, fulltextandstructured</td>
</tr>
<tr>
<td>EnumRepositoryRelationship</td>
<td>String</td>
<td>self, replica, peer, parent, child, archive</td>
</tr>
<tr>
<td>EnumTypesOfFileableObjects</td>
<td>String</td>
<td>documents, folders, policies, any</td>
</tr>
<tr>
<td>EnumVersioningState</td>
<td>String</td>
<td>checkedout, minor, major</td>
</tr>
<tr>
<td>EnumReturnVersion</td>
<td>String</td>
<td>this, latest, latestmajor</td>
</tr>
<tr>
<td>EnumUnfileNonfolderObjects</td>
<td>String</td>
<td>unfile, deletesinglefiled, delete</td>
</tr>
<tr>
<td>EnumRelationshipDirection</td>
<td>String</td>
<td>source, target, both</td>
</tr>
<tr>
<td>EnumIncludeRelationships</td>
<td>String</td>
<td>none, source, target, both</td>
</tr>
</tbody>
</table>

### Property Filter

A property filter can be used to specify which properties to be returned in a request. A valid property filter will follow the following grammar:

```
[*] | <propertyname>[,][*]<propertyname>*
```

### Versioning

CMIS supports versioning of document objects. Folder objects, relationship objects, and policy objects are not versionable. Whether or not a document object is versionable is specified in its object type definition. A **Version** of a versionable document object is an explicit copy of the document object, preserving its state at a certain point in time. In general, a versionable document object MAY have multiple versions co-existing in a repository. A document object that is not versionable has only one version. Each version of a document object is itself managed as a separate, standalone document object. As such, it has its own object ID and its own administrative metadata such as **CreationDate** and **LastModificationDate** properties. The set of all versions of a document object has transitive closure. That is, if object B is a version of object A, and object C is a version of object B, then object C is also a version of object A. Such a transitive set is called a **Version Series** (or **Series** for short). Each version series has a unique, system-assigned, and immutable **Version Series ID**. The objects in a version series may be conceptually sequenced by their respective **LastModificationDate** property. The version that has the newest **LastModificationDate** is called the **Latest Version** of the series, or equivalently, the latest version of any document object in the series. Aside from such chronological sequencing, an explicit version history of a document is not maintained. A version may be designated a **Major Version**. The semantics of being a “major version” is repository-related.
specific (or application-specific). Among all the major versions of a document object, if there is any, the one that has the newest LastModificationDate is called the Latest Major Version of the series. (Note: A document object always has a latest version, but may not necessarily have a latest major version.) Whether the non-latest versions of a document object are updatable, queryable or full-text-searchable is repository-specific.

A document object has the following properties pertaining to versioning:

- **IsLatestVersion**
  - Whether or not this object is the latest version

- **IsMajorVersion**
  - Whether or not this object is a major version

- **IsLatestMajorVersion**
  - Whether or not this object is the latest major version

- **VersionSeriesID**
  - Version series ID

- **VersionSeriesIsCheckedOut**
  - Whether or not this object's series is checked out

- **VersionSeriesCheckedOutBy**
  - The user who checked out this object's series

- **VersionSeriesCheckedOutID**
  - Private working copy ID if the series is checked out

- **CheckinComment**
  - A user comment when this version is created

When the latest version of a version series is deleted, a previous version (if there is one) becomes the latest version. Similarly, when the latest major version is deleted, a previous major version (if there is one) becomes the latest major version. Although the IsLatestVersion property, or respectively the IsLatestMajorVersion property, of that previous version is now changed, this is not considered a modification to the previous version.

To create a new version of a versionable document object, a user first “checks out” (checkOut) the object. Whether only the latest version of a document object can be checked out or any version of a document object can be checked out is repository-specific. Only one version of a version series may be checked out at any one point in time. A check-out service:

- “Locks” the entire version series of the checked-out object such that all versions of the checked-out object are now not eligible for check-out while this check-out is in effect,

- Creates a Private Working Copy (PWC) object, which may not be visible to other users. The PWC is not considered the latest version and is not a major version. In creating the PWC, a repository MUST copy the checked-out object's properties to the PWC, and MAY copy the checked-out object's content-stream if there is one. If content-stream is not copied, the PWC’s content-stream properties SHOULD be adjusted, if necessary, to reflect that the PWC does not have a content-stream.

- Returns the ID of this private working copy object.

After a successful completion of a check-out call, all objects in the version series, including the checked-out object, remain read-accessible in their respective states just before the check-out is performed, with the VersionSeriesIsCheckedOut property set to TRUE, the VersionSeriesCheckedOutBy property set to the ID of the user who performed the check-out, and the VersionSeriesCheckedOutID property set to the ID of the private working copy. These changes are not considered a modification of the previous versions. A repository MAY withhold the check-out user ID or the ID of the private working copy if a user does not have sufficient privilege to receive such information. The latest version and the latest major version of the series remain unchanged.

Using the private working copy ID, an (authorized) user MAY explicitly update the private working copy (updateProperties, setContentStream, deleteContentStream). All constraints will be enforced on the private working copy as they will be on any document object. Thereafter, the user MAY “check in” (checkIn) the private working copy. The check-in service:
- Provides an option to update the private working copy (in lieu of, or in addition to, explicit update service calls),
- Creates from the private working copy a new object that is accessible by other users. This new object becomes the latest version of the series. The user MAY designate this object a major version. In that case, this object also becomes the latest major version of the series. In addition, the user MAY submit a comment to be stored in the CheckinComment property of the new object.
- Deletes the private working copy object,
- Unlocks the version series so that all versions in the series become eligible for check-out again, and
- Returns the ID of the new object, which may or may not be the same as the ID of the deleted private working copy object.

A successful completion of a check-in call will change the IsLatestVersion property from TRUE to FALSE for the previous “latest version”. If the new object is a major version, it will also change the IsLatestMajorVersion property from TRUE to FALSE for the previous “latest major version”. In addition, it will change the VersionSeriesIsCheckedOut property from TRUE to FALSE, and cause the VersionSeriesCheckedOutBy and VersionSeriesCheckedOutID properties to become “value not set” for every version in the series. These changes are not considered a modification to the previous versions.

Additionally, some repositories MAY automatically create a new version of a Document without an explicit check-out/check-in operation sequence, when the Document is updated.

Some repositories may not allow explicit update of the private working copy. In that case, such an update will fail and the user SHOULD supply a new version of the object in entirety with the check-in call. The ability to update the private working copy is an optional feature that is discoverable through the “Get Repository Information” service. If a private working copy is not updatable, its ID (returned by the check-out service) MAY be the same as the ID of the checked-out object. In that case, the check-in service MUST return a new ID for the new object created by check-in.

In lieu of checking-in the private working copy, the user MAY use the private working copy ID to “Cancel Checkout” (cancelCheckOut), which removes the effect of the check-out by deleting the private working copy, invalidating its ID, and unlocking the version series. A user may use the “Get Checked-out Documents” service (getCheckedoutDocs) to retrieve the private working copies of checked-out objects that the user is allowed to update.

When the "Create Document" service (createDocument) is used to create a new document object, the new object may be placed in a "checked in – major" state, in a "checked in – minor" state, or in a "checked out" state. If it is in a "checked in – major" or "checked in – minor" state, a first and the latest version of the document is created, which is a major version, or respectively, is not a major version. If the new object is placed in a "checked out" state, then the new object is a PWC. In that case, there is no "checked out document" although a version series is created (i.e., a version series ID is assigned). If a "Check In" service is called on this PWC, a first version of the document will be created. On the other hand, if a "Cancel Checkout" service is called on this PWC, the PWC and the version series will be deleted.

CMIS does not specify an explicit way of linking separate versions of a document together. Instead, CMIS allows an application to access different versions of a document by encapsulating the affinity logic in CMIS services.
• Using a document ID, the “Get Properties” service (getProperties) may be used to retrieve the specific document version identified by that ID, or the latest version of the identified object, or the latest major version of the identified object.

• Using a document ID, the “Delete Object” service (deleteObject) may be used to delete the specific document version identified by that ID.

• Using a version series ID, the “Get All Versions” service (getAllVersions) may be used to retrieve all versions of the identified version series.

• Using a version series ID, the “Delete All Versions” service (deleteAllVersions) may be used to delete all versions of the identified version series.

• Using a version series ID, the “Get Properties of Latest Version” service (getPropertiesOfLatestVersion) may be used to retrieve the latest version or the latest major version of the identified version series.

• The membership of a document object in a folder collection may be version-specific or version-independent. This version-sensitivity in child-binding is repository-specific and can be discovered through the “Get Repository Information” service (getRepositoryInfo). The “Get Children” and “Get Descendants” services (getChildren, getDescendants) return either the specific version or the latest version of a child document object according to the binding semantics. Similarly, with version-specific filing, some versions of a document may be filed and other versions may be not. This affects the behavior of the “Delete Tree” service (deleteTree) which deletes either a specific version or all versions of a child document object depending on the binding semantics.

• A relationship object may have either a version-specific or a version-independent binding to the source object and to the target object. This binding semantics is repository-specific and may also be type-specific. The “Get Relationships” service (getRelationships) returns relationship objects based on the binding semantics maintained by the repository. Aside from that, version-sensitivity of an explicit relationship is transparent to CMIS.

• All other document services operate on the specific version identified by a given object ID.

**QUERY**

CMIS provides a type-based query service for user to find/discover objects associatively. Instead of inventing a new query language for the CMIS data model, CMIS defines a read-only Relational View for the CMIS data model. Through this relational view, a user may submit a simple SQL SELECT statement to find CMIS objects. This query language, called CMIS SQL, is based on a subset of the SQL-92 grammar (ISO/IEC 9075: 1992 – Database Language SQL), with a few extensions to enhance its filtering capability for the CMIS data model, such as existential quantification for multi-valued property, full-text search, and folder membership. Other statements of the SQL language are not adopted by CMIS. The semantics of CMIS SQL is defined by the SQL-92 standard, plus the extensions, in conjunction with the model mapping defined by CMIS’s relational view.

Repositories may support additional query grammars besides CMIS SQL. However, accessing them is outside the scope of CMIS. Each binding, may provide non-normative guidance on how to access the underlying native query grammars.
Relational View

The relational view of a CMIS repository consists of a collection of virtual tables that are defined on top of the CMIS data model. These tables may be used by a user to compose CMIS SQL queries. The repository can then use these same virtual tables and their mapping to the CMIS data model to interpret user’s queries for proper processing. This relational view is logical. It does not suggest how a CMIS repository should organize its data. Furthermore, not all the data exposed through this view are necessarily stored physically in a repository. Some values may be computed by a repository at the time a query is processed.

In this relational view, a virtual table is implicitly defined for every queryable object type that is defined in the repository. Non-queryable object types are not visible through this relational view. Each row in these virtual tables corresponds to an object of the corresponding type or of one of its descendant types.

In each virtual table, a column is implicitly defined for every property that is either defined for, or inherited by, the corresponding object type. A column corresponding to a multi-valued property contains a single list value in each row. A property instance that is in a “value not set” state has a SQL NULL value in this relational view. If a property of the parent type is not inherited by this type, a corresponding column still appears in the virtual table, containing a NULL value in every row. For a row that corresponds to an object of a descendant type, any additional property the object may have due to subtyping is not visible through this table. It is visible, however, through the subtype’s own corresponding virtual table. Content-streams are not exposed through this relational view. Therefore they are not addressable in a SQL statement and hence not returnable by a query. However, text content may be fulltext-searched through a SQL function if this is supported by a repository. An object type’s ObjectTypeQueryName...
attribute is used as the table name for the corresponding virtual table, and a property’s PropertyName attribute is used as the column name for the corresponding table column. For this reason, these attributes MUST conform to the syntax rules for SQL identifiers.

Query Search Scope

B is a subtype of A.
C is a subtype of B.

Relational View

Objects of Type A

Objects of Type B

Objects of Type C

Search scope for query on A
Search scope for query on B
Search scope for query on C

= Inherited property definitions

CMIS SQL

Using the “Query” service (query) to submit a simple “SELECT ... FROM ... WHERE ... ORDER BY ...” SQL SELECT statement, a user can find and retrieve objects of a certain type based on the relational view.

- Only queryable object types may be included in the FROM clause. A query’s search scope is defined by the object types listed in the FROM clause, implicitly expanded by each listed object type’s descendant types but possibly excluding non-queryable sub-types. That is, a type (virtual table) listed in the FROM clause contains not only a row for each object of the corresponding object type but also a row for each object of its queryable descendant types. However, properties defined specifically for the descendant types do not appear in the virtual table.

- Inner join and left outer join are supported as an optional feature that is discoverable through the “Get Repository Information” service. Only explicit join using the JOIN keyword is supported. Implicit join through the WHERE clause is not supported. Only an “equal” predicate of single-valued properties is allowed as the join specification.
Only properties of the object types listed in the FROM clause (including their inherited properties) may be used in the SELECT clause, in the WHERE clause, and/or in the ORDER BY clause. Properties defined for a descendant type of the listed types can not be used in the query statement, unless the descendant type is also listed in the FROM clause. If one listed object type is a descendant type of another listed object type, then they are considered separate tables in the query. In that case, the first table happens to contain only a subset of the rows of the second table, but it may have additional columns.

If "*" is "SELECTed" from a table, all single-valued properties of the corresponding object type MUST be returned, including explicitly defined and inherited properties, and all multi-valued properties SHOULD be returned as well. Consistent with relational algebra, the result of a SQL query is a virtual table. For a multi-valued property, a list value is returned. Content-streams are not addressable in the relational view and therefore can not be returned by a query. However, content-stream properties MAY be queryable and CAN be returned.

The SELECT clause will be interpreted as a property filter and only the specified columns/properties will be returned.

In the WHERE clause, only queryable properties are allowed. (See the “Object Type” section. Note: The adjective “queryable” has different meanings for property and for object type.) For CMIS base object types, properties that are “likely to be computed by a repository” are not queryable. ID columns are in general queryable. But since an ID is opaque, filtering on ID is restricted to an “equal” or “not equal” comparison, which is useful for matching an ID or joining two tables. SQL's simple comparison predicate, IN predicate, and LIKE predicate are supported, for single-valued properties only (so that SQL’s semantics is preserved). Boolean conjunction (AND), disjunction (OR), and negation (NOT) of predicates are also supported.

Language extensions are provided to support existential quantification for multi-valued properties. In addition, a CONTAINS() predicate function may be used in the WHERE clause to include a full-text search, and a companion SCORE() scalar function may be used in the SELECT clause to return the full-text search score. Full-text search is an optional feature that is discoverable through the “Get Repository Information” service. There are three support levels:

- No full-text search support: The CONTAINS() and SCORE() functions are not supported.
- Full-text search only: Only a text-search-predicate (i.e., CONTAINS()) is allowed in the WHERE clause. Other predicates (i.e., property-based structured/typed search) are not supported.
- Full-text and structured search: No restriction on full-text and property-based structured/typed search.

Furthermore, an IN_FOLDER() or an IN_TREE() predicate function may be used to test an object’s membership in a folder or respectively in a folder tree.

Only orderable properties are allowed in the ORDER BY clause. (See the “Object Type” section.) Note: Multi-valued properties are not orderable.

Subquery, UNION, GROUP BY clause, HAVING clause, and set function are not supported.
This BNF grammar is a “subset” of the SQL-92 grammar (ISO/IEC 9075: 1992 – Database Language SQL), except for the production alternatives that are shown in red bold italic face. Specifically, except for these extensions, the following production rules are derived from the SQL-92 grammar. Therefore, all the clauses and statements generated by this grammar without the red tokens are valid SQL-92 clauses and statements. The non-terminals used in this grammar are also borrowed from the SQL-92 grammar without altering their semantics. Accordingly, the non-terminal <column name> is used for single-valued properties only so that the semantics of SQL can be preserved and borrowed. This approach not only facilitates comparison of the two query languages, and simplifies the translation of a CMIS query to a SQL query for a RDBMS-based implementation, but also allows future expansion of CMIS SQL to cover a larger subset of SQL with minimum conflict. The CMIS extensions are introduced primarily to support multi-valued properties and full-text search, and to test folder membership. Multi-valued properties are handled separately from single-valued properties, using separate non-terminals and separate production rules to prevent the extensions from corrupting SQL-92 semantics.
<CMIS 1.0 query statement> ::= <simple table> [ <order by clause> ]
<simple table> ::= SELECT <select list> <from clause> [ <where clause> ]
<select list> ::= "*"
   | [ <select sublist> [ { "", <select sublist> }... ] ]
<select sublist> ::= <value expression> [ [ AS ] <column name> ]
   | [ <qualification> "*" ]
   | <multi-valued-column reference>
<value expression> ::= <column reference> | <string value function> | <numeric value function>
<column reference> ::= { [ <qualifier> "." ] <column name> }
<multi-valued-column reference> ::= { [ <qualifier> "." ] <multi-valued-column name> 
<string value function> ::= { UPPER | LOWER } "(" <column reference> ")"
<numeric value function> ::= SCORE()
<qualification> ::= <table name> | <correlation name>
<from clause> ::= FROM <table reference>
<table reference> ::= <table name> [ [ AS ] <correlation name> ]
   | <joined table>
<joined table> ::= "(" <table reference> [ <join type> ] JOIN <table reference> [ <join specification> ]
<join type> ::= INNER | LEFT | FULL | RIGHT | FULL OUTER | LEFT OUTER | RIGHT OUTER
<join specification> ::= ON "(" <column reference> "= " <column reference> ")"
<where clause> ::= WHERE <search condition>
<search condition> ::= <boolean term> | <search condition> OR <boolean term>
<boolean term> ::= <boolean factor> | <boolean term> AND <boolean factor>
<boolean factor> ::= NOT | <boolean test>
<boolean test> ::= <predicate> | (" <search condition> ")
<predicate> ::= <comparison predicate> | <in predicate> | <like predicate> | <null predicate>
   | <quantified comparison predicate> | <quantified in predicate>
   | <text search predicate> | <folder predicate>
<comparison predicate> ::= <value expression> <comp op> <literal>
<comp op> ::= "=" | "<>" | "<" | ">" | "<=" | ">="
<literal> ::= <signed numeric literal> | <character string literal>
/in predicate> ::= <column reference> [ NOT ] IN "(" <in value list> ")"
/in value list> ::= <literal> | { "", <literal> }...
/like predicate> ::= <column reference> [ NOT ] LIKE <character string literal>
/null predicate> ::= { <column reference> | <multi-valued-column reference> } IS [ NOT ] NULL
/quantified comparison predicate> ::= <literal> [ comp op ] <column reference> <any comparison operator>
/quantified in predicate> ::= ANY <multi-valued-column reference> [ NOT ] IN "(" <in value list> ")"
/text search predicate> ::= CONTAINS "(" [ <qualifier> ] ""," text search expression ")"
/folder predicate> ::= { IN_FOLDER | IN_TREE } "(" [ <qualifier> ] "," <folder id> ")"
/order by clause> ::= ORDER BY <sort specification> [ { "", <sort specification> }... ]
<sort specification> ::= <column name> [ ASC | DESC ]
<correlation name> ::= <identifier>!! This MUST be the name of an object type.
<table name> ::= <identifier>!! This MUST be the name of a single-valued property, or an alias for a scalar output value.
<column name> ::= <identifier>!! This MUST be the name of a single-valued property, or an alias for a scalar output value.
<multi-valued-column name> ::= <identifier>  !! This MUST be the name of a multi-valued property.
<folder id> ::= <character string literal>  !! This MUST be the object identity of a folder object.
<text search expression> ::= <character string literal>  !! This is full-text search criteria. The syntax is implementation-specific.
<identifier> ::= !! As defined by SQL-92 grammar.
<signed numeric literal> ::= !! As defined by SQL-92 grammar.
<character string literal> ::= !! As defined by SQL-92 grammar.

CMIS EXTENSIONS

New non-terminals are added to encapsulate new semantics to avoid altering the semantics of SQL-92 production rules. New terminals are added as functions, so that their parameters are encapsulated, to minimize the impact on the SQL language structure.

<Multi-valued-column reference>, <multi-valued-column name>

- These are non-terminals defined for multi-valued properties whereas SQL-92’s <column reference> and <column name> are retained for single-valued properties only. This is to preserve the single-value semantics of a regular “column” in the SQL-92 grammar.

<Quantified comparison predicate>

- The SQL-92 production rule for <quantified comparison predicate> is extended to accept a multi-valued property in place of a <table subquery>. <Table subquery>, on the other hand, is not supported.
- The SQL-92 <quantifier> is restricted to ANY only.
- The SQL-92 <row value constructor> is restricted to a literal only.

<Quantified in predicate>

- This is a new IN predicate defined for a multi-valued property. Syntactically it is patterned after the regular IN predicate for ease of use. Since the entire predicate is different semantically, it has its own production rule (so it can be handled by a different semantic routine).
- The quantifier is restricted to ANY. The predicate is evaluated to TRUE if at least one of the property’s values is (or, is not, if NOT is specified) among the given list of literal values. Otherwise the predicate is evaluated to FALSE.

“*” in the SELECT clause

- The query MUST return all single-valued properties, and SHOULD return all multi-valued properties as well.

CONTAINS ([ [ <qualifier> ] ], <text search expression> )

- This is a predicate function that encapsulates a full-text search and returns a Boolean value, TRUE or FALSE, for each candidate object. A TRUE value means the object is considered by the repository as being “relevant” with respect to the given <text search expression>. A FALSE value means the object is not considered “relevant” by the repository. The criteria for being “relevant” are repository-specific.
- The optional <qualifier> parameter may be used to specify the search scope for the CONTAINS() function. For example, it may specify one of the object types named in the FROM clause. Search scope support for the CONTAINS() function is repository-specific. It may, for example, depend on what text data are full-text-indexed. Therefore, the support for <qualifier> is repository-specific. A repository may full-text-index
content-streams only, or it may full-text-index both content-streams and properties. A repository that
does not support user-specifiable fulltext search scope MAY ignore the <qualifier> parameter.

- The <text search expression> parameter is a character string literal typically given in quotes, specifying the
  full-text search criteria. The syntax and semantics are repository-specific.
- At most one CONTAINS() call is allowed in a query statement, and it can only be included conjunctively
  (ANDed) with the aggregate of all other predicates, if there is any, in the WHERE clause. If a query has
  more than one CONTAINS() call, or if the CONTAINS() call is associated with a Boolean disjunction (OR) or
  negation (NOT), an “Invalid Argument” exception is thrown. The CONTAINS() call is always associated with
  a “top level” query since subquery is not supported.
- Support for full-text search is optional. A repository that does not support full-text search MAY throw an
  “Operation Not Supported” exception if a query statement included a CONTAINS() function call.

SCORE()
- This is a scalar function that returns a full-text search score for each candidate object with respect to a
  companion CONTAINS( [ <qualifier> ], … ) function call in the same query statement. This score is a
  measure of “degree of relevance”, which is a real number in the interval [0,1]. A value of 0 means “no
  relevance” and a value of 1 means “maximum relevance”. The meaning of “relevance” is outside the
  scope of CMIS. Relevance measurement is repository-specific.
- This function can only be used in the SELECT clause of a query. It can not be used in the WHERE clause or
  in the ORDER BY clause. However, an alias column name defined for the SCORE() function call in the
  SELECT clause (i.e., "SELECT SCORE() AS column_name ...") may be used in the ORDER BY clause. An
  “Invalid Argument” exception is thrown if a SCORE() function call appears anywhere in a query other than
  in the SELECT clause, or if a SCORE() function call is included without a CONTAINS() function call. A query
  MAY include a CONTAINS() function call without a SCORE() function call. If SCORE() is included in the
  SELECT clause and an alias column name is not provided, then a default column name of SEARCH_SCORE
  is used for the query output, which will be mapped to a (pseudo) property name through protocol
  binding.

IN_FOLDER([ <qualifier> ], <folder id>)
- This is a predicate function that tests whether or not a candidate object is a child of the folder object
  identified by the given <folder id>, where <folder id> MUST be the ID of a folder object. A Boolean value
  TRUE is returned if the candidate object is a child of the folder. Otherwise a Boolean value FALSE is
  returned. (Note: A multi-filed document object may be tested TRUE for more than one <folder id> value.)
- The optional <qualifier> parameter, if given, identifies one of the tables listed in the FROM clause. In that
  case, this predicate only applies to objects of that specific type. If this parameter is not specified, then the
  test applies to all object types listed in the FROM clause. Support for table-specific IN_FOLDER() is
  optional. A repository that does not support table-specific IN_FOLDER() MAY ignore this <qualifier>
  parameter.
- An “Invalid Argument” exception is thrown if the <qualifier> parameter does not identify one of the tables
  listed in the FROM clause, or if the <folder id> parameter is not the ID of a folder object.

IN_TREE([ <qualifier> ], <folder id>)
- This is a predicate function that is in every aspect the same as the IN_FOLDER() function except that the
  test is on whether or not a candidate object is a descendant object of the given folder object, rather than
  an (immediate) child of the given folder.

QUERY EXAMPLES

- Example of IN_TREE(), quantified comparison:
  (Note: AUTHORS is a multi-valued property.)
  SELECT TITLE, AUTHORS, DATE
FROM WHITE_PAPER
WHERE ( IN_TREE( , 'ID00093854763') ) AND ( 'SMITH' = ANY AUTHORS )

- Example of CONTAINS(), SCORE(), quantified comparison:
  (Note: DEPARTUREDATES is a multi-valued property.)
  SELECT OBJECT_ID, SCORE() AS X, DESTINATION, DEPARTUREDATES
  FROM TRAVEL_BROCHURE
  WHERE ( CONTAINS( , 'CARIBBEAN CENTRAL AMERICA CRUISE TOUR') ) AND
  ( '2010-1-1' < ANY DEPARTUREDATES )
  ORDER BY X DESC

- Example of LOWER(), quantified IN:
  (Note: FEATURES is a multi-valued property.)
  SELECT *
  FROM CAR_REVIEW
  WHERE ( LOWER(MAKE) = 'buick' ) OR
  ( ANY FEATURES IN ('NAVIGATION SYSTEM', 'SATELLITE RADIO', 'MP3') )

- Example of join, quantified comparison, LIKE:
  (Note: DAMAGE_ESTIMATES is a multi-valued property.)
  SELECT Y.CLAIMNUM, X.PROPERTY_ADDRESS, Y.DAMAGE_ESTIMATES
  FROM POLICY AS X JOIN CLAIMS AS Y ON ( X.POLICYNUM = Y.POLICYNUM )
  WHERE ( 100000 <= ANY Y.DAMAGE_ESTIMATES ) AND ( Y.CAUSE NOT LIKE '%Katrina%' )
This section of the CMIS specification defines an intentionally generic set of services that can be performed on CMIS entities.

The services are described in a protocol/binding-agnostic fashion, and every protocol/binding of the CMIS interface (covered in Part II of the CMIS specification) will implement all of the methods described here. Details of how each service & method is implemented in each protocol binding will be covered in Part II of the CMIS specification.

COMMON SERVICE ELEMENTS

The following constructs are common across all of the CMIS services.

Common exceptions

All CMIS services will throw the following common exceptions where appropriate:

- **InvalidArgumentException** if an argument is not valid (wrong type, not supplied etc)
  - **ConstraintViolationException** if an action fails some constraint
  - **ObjectNotFoundException** if Id not found
- **PermissionDeniedException** if not sufficient permission for operation
- **OperationNotSupportedException** if operation not in allowable actions at the point of time of operation invocation (By you, at this time, on this object, ...)
- **RuntimeException** if other

Other exceptions

The following exceptions will be thrown by some CMIS service methods. The individual service definitions below will list the exact exceptions thrown by each method.

- **ContentAlreadyExistsException**: If the caller attempts to set the content stream for a Document that already has a content stream, without specifying that they intend to overwrite the existing content stream.
- **FilterNotValidException** if the caller provided an invalid property filter to a service that takes a property filter as an input (e.g. getChildren).
- **StorageException** if the repository is not able to store the object that the user is creating/updating (for example, because a storage quota has been exceeded.)
- **StreamNotSupportedException** if the caller is attempting to set a contentStream for a Document whose Object Type specifies that a content stream is not allowed for Document's of that type.

- **UpdateConflictException** if performing an operation that updates an object out of sequence:
  a. For example, if you have two clients (Alice and Bob), trying to get and update a document in the following sequence, a **UpdateConflict Exception** would occur:
     - Alice.GetDocument()
     - Bob.GetDocument()
     - Alice.UpdateDocument()
     - Bob.UpdateDocument() \(\rightarrow\) UpdateConflictException

- **VersioningException**: If performing an update operation on a Document that cannot be modified because of its position in a version series (e.g. it's not the current version).

---

**Property Filters**

Any method that allows for the retrieval of object properties can take a Property Filter, to specify what subset of an object's properties are to be returned.

A property filter is a string that contains either '*' (to return all properties) or a comma-separated list of property names (to return selected properties). An arbitrary number of spaces are allowed before or after each comma.

A repository MAY have a core set of properties that it always returns. These properties will always be returned even if they are not specified by the filter. That is, an application requesting a filtered set of properties for an object will always get AT LEAST the properties specified in the filter, and possibly additional properties that the repository chooses to return always.

---

**REPOSITORY SERVICES**

- **getRepositories**

<table>
<thead>
<tr>
<th>Description</th>
<th>Returns a list of available repositories for this CMIS service endpoint.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Inputs</td>
<td>None</td>
</tr>
</tbody>
</table>
| Outputs     | - `<Array>` repositories
  | o ID repositoryId: Repository Identity
  | o String repositoryName: Repository name
  | o URI repositoryURI: Repository URI                                    |
| Exceptions  | - Common Exceptions                                                     |
| Notes       | - This is the only CMIS service that is not scoped to a single repository. |
### getRepositoryInfo

<table>
<thead>
<tr>
<th><strong>Description</strong></th>
<th>This service is used to retrieve information about the CMIS repository and the capabilities it supports.</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Inputs</strong></td>
<td>• ID repositoryId: Repository Id</td>
</tr>
</tbody>
</table>
| **Outputs**     | • ID repositoryId: Repository Id (same as input)  
                  • String repositoryName: Repository name  
                  • URI repositoryURI: URI for this repository  
                  • String repositoryDescription: Description of this repository  
                  • ID rootFolderId: Root folder Id  
                  • String vendorName: Repository vendor name  
                  • String productName: Repository product name  
                  • String productVersion: Product Version Information  
                  • String cmisVersionsSupported: Version of CMIS standard supported.  
                  • XML repositorySpecificInformation: Repository-specific information  
                  • Capabilities  
                      • Boolean capabilityMultifiling  
                      • Boolean capabilityUnfiling  
                      • Boolean capabilityVersionSpecificFiling  
                      • Boolean capabilityPWCUpdatable  
                      • Boolean capabilityAllVersionsSearchable  
                      • Boolean capabilityPWCSearchable  
                      • Enum capabilityJoin: NoJoin, InnerOnly, InnerAndOuter  
                      • Enum capabilityFulltext: NoFulltext, FulltextOnly, FulltextAndStructured  
                  • <Array> relatedRepositories  
                      • ID repositoryId  
                      • String repositoryName  
                      • URI repositoryURI  
                      • String repositoryRelationshipName  |
| **Exceptions**  | • Common Exceptions |
| **Notes**       | |
| **Questions**   | |
**getTypes**

<table>
<thead>
<tr>
<th>Description</th>
<th>Returns the list of all types in the repository.</th>
</tr>
</thead>
</table>
| **Inputs**    | • ID repositoryId: Repository Id  
• (Optional) ID typeId: not set (default)  
• (Optional) Bool returnPropertyDefinitions: False (default)  
• (Optional) int maxItems: 0 = Repository-default number of items (Default)  
• (Optional) int skipCount: 0 = start (Default) |
| **Outputs**   | • Result Set - List of types  
• Boolean hasMoreItems  
• The type attributes of each type will be returned. See the “Object Type” section for a complete list of type attributes.  
• The property definitions of each type will be returned if returnPropertyDefinitions is TRUE. |
| **Exceptions**| • Common Exceptions |
| **Notes**     | • A repository may support a hierarchy of types but CMIS will return them as a flat list.  
• If provided, the input parameter “TypeId” specifies to only return the specific Object Type and its descendants. If not provided, all Object Types are to be returned.  
• If no “maxItems” value is provided, then the Repository will determine an appropriate number of items to return. How the Repository determines this value is repository-specific and opaque to CMIS.  
• If “returnPropertyDefinitions” is False, then the Repository will return only the “Attributes” of the Object Type Definition as specified in the “Object Type” section of the Data Model. Otherwise, property definitions will also be returned for each object type. |
| **Questions** | |

**getTypeDefinition**

<table>
<thead>
<tr>
<th>Description</th>
<th>Gets the definition for specified object type</th>
</tr>
</thead>
</table>
| **Inputs**    | • ID repositoryId: Repository Id  
• ID typeId: Type Id |
| **Outputs**   | • Collection typeAttributeCollection: Type attributes of an object type definition. See the “Object Type” section for a complete list of type attributes.  
• <Array> propertyDefinition: A list of property definitions. Each property definition consists of a collection of property attributes. See the “Object Type” section for a complete list of property attributes. |
NAVI GATION SERVICES

The Navigation services are used to traverse the folder hierarchy in a CMS Repository, and to locate Documents that are checked out.

getDescription

Description
Gets the list of descendant objects contained at one or more levels in the tree rooted at the specified folder. Only the filter-selected properties associated with each object are returned. The content-stream is not returned.

For paging through the children (depth of 1) only use getChildren. For returning a tree of objects of a certain depth, use getDescendants.

For a repository that supports version-specific filing, this will return the version of the documents in the folder specified by the user filing the documents into the folder. Otherwise, the latest version of the documents will be returned.

Inputs
- ID repositoryId: Repository Id
- ID folderId
- (Optional) Enum type: Documents, Folders, Policies, Any (default)
- (Optional) Int depth: 1 this folder only (Default), ... N folders deep, -1 for all levels
- (Optional) String filter: Filter specifying which properties to return.
- (Optional) Boolean includeAllowableActions: False (default)
- (Optional) Enum includeRelationships: none (default), source, target, both
- (Optional) String orderBy: must be a valid ORDER BY clause from the query grammar excluding ‘ORDER BY’. Example ‘name DESC’.

Outputs
- Result set specified by Filter of each descendant object in the specified folder
- This result set will nest the contained objects

Exceptions
- Common Exceptions
- FilterNotValidException: if Filter not valid
### Notes
- The ordering and tree walk algorithm is repository-specific, but SHOULD be consistent.
- This method will return all objects of the specified type in the specified depth.
- If no type is specified, then objects of all types will be returned.
- When returning the results of a call where the caller specified “Any” type, the repository SHOULD return, at each nesting level, all folder objects first followed by other objects.
- If “includeAllowableActions” is TRUE, the repository will return the allowable actions for the current user for each descendant object as part of the output.
- "IncludeRelationships" indicates whether relationships are also returned for each returned object. If it is set to "source" or "target", relationships for which the returned object is a source, or respectively a target, will also be returned. If it is set to "both", relationships for which the returned object is either a source or a target will be returned. If it is set to "none", relationships are not returned.

### Questions

### getChildren

#### Description
Gets the list of child objects contained in the specified folder. Only the filter-selected properties associated with each object are returned. The content-streams of documents are not returned.

For paging through the children (depth of 1) only use getChildren. For returning a tree of objects of a certain depth, use getDescendants.

For a repository that supports version-specific filing, this will return the version of the documents in the folder specified by the user filing the documents into the folder. Otherwise, the latest version of the documents will be returned.

#### Inputs

- ID repositoryId: Repository Id
- ID folderId
- (Optional) Enum type: Documents, Folders, Policies, Any (default)
- (Optional) String filter: Filter specifying which properties to return.
- (Optional) Boolean includeAllowableActions: False (default)
- (Optional) Enum includeRelationships: none (default), source, target, both
- (Optional) int maxItems: 0 = Repository-default number of items (Default)
- (Optional) int skipCount: 0 = start (Default)
- (Optional) String orderBy: must be a valid ORDER BY clause from the query grammar excluding ‘ORDER BY’. Example ‘name DESC’.

#### Outputs

- Result set specified by Filter of each child object in the specified folder
- If maxItems > 0, Bool hasMoreItems

#### Exceptions

- Common Exceptions
- **FilterNotValidException**: if Filter not valid
### Notes
- Between invocations the order of the results may change due to repository state changing, i.e. skipCount might not show objects or more likely show an object twice (bottom of first page and top of second) when an object is added to the top of the list.
- Ordering is repository-specific except the ordering MUST remain consistent across invocations, provided that the repository state has not changed.
- When returning the results of a call where the caller specified “Any” type, the repository SHOULD return all folder objects first followed by other objects.
- If “includeAllowableActions” is TRUE, the repository will return the allowable actions for the current user for each child object as part of the output.
- "IncludeRelationships" indicates whether relationships are also returned for each returned object. If it is set to "source" or "target", relationships for which the returned object is a source, or respectively a target, will also be returned. If it is set to "both", relationships for which the returned object is either a source or a target will be returned. If it is set to "none", relationships are not returned.
- If no “maxItems” value is provided, then the Repository will determine an appropriate number of items to return. How the Repository determines this value is repository-specific and opaque to CMIS.

### Questions

#### getFolderParent

<table>
<thead>
<tr>
<th>Description</th>
<th>Returns the parent folder object, and optionally all ancestor folder objects, above a specified folder object.</th>
</tr>
</thead>
</table>
| Inputs                           | - ID repositoryId: RepositoryId  
- ID folderId: Source folder to get the parent or ancestors of  
- (Optional) String filter: Property filter specifying which properties to return.  
- (Optional) Boolean includeAllowableActions: False (default)  
- (Optional) Enum includeRelationships: none (default), source, target, both  
- (Optional) Bool returnToRoot: If false, return only the immediate parent of the folder. If true, return an ordered list of all ancestor folders from the specified folder to the root folder. Default=False |
| Outputs                          | - ResultSet resultSet of folders  
  - Parent Folder ID + Specified properties  
- Can be a list of folders |
| Exceptions                       | - Common Exceptions  
- FilterNotValidException: if Filter not valid |
| Notes                            | - Output is ordered by ancestry, closest to specified folder first. However XML clients may not always respect ordering. For that reason repositories SHOULD always include the parent and the ObjectID property in the filter to allow re-ordering if necessary.  
- No parent folder is returned if the specified folder is the root folder.  
- If “includeAllowableActions” is TRUE, the repository will return the allowable actions for the current user for each parent or ancestor folder as part of the output. |
**IncludeRelationships** indicates whether relationships are also returned for each returned object. If it is set to "source" or "target", relationships for which the returned object is a source, or respectively a target, will also be returned. If it is set to "both", relationships for which the returned object is either a source or a target will be returned. If it is set to "none", relationships are not returned.

### Questions

**getObjectParents**

<table>
<thead>
<tr>
<th>Description</th>
<th>Returns the parent folders for the specified non-folder, fileable object</th>
</tr>
</thead>
</table>
| Inputs      | • ID repositoryId: Repository Id  
• ID objectId: ID of a non-folder, fileable object.  
• (Optional) String filter: filter specifying which properties to return.  
• (Optional) Boolean includeAllowableActions: False (default)  
• (Optional) Enum includeRelationships: none (default), source, target, both |
| Outputs     | • ResultSet resultSet - Set of folders containing the object. |
| Exceptions  | • Common Exceptions  
• FilterNotValidException |
| Notes       | • Order is repository-specific  
• It is suggested that the parent and the ObjectId properties are included in the filter to allow re-ordering if necessary.  
• If “includeAllowableActions” is TRUE, the repository will return the allowable actions for the current user for each parent folder as part of the output.  
• ”IncludeRelationships” indicates whether relationships are also returned for each returned object. If it is set to "source" or "target", relationships for which the returned object is a source, or respectively a target, will also be returned. If it is set to "both", relationships for which the returned object is either a source or a target will be returned. If it is set to "none", relationships are not returned. |
| Questions   | |

### getCheckedoutDocuments

| Description | Gets the list of documents that are checked out that the user has access to. Most likely this will be the set of documents checked out by the user. Content-streams are not returned. |
### Inputs
- ID repositoryId: Repository Id
  - (Optional) ID folderId
  - (Optional) String filter specifying which properties to return.
  - (Optional) Boolean includeAllowableActions: False (default)
  - (Optional) Enum includeRelationships: none (default), source, target, both
  - (Optional) int maxItems: 0 = Repository-default number of items (Default)
  - (Optional) int skipCount: 0 (Default)

### Outputs
- Result set specified by Filter
- Bool hasMoreItems

### Exceptions
- Common Exceptions
- FilterNotValidException

### Notes
- The documents will be returned in a repository-specific order.
- The repository may include checked-out objects that the calling user has access to, but did not check out.
- If folderId is specified, then the results MUST include only the children of that folder, NOT other descendants of the folder nor documents outside this tree.
- If "includeAllowableActions" is TRUE, the repository will return the allowable actions for the current user for each document as part of the output.
- "IncludeRelationships" indicates whether relationships are also returned for each returned object. If it is set to "source" or "target", relationships for which the returned object is a source, or respectively a target, will also be returned. If it is set to "both", relationships for which the returned object is either a source or a target will be returned. If it is set to "none", relationships are not returned.
- If no “maxItems” value is provided, then the Repository will determine an appropriate number of items to return. How the Repository determines this value is repository-specific and opaque to CMIS.

### Questions

---

**OBJECT SERVICES**

The Object Services are used to perform operations such as Create, Retrieve, Update, Delete (CRUD) on an object.

**createDocument**

<table>
<thead>
<tr>
<th>Description</th>
<th>Creates a document object of the specified type, and optionally adds the document to a folder</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Inputs</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>• ID repositoryId: Repository Id</td>
</tr>
<tr>
<td></td>
<td>• ID typeId: Document type</td>
</tr>
<tr>
<td></td>
<td>• Collection properties</td>
</tr>
<tr>
<td></td>
<td>• (Optional) ID folderId: Parent folder for this new document</td>
</tr>
<tr>
<td></td>
<td>• (Optional) ContentStream contentStream</td>
</tr>
</tbody>
</table>
### createFolder

<table>
<thead>
<tr>
<th>Description</th>
<th>Creates a folder object of the specified type</th>
</tr>
</thead>
</table>

#### Inputs

| ID repositoryId: Repository Id |
| ID typeId: Folder type |
| Collection properties |
| ID folderId: Parent folder for this new folder |

#### Outputs

| ID objectId: Id of the created folder object |

#### Exceptions

| Common Exceptions |
| StorageException |

#### Notes

| If the to-be-created Folder’s Object Type is not one of the “Allowed_Child_Object_Types” for the parent Folder, throw ConstraintViolationException. |
### createRelationship

<table>
<thead>
<tr>
<th>Description</th>
<th>Creates a relationship object of the specified type</th>
</tr>
</thead>
</table>
| Inputs      | • ID repositoryId: Repository Id  
              • ID typeId: Relationship type  
              • Collection properties  
              • ID sourceObjectId  
              • ID targetObjectId |
| Outputs     | • ID objectId: Id of the created relationship object |
| Exceptions  | • Common Exceptions  
              • StorageException |
| Notes       | • If either the source object or the target object is not found, throw the ObjectNotFoundException.  
              • Whether the binding to the source and target objects is version-specific is repository specific and not exposed via CMIS  
              • If the Type ID of the Source or Target object is NOT one of the Object Types allowed on the Relationship Type, throw ConstraintViolationException. |

### createPolicy

<table>
<thead>
<tr>
<th>Description</th>
<th>Creates a policy object of the specified type, and optionally adds the policy to a folder</th>
</tr>
</thead>
</table>
| Inputs      | • ID repositoryId: Repository Id  
              • ID typeId: Policy type  
              • Collection properties  
              • (Optional) ID folderId: Parent folder for this new policy object |
| Outputs     | • ID objectId |
| Exceptions  | • Common Exceptions  
              • StorageException |
<table>
<thead>
<tr>
<th>Notes</th>
</tr>
</thead>
</table>
| • If this policy object type is not fileable and a parent Folder is specified, throw FolderNotValidException.  
• If this policy object type is fileable and a parent Folder is not specified, but un filing is not supported, throw FolderNotValidException.  
• If a parent Folder is specified, and the Policy's Object Type is not one of the “Allowed Child Object Types” for the parent Folder, throw ConstraintViolationException. |

<table>
<thead>
<tr>
<th>Questions</th>
</tr>
</thead>
</table>

---

### getAllowableActions

<table>
<thead>
<tr>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>This service returns the list of allowable actions (CMIS service calls) for an object based on the current user’s context, subject to any access constraints that are currently imposed by the repository.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Inputs</th>
</tr>
</thead>
</table>
| • ID repositoryId: Repository Id  
• ID objectId: Object Id  
• (Optional) String asUser: Not Set (Default) |

<table>
<thead>
<tr>
<th>Outputs</th>
</tr>
</thead>
</table>
| • Collection allowableActionCollection.  
• The actions that are maximally allowed when there is no access constraint:  
  o For a Document object:  
    getProperties, updateProperties, deleteObject, getObjectParents, moveObject, addObjectToFolder, removeObjectFromFolder, createRelationship, getRelationships, applyPolicy, removePolicy, getAppliedPolicies, getContentType, setContentType, deleteContentStream, checkOut, cancelCheckout, checkIn  
  o For a Folder object:  
    getProperties, updateProperties, deleteObject, getFolderParent, moveObject, addObjectToFolder, removeObjectFromFolder, createRelationship, getRelationships, applyPolicy, removePolicy, getAppliedPolicies, getChildren, getDescendants, createDocument, createFolder, createPolicy, deleteTree  
  o For a Relationship object:  
    getProperties, updateProperties, deleteObject, applyPolicy, removePolicy, getAppliedPolicies  
  o For a Policy object:  
    getProperties, updateProperties, deleteObject, getObjectParents, moveObject, addObjectToFolder, removeObjectFromFolder, createRelationship, getRelationships, applyPolicy, removePolicy, getAppliedPolicies |

<table>
<thead>
<tr>
<th>Exceptions</th>
</tr>
</thead>
<tbody>
<tr>
<td>• Common Exceptions</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Notes</th>
</tr>
</thead>
</table>
| • Not Set is the current user context  
• Repository-specific allowable actions can also be returned. However, as they are repository-specific their semantics and invocation is not defined through CMIS. |
### getProperties

| Description | Returns the properties of an object, and optionally the operations that the user is allowed to perform on the object |
| Inputs | - ID repositoryId: Repository Id  
- ID objectId  
- (Optional) Enum returnVersion: This (Default), Latest, LatestMajor  
- (Optional) String filter: Filter for properties to be returned  
- (Optional) Boolean includeAllowableActions: False (default)  
- (Optional) Enum includeRelationships: none (default), source, target, both |
| Outputs | - Collection propertyCollection  
- Collection allowableActionCollection |
| Exceptions | - Common Exceptions  
- FilterNotValidException |
| Notes | - If "includeAllowableActions" is TRUE, the repository will return the allowable actions for the current user for the object as part of the output.  
- "IncludeRelationships" indicates whether relationships are also returned for the object. If it is set to "source" or "target", relationships for which the returned object is a source, or respectively a target, will also be returned. If it is set to "both", relationships for which the returned object is either a source or a target will be returned. If it is set to "none", relationships are not returned.  
- Does not return the content-stream of a document  
- PropertyCollection includes changeToken (if applicable to repository) |

### Questions

### getContentTypeStream

| Description | The service returns the content-stream for a document. This is the only service that returns content-stream. |
| Inputs | - ID repositoryId: Repository Id  
- ID documentId: Document to return the content-stream  
- (Optional) Integer offset:  
- (Optional) Integer length: |
## updateProperties

<table>
<thead>
<tr>
<th>Description</th>
<th>This service updates properties of the specified object. As per the data model, content-streams are not properties</th>
</tr>
</thead>
</table>
| Inputs      | ID repositoryId: Repository Id  
ID objectId  
(Optional) String changeToken  
Collection propertyCollection - Subset list of Properties to update  
  o Name, Description, ... |
| Outputs     | ID objectId |
| Exceptions  | Common Exceptions  
UpdateConflictException  
VersioningException |
<table>
<thead>
<tr>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>• Preserves the ID of the object</td>
</tr>
<tr>
<td>• Subset of properties: Properties not specified in this list are not changed</td>
</tr>
<tr>
<td>• To remove a property, specify property with no value</td>
</tr>
<tr>
<td>• If an attempt is made to update a read-only property, throw ConstraintViolationException.</td>
</tr>
<tr>
<td>• If a ChangeToken is provided by the repository when the object is retrieved, the change token MUST be included as-is when calling updateProperties.</td>
</tr>
<tr>
<td>• For Multi-Value properties, the whole list of values MUST be provided on every update.</td>
</tr>
<tr>
<td>• Use getAllowableActions to identify whether older version specified by ID is updatable.</td>
</tr>
<tr>
<td>• If this is a private working copy, some repositories may not support updates.</td>
</tr>
<tr>
<td>• Because repositories MAY automatically create new Document Versions on a user’s behalf, the objectID returned may not match the one provided as an input to this method.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Questions</th>
</tr>
</thead>
</table>

---

**moveObject**

<table>
<thead>
<tr>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Moves the specified filed object from one folder to another</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Inputs</th>
</tr>
</thead>
<tbody>
<tr>
<td>• ID repositoryId: Repository Id</td>
</tr>
<tr>
<td>• ID objectId: Object to be moved</td>
</tr>
<tr>
<td>• ID targetFolderId: The target folder to be moved into</td>
</tr>
<tr>
<td>• (Optional) ID sourceFolderId: The source folder to be moved out of</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Outputs</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Exceptions</th>
</tr>
</thead>
<tbody>
<tr>
<td>• Common Exceptions</td>
</tr>
<tr>
<td>• UpdateConflictException</td>
</tr>
<tr>
<td>• VersioningException</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>• Preserves the ID of the moved object.</td>
</tr>
<tr>
<td>• Does not modify the source and the target folders.</td>
</tr>
<tr>
<td>• If Object is multi-filed and source folder is not specified, throw IllegalArgumentException.</td>
</tr>
<tr>
<td>• If Object is not filed, throw OperationNotSupportedException. This includes the root folder.</td>
</tr>
<tr>
<td>• If the object's Type is not one of the “Allowed_Child_Object_Types” for the target Folder, throw ConstraintViolationException.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Questions</th>
</tr>
</thead>
</table>
### deleteObject

<table>
<thead>
<tr>
<th>Description</th>
<th>Deletes specified object</th>
</tr>
</thead>
</table>
| Inputs      | • ID repositoryId: Repository Id  
• ID objectId |
| Outputs     |                           |
| Exceptions  | • Common Exceptions  
• UpdateConflictException |
| Notes       | • If the object is a Folder with at least one child, throw ConstraintViolationException.  
• If the object is the Root Folder, throw OperationNotSupportedException.  
• When a filed object is deleted, it is removed from all folders it is filed in.  
• This service deletes a specific version of a document object. To delete all versions, use deleteAllVersions().  
• Deletion of a private working copy (checked out version) is the same as to cancel checkout. |
| Questions   |                           |

### deleteTree

<table>
<thead>
<tr>
<th>Description</th>
<th>Deletes the tree rooted at specified folder (including that folder)</th>
</tr>
</thead>
</table>
| Inputs      | • ID repositoryId: Repository Id  
• ID folderId  
• Enum unfileNonfolderObjects:  
  o Unfile – unfile all non-folder objects from folders in this tree. They may remain filed in other folders, or may become unfiled.  
  o DeleteSingleFiled – delete non-folder objects filed only in this tree, and unfile the others so they remain filed in other folders.  
• (Optional) Bool continueOnFailure: False (Default) |
| Outputs     | • Collection failedToDelete - List of object IDs that failed to delete (if continueOnFailure is FALSE, then single object ID) |
| Exceptions  | • Common Exceptions  
• UpdateConflictException |
| Notes       | • If a non-folder object is removed from the last folder it is filed in, it can continue to survive outside of the folder structure if the repository supports the “Unfiling” capability. |
• If the specified folder is the Root Folder, throw OperationNotSupportedException.
• If unfiling is not supported, throw OperationNotSupportedException if deleteTree is called with Unfile.
• For repositories that support version-specific filing, this may delete some versions of a document but not necessarily all versions. For repositories that do not support version-specific filing, if a document is to be deleted, all versions are deleted.
• This is not transactional.
  o However, if DeleteSingleFiled is chosen, then having the objects unfiled is not sufficient if some objects fail to delete. The user MUST be able to re-issue command (recover) from the error by using the same tree.
• Does not specify the order in which delete will happen
  o However, any objects that are not deleted (e.g. because a previous object failed to delete), they MUST remain valid CMIS objects (including any applicable filing constraint for each object).

Questions

setContentView

<table>
<thead>
<tr>
<th>Description</th>
<th>Sets (creates or replaces) the content-stream for the specified document object.</th>
</tr>
</thead>
</table>
| Inputs      | • ID repositoryId: Repository Id  
• ID documentId  
• (Optional) Bool overwriteFlag: Default True  
• ContentStream contentStream |
| Outputs     | • ID documentId  |
| Exceptions  | • Common Exceptions  
• StorageException  
•  
• ContentAlreadyExistsException  
• StreamNotSupportedException  
• UpdateConflictException  
• VersioningException |
| Notes       | • If OverwriteFlag is FALSE, throw ContentAlreadyExistsException if content-stream already exists.  
• This is considered an update of the document object.  
• If this is a private working copy, some repositories may not support updates.  
• If the Document has “Content_Stream_Allowed” set to FALSE, throw ConstraintViolationException  
• Some CMIS protocol bindings may choose not to explicitly implement a “setContentStream” method, in cases where the protocol itself provides built-in mechanisms for updating byte streams. (E.g. in the ATOM/REST binding, content streams may be updated via standard HTTP puts on an “edit-media” URL, rather than a CMIS-specific “setContentStream” URL). See Part II of the CMIS specification for |
Because repositories MAY automatically create new Document Versions on a user's behalf, the objectid returned may not match the one provided as an input to this method.

### deleteContentStream

**Description**
Deletes the content-stream of the specified document. This does not delete properties. If there are other versions this does not affect them, their properties or content. This does not change the ID of the document.

**Inputs**
- ID repositoryId: Repository Id
- ID documentId

**Outputs**

**Exceptions**
- Common Exceptions
- StorageException
- VersioningException
- StreamNotSupportedException
- UpdateConflictException

**Notes**
- If content-stream is required for documents of this type, throw ConstraintViolationException.
- If versioning does not allow modification to this version, VersioningException is thrown.
- This is considered an update of the document object.
- If this is a private working copy, some repositories may not support updates.

**Questions**

### MULTI-FILING SERVICES

The multi-filing services, which are supported if-and-only-if the Repository supports the optional multi-filing capability, is used to file/un-file objects into/from Folders.

This service is NOT used to create or delete objects in the Repository.

### addObjectToFolder

**Description**
Adds an existing non-folder, fileable object to a folder.
## removeObjectFromFolder

### Description
Removes a non-folder child object from a folder or from all folders. This does not delete the object and does not change the ID of the object.

### Inputs
- ID repositoryId: Repository Id
- ID objectId: The object to be removed from a folder
- (Optional) ID folderId: The folder to be removed from. If not specified, remove from all folders.

### Outputs

### Exceptions
- Common Exceptions

### Notes
- If unfiling is not supported, and an object is to be removed from the last folder it exists in, or is to be removed from all folders, throw OperationNotSupportedException.
- This does not change the status of the Folder

### Questions

---

**DISCOVERY SERVICES**

The Discovery services are used to search queryable objects within the Repository.
| Description | Queries the repository for queryable object based on properties or an optional full-text string. Relationship objects are not queryable. Content-streams are not returned as part of query. |
| Inputs | **ID repositoryId**: Repository Id  
  **String statement**: Query statement  
  **(Optional) Boolean searchAllVersions**: False (Default)  
  **(Optional) Boolean includeAllowableActions**: False (default)  
  **(Optional) Enum includeRelationships**: none (default), source, target, both  
  **(Optional) int maxItems**: 0 = Repository-default number of items (Default)  
  **(Optional) int skipCount**: 0 = Start at first position (Default) |
| Outputs | **Collection objectCollection**  
  This collection represents a result table produced by the query statement. Typically each row of this table corresponds to an object, and each column corresponds to a property or a computed value as specified by the SELECT clause of the query statement. A CMIS SQL 1.0 query without JOIN always produces one object per row.  
  **Bool hasMoreItems** |
| Exceptions | **Common Exceptions** |
| Notes | **If SearchAllVersions is True, and CONTAINS() is used in the query, OperationNotSupported will be thrown if full-text search is not supported or if the repository does not have previous versions in the full-text index.**  
  **Returns set of objects from (skipCount, maxItems+skipCount)**  
  **If no “maxItems” value is provided, then the Repository will determine an appropriate number of items to return. How the Repository determines this value is repository-specific and opaque to CMIS.**  
  **If “includeAllowableActions” is TRUE, the repository will return the allowable actions for the current user for each result object in the output table as an additional multi-valued column containing computed values of type string, provided that each row in the output table indeed corresponds to one object (which is true for a CMIS SQL 1.0 query without JOIN).**  
  If each row in the output table does not correspond to a specific object and “includeAllowableActions” is TRUE, then InvalidArgumentException will be thrown.  
  It is recommended that “includeAllowableActions” be used with query statements without JOIN, and that the Object ID property or “*” be included in the SELECT list.  
  **"IncludeRelationships" indicates whether relationships are also returned for each returned object. If it is set to "source" or "target", relationships for which the returned object is a source, or respectively a target, will also be returned. If it is set to "both", relationships for which the returned object is either a source or a target will**
The Versioning service is used to perform operations on Documents like check-in and check-out, as well as to locate and act on the version history for Documents.

### checkOut

<table>
<thead>
<tr>
<th>Description</th>
<th>Create a private working copy of the object, copies the metadata and optionally content. It is up to the repository to determine if updates to the current version (not PWC) and prior versions are allowed if checked-out.</th>
</tr>
</thead>
</table>
| Inputs                           | • ID repositoryId: RepositoryId  
• ID documentId: ObjectId of Doc Version to checkout                                                                                                                                       |
| Outputs                          | • ID documentId: ObjectId of Private Working Copy  
• Bool contentCopied                                                                                                                  |
| Exceptions                       | • Common Exceptions  
•  
• VersioningException  
• StorageException                                                                                                                  |
| Notes                            | • It is repository-specific to determine the scope of visibility to the private working copy  
• Other users not in the scope of checkout will see the public (pre-checkout) version while those in scope will be able to work on the checked-out version.  
• Copying content on checkout or not is repository-specific.  
• CheckOut() may remove update permission on prior versions  
• CheckOut() on a non-document object will throw OperationNotSupportedException.  
• Some repositories may not support updating of private working copies and the updates MUST be supplied via checkIn() |
| Questions                        |                                                                                                                                  |

### cancelCheckOut

<table>
<thead>
<tr>
<th>Description</th>
<th>Reverses the effect of a check-out. Removes the private working copy of the checked-out document object, allowing other documents in the version series to be checked out again.</th>
</tr>
</thead>
</table>
checkIn

**Description**
Makes the private working copy the current version of the document.

**Inputs**
- ID repositoryId: Repository Id
- ID documentId: ObjectId of the private working copy
- (Optional) Boolean major: True (Default)
- (Optional) Property bag
- (Optional) ContentStream stream
- (Optional) String CheckinComment

**Outputs**
- ID documentId: ID for the new version of the document.

**Exceptions**
- Common Exceptions
- StreamNotSupportedException
- StorageException
- UpdateConflictException

**Notes**
- It is left to the repository to determine who can check-in a document.
- CheckinComment is persisted if specified.
- For repositories that do not support updating private working copies, all updates MUST be set on the check-in service.
- If Document is not checked out, throw OperationNotSupportedException.
- If the Document has “Content_Stream_Allowed” set to FALSE, and a call is made to checkin that includes a content-stream, throw ConstraintViolationException.

**Questions**

getPropertiesOfLatestVersion
<table>
<thead>
<tr>
<th>Description</th>
<th>Returns the properties of the latest version, or the latest major version, of the specified version series</th>
</tr>
</thead>
</table>
| Inputs      | - ID repositoryId: Repository Id  
- ID versionSeriesId: Version series Id  
- (Optional) Boolean majorVersion: Whether or not to return the latest major version. Default=FALSE.  
- (Optional) String filter: Filter for properties to be returned |
| Outputs     | - Collection propertyCollection |
| Exceptions  | - Common Exceptions  
- FilterNotValidException |
| Notes       | - Does not return the content-stream of document.  
- PropertyCollection includes changeToken (if applicable to repository).  
- If the latest major version is requested and the series has no major version, throw ObjectNotFoundException. |
| Questions   |                                                                                                     |

**getAllVersions**

<table>
<thead>
<tr>
<th>Description</th>
<th>Returns the list of all document versions for the specified version series, sorted by CREATION_DATE descending.</th>
</tr>
</thead>
</table>
| Inputs      | - ID repositoryId: Repository Id  
- ID versionSeriesId: Version series Id  
- (Optional) String filter: Property Filter |
| Outputs     | - Collection documentCollection - Set of objects containing Ids and specified properties |
| Exceptions  | - Common Exceptions  
- FilterNotValidException |
| Notes       | - Returns all versions the user can access including checked-out version and private working copy. |
| Questions   |                                                                                                     |

**deleteAllVersions**
### Description
Deletes all document versions in the specified version series.

### Inputs
- ID repositoryId: Repository Id
- ID versionSeriesId: Version series Id

### Outputs

### Exceptions
- CommonExceptions

### Notes
- Throw ConstraintViolationException if version can not be deleted due to repository integrity exception (e.g., branching and child versions exist)

---

**RELATIONSHIPS SERVICES**

The Relationship Services are used to read the relationship information on an independent object.

#### getRelationships

### Description
Returns a list of relationships associated with the object, optionally of a specified relationship type, and optionally in a specified direction

### Inputs
- ID repositoryId: Repository Id
- ID objectId: The object with which relationships are associated with
- (Optional) Enum direction: Where I am Source (Default), Target, Either
- (Optional) ID typeld: Relationship Type
- (Optional) Bool includeSubRelationshipTypes: False (Default)
- (Optional) String filter: Property filter
- (Optional) Boolean includeAllowableActions: False (default)
- (Optional) int maxItems: 0 = Unlimited
- (Optional) int skipCount: 0 = start at beginning

### Outputs
- List of relationship IDs and specified properties
- Boolean hasMoreItems

### Exceptions
- Common Exceptions
- FilterNotValidException

### Notes
- If typeId is not specified, return relationships of any type.
- Ordering is consistent across requests and repository specific.
- If “includeAllowableActions” is TRUE, the repository will return the allowable actions for the current user for each relationship object as part of the output.
POLICY SERVICES

The Policy Services are used to apply or remove a policy object to a controllable object.

### applyPolicy

<table>
<thead>
<tr>
<th>Description</th>
<th>Applies a policy object to a target object.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Inputs</td>
<td>• ID repositoryId: Repository Id</td>
</tr>
<tr>
<td></td>
<td>• ID policyId: Policy object Id</td>
</tr>
<tr>
<td></td>
<td>• ID objectId: Target object Id</td>
</tr>
<tr>
<td>Outputs</td>
<td></td>
</tr>
<tr>
<td>Exceptions</td>
<td>• Common Exceptions</td>
</tr>
<tr>
<td></td>
<td>• UpdateConflictException</td>
</tr>
<tr>
<td></td>
<td>• VersioningException</td>
</tr>
<tr>
<td></td>
<td>• StorageException</td>
</tr>
<tr>
<td>Notes</td>
<td>• If the target object is not controllable, throw ConstraintViolationException.</td>
</tr>
<tr>
<td></td>
<td>• If the policy can not be applied to the target object due to a repository-specific constraint, throw ConstraintViolationException.</td>
</tr>
<tr>
<td>Questions</td>
<td></td>
</tr>
</tbody>
</table>

### removePolicy

<table>
<thead>
<tr>
<th>Description</th>
<th>Removes a previously applied policy from a target object. The policy object is not deleted, and may still be applied to other objects.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Inputs</td>
<td>• ID repositoryId: Repository Id</td>
</tr>
<tr>
<td></td>
<td>• ID policyId: Policy object Id</td>
</tr>
<tr>
<td></td>
<td>• ID objectId: Target object Id</td>
</tr>
<tr>
<td>Outputs</td>
<td></td>
</tr>
<tr>
<td>Exceptions</td>
<td>• Common Exceptions</td>
</tr>
<tr>
<td></td>
<td>• UpdateConflictException</td>
</tr>
</tbody>
</table>

Questions

---

<table>
<thead>
<tr>
<th>POLICY SERVICES</th>
</tr>
</thead>
<tbody>
<tr>
<td>The Policy Services are used to apply or remove a policy object to a controllable object.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>applyPolicy</th>
</tr>
</thead>
<tbody>
<tr>
<td>Description</td>
</tr>
<tr>
<td>Inputs</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>Outputs</td>
</tr>
<tr>
<td>Exceptions</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>Notes</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>Questions</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>removePolicy</th>
</tr>
</thead>
<tbody>
<tr>
<td>Description</td>
</tr>
<tr>
<td>Inputs</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>Outputs</td>
</tr>
<tr>
<td>Exceptions</td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>
**getAppliedPolicies**

<table>
<thead>
<tr>
<th>Description</th>
<th>Gets the list of policy objects currently applied to a target object.</th>
</tr>
</thead>
</table>
| Inputs      | • ID repositoryId: Repository Id  
               • ID objectId: Target object Id  
               • (Optional) String filter: Filter specifying which properties to return |
| Outputs     | • Result set specified by Filter for each policy object applied to the target object |
| Exceptions  | • ObjectNotFoundException  
               • FilterNotValidException |
| Notes       | • Only policies that are directly (explicitly) applied to the target object are returned. Any notion of "indirectly applying" a policy to an object (e.g. through inheritance) is outside the scope of CMIS 1.0.  
               • If the target object is not controllable, throw ConstraintViolationException.  
               • If Filter is not valid, throw FilterNotValidException. |
| Questions   | |