# Content Management Interoperability Services

# Browser Bindings

Version:

Draft v0.63, 29. May 2009

# Table of Contents

# 1 Preface

This document describes the "Browser Bindings" of CMIS. In early testing of the two existing Bindings (SOAP and AtomPub), despite being HTTP based, basic use cases can not be covered when using a standard web browser. The intention of this document is to cover that gap and make CMIS a "Browser-enabled" protocol that delivers on the promise of "Browser mash-up" as a use-cases.

## 1.1 Status

TODO: status of this document

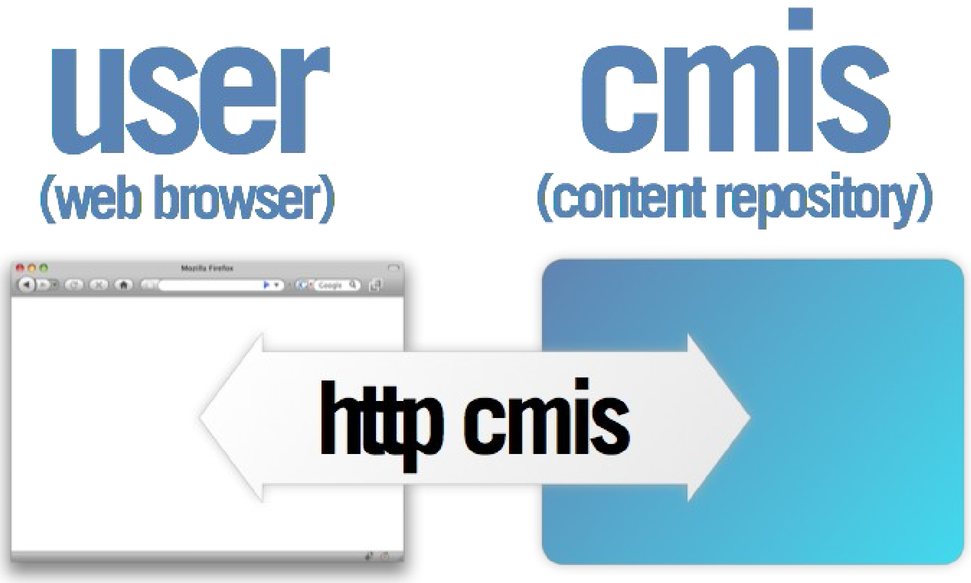
Known TODOs:

- describe error handling
- map CMIS exceptions
- relationship objects creation/modification/deletion

## 1.2 CMIS version

Unless otherwise stated all references point to Version 0.63.c of the specification "CMIS Part I - Domain Model".

# 2 Introduction



## 2.1 Motivation

Currently the SOAP and AtomPub bindings of CMIS don't lend themselves to a straight forward consumption by standard web browsers and require the creation and use of large java script libraries for the most basic use cases. Other use cases as simple as “uploading a file” even cannot be covered at all with a browser talking to a CMIS repository.

The lack of this ability even triggered conversations in the TC about having server-sided proxies that would again speak a proprietary protocol just to able to translate CMIS to a protocol that allows for mash-ups and browser interaction, which seems like very undesirable effect when specifying an HTTP based protocol for consumption by browser.

Since mash-ups and simple browser interaction is a stated goal and use case of CMIS this document tries to close this gap by introducing a lightweight and easy to consume binding that makes CMIS effective in a standard Web environment.

## 2.2 Goals

The goal of this specification is to offer a simple protocol that is efficient from a network perspective and intuitive to use from a standard (javascript enabled) web browser. This document does not attempt to cover the full breadth of CMIS and will always compromise edge-cases for simplicity. Also this binding does not necessarily attempt to cover all the features and domain model expressed in CMIS, but there are no implicit limitations on how much of the domain model is covered.

The binding should be designed in a fashion that allows web browsers and other web infrastructure (such as spiders) to easily introspect the contents of the repository and interact with it.

The overall goal of this document is to produce a binding that is intuitive to use and hence ease of use is the most important guiding principal.

# 3 Use Cases

## 3.1 Simple Reading

A simple browser based application should be able to retrieve content from the repository and make sense of the properties and display documents (content stream).

The specification should allow the simple build of a “CMIS browser application” with just a few lines of javascript to satisfy the mash-up use case.

After browsing for a “document”, the “document” should be displayed with a proper resolution of all the relative references pointing to other documents in the CMIS repository.

When displaying a document in a browser (such as an HTML file) the relative URL references need to be kept intact, this includes for example typical HTML elements like

```
<a href=”other.html”>
```

or

```
<img src=”images/my.png”>.
```

## 3.2 Simple Writing

Creating a simple HTML form in a browser should be enough to update properties and the content stream. The names of properties (potentially the relative paths) are addressed using form field names in the browsers.

Just uploading a file is as simple as having one

```
<input type=”file”>
```

posting the form to the parent folders child list.

To create a custom way to update or create content in CMIS is as simple as setting up a simple web form and POSTing it to the respective URL identifying the document or folder.

### 3.3 Batch Writing

Posting a number of documents to the repository or modifying a larger number of properties in the repository is as simple as creating a more complex form.

Given the fact that some operations are not expressed in a simple and efficient manner by the form fields name value pairs, a "patch" mechanism can be used that is applied against the repository. This provides a more efficient way from a network perspective but also allows javascript applications to interact with the repository from the browser client without having to simulate full form submissions.

### 3.4 Simple Search

Queries can be issued in an intuitive way exposing a full-text search engine style GET request that can be linked to. Appending a simple query parameter will issue the human readable query.

*Example:*

`Searching myfolder`

```
http://<host:port>/<mount-point>/myfolder.query.json?q=
```

## 4 Usage of the URL Space

This binding assumes that it is "mounted" into the URL space and has a potentially unidentified root path that is used as a prefix for all URLs managed by this binding. It is allowed and recommended that an implementation makes use of the URLs that are not identified by the combination of verbs below. For example it should be possible to have a complete WebDAV implementation or an existing implementation sharing the same URL space. An example of an application that could share the same URL space would for example be a simple HTML based repository browser to navigate folders similar to the usual directory listing found in FS based web servers.

### 4.1 Use of path and CMIS identifier

The path in the URL that is prefixed with the mount-point of this binding is used to identify a document and folder following the folder hierarchy exposed by the CMIS. At the same time any document or folder can be access using its identifier (*ObjectId*).

A document "d.doc" that is located in folder "myfolder" and has the identifier "1234" can be identified by the following urls.

*Example:*

```
http://<host:port>/<mount-point>/myfolder/d.doc
http://<host:port>/<mount-point>/[1234]
```

### 4.2 Use of extensions & "selectors"

Generally an implementor is encouraged to use file extensions that match the mime-types to enhance interoperability with filesystems that a user of CMIS may use to store data retrieved from the server. While for the documents this is probably mostly guided by the mime-type information and the documents provided by the end user, we choose to use the proper mime types to access meta information about a document. Responses that have a mime-type of json also have a file extension .json.

To further identify the appropriate operation that is requested on a particular CMIS object we introduce a period delimited so-called selector. A selector is very similar to a URL query parameter but exposes additional constraints and offers a slightly more static feel from a URL perspective.

The properties of a structure of a document "d.doc" that is located in folder "myfolder" with an identifier "6789" and has the identifier "1234" can be queried by issuing a GET operation to the following example urls.

*Example:*

```
Query variants to find properties within the object tree starting at
"myfolder"
```

```
http://<host:port>/<mount-point>/myfolder.json (get list of children)
http://<host:port>/<mount-point>/myfolder.1.json (get children)
http://<host:port>/<mount-point>/myfolder.infinity.json (get descendents)
http://<host:port>/<mount-point>/myfolder.infinity.json (get descendents)
http://<host:port>/<mount-point>/[6789].infinity.json (get descendents)
http://<host:port>/<mount-point>/myfolder/d.doc.json (get document)
http://<host:port>/<mount-point>/[1234].json (get document)
http://<host:port>/<mount-point>/myfolder.query.json?q=cmis (querys folder
and descendants for fulltext "cmis")
```

### 4.3 HTTP Methods & Response codes

In this specification we use GET for operations that are safe and do not modify the content and we use POST for writing.

Given the fact that there are a number of popular HTTP clients that do not implement HTTP properly an implementation is specifically allowed to take a more lenient approach to support some of those clients.

### 4.4 Reserved locations

The namespaces */cmis:object-types* and */cmis:unfiled* are mapped to the respective information. See Reading Object-Types and Access Unfiled Content, respectively.

# 5 Reading

## 5.1 Overview

All operation that read from the CMIS content repository in this binding use GET as a safe operation. CMIS Objects can be addressed either by the hierarchy exposed in the folder structure of CMIS or by their identifier. Both the path or the identifier are a appended to mount-point which for the sake of simplicity will be assumed to the root of a web server in all subsequent examples.

Both getting the content stream of a document but also getting the meta information (properties and relationships) are equally simply dealt with by simple GETs to intuitively meaningful URLs.

## 5.2 Format

To transport fine-grained about a CMIS object the Javascript Object Notation (JSON) was selected since it is both efficient and can be easily consumed by almost any programming language environment most importantly javascript in browsers. The easiest way to consume JSON is to call the eval function and get a object tree back.

### 5.2.1 Invalid JSON Characters

In order to form valid JSON keys and values, all CMIS identifiers, properties and attributes and their corresponding values must be properly escaped. See http://www.ietf.org/rfc/rfc4627.txt.

## *5.3 Addressing CMIS Objects*

### 5.3.1 Overview

TODO

### 5.3.2 Hierarchical Access

Hierarchical relationship between CMIS objects are exposed by the navigation services, that allow to traverse the hierarchy (or hierarchies) present in the repository.

The JSON representation of CMIS objects defined in the following sections allows to naturally map the ancestor/descendant relationships present between CMIS objects. Consequently, the relationships exposed in the JSON representation may be used to directly access a CMIS objects and its descendants.

To ease the usability of the JSON representation on the client side the following rules should be followed: TODO

- If present the Name of CMIS object should be used as key in the JSON representation.
- If no Name is available the implementation should define a human readable string that identifies the among it's siblings.
- The implementation is in charge of properly dealing with naming conflicts (see also Conflicts Between Child Objects).

*Example:*

Reading a folder.

Request:

```
GET /myfolder.json HTTP/1.1
```

Response:

```
{
"cmis:properties" :
  {
  "CreatedBy" : "uncled",
  "Name" : "myfolder",
  "ObjectId" : "50d9317a-3a95-401a-9638-333a0dbf04bb"
  "ObjectTypeId" : "Folder"
  …

  },
"cmis:relationships" : { … },
"child1" : { … },
"child2" : { … },
"child3" : { … }
}
```

### 5.3.3 Access by Identifier

An alternative way to retrieve the JSON representation of a CMIS object is the access by identifier. Instead of traversing the object hierarchy a CMIS object is addressed by its *ObjectId*.

In order to properly resolve the identifier and avoid any conflicts the *ObjectId* is enclosed in brackets. Similar to the hierarchical access the extension *.json* is appended.

*Example:*

```
Request:
```

```
GET /[50d9317a-3a95-401a-9638-333a0dbf04bb].json
```

```
Response:
```

```
HTTP/1.1 200 OK
Content-Type: text/plain; charset="utf-8"
Content-Length: xxxx

{
"cmis:properties" :
  {
  "ObjectId" : "50d9317a-3a95-401a-9638-333a0dbf04bb",
  "Name" : "myfolder",
  …

  },
"cmis:relationships" : { … },
"child" : {},
…

}
```

### 5.3.4 Access Unfiled Content

Unfiled CMIS objects that don't have a parent are also exposed under the reserved resource */cmis:unfiled*.

*Example:*

```
GET /cmis:unfiled/unfiledObject.json
```

### 5.3.5 Deep Reading

For all object types that allow children objects deep reading should be supported in order to allow for efficient retrieval of hierarchy information. This applies both to hierarchical access as well as to access by identifier.

In case of deep reading the JSON representation of descendant objects are included in the response as members of the JSON representation of the requested CMIS object. The depth may either be explicitly specified by the client or some implementation specific default. In any case the client is obligated to properly deal with the response containing several hierarchy levels.

#### 5.3.5.1 Client-Specified Depth

The user may optionally specify the desired depth in the request by adding a depth selector:

- Depth 0 indicates that children objects should not be included in the response.
- Depth 1 indicates that only the child objects should be included.

- Any value > 1 requests for descendants up to the specified level.
- Infinite depths is indicated by an '*infinity*' selector.
  Note however that the implementation may restrict the maximum number of levels to be included in the response. In this case an empty JSON object present with any of the descendants indicates that the hierarchy isn't completed included in the response.

*Example:*

Reading a folder without child objects (Depth 0). For simplicity properties, relationships are omitted.

Request:

```
GET /myfolder.0.json HTTP/1.1
```

Response:

```
{
"child1" : {},
"child2" : {},
…

}
```

*Example:*

Reading a folder including it's children (Depth 1). For simplicity properties, relationships are omitted.

Request:

```
GET /myfolder.1.json HTTP/1.1
```

Response:

```
{
"child1" :
  {
  "grandchild11" : {},
  …

  },
…

}
```

*Example:*

Reading a folder and it's descendants (Depth > 1). For simplicity properties, relationships are omitted. Every object and depth I only has a single child. The children of the object at max-depths are included with an empty JSON value.

Request:

```
GET /myfolder.4.json HTTP/1.1
```

Response:

```
{
"child1" :
  {
  "grandchild11" :
    {
    "depth3" :
      {
      "depth4 :
        {
        "depth5" : {}
        }
      }
    }
  }
}
```

*Example:*

`Request to read the complete tree starting at a given folder.`

```
GET /myfolder.infinity.json HTTP/1.1
```

#### 5.3.5.2 Default Depth

If the client does not specify the desired depth, some implementation specific default depth should be used. The default depth may vary between individual CMIS objects or object-types. In any case the client must be able to deal with responses containing multiple hierarchy levels not explicitly requested.

#### 5.3.5.3 Configurable Depth

An implementation may allow to configure the default depth. Possible approaches include configuration by object-type, by name pattern or by hierarchy.

## 5.4 Reading CMIS Objects

CMIS defines the following typed Objects:

- Document object (see Reading Documents)
- Folder objects (see Reading Folders)
- Relationship objects (see Reading Relationship Objects)

Reading a CMIS Object by default includes information about

- child objects (if present)
- properties.

If a given JSON representation of a CMIS object recursively contains the complete JSON representation depends on the specified request selectors:

The DEPTH selector (see Deep Reading) in turn specifies whether and to which extend the descendants CMIS objects will be included in the response.

The structure of a CMIS object is defined by it's object-type (see Reading Object-Types).

### 5.4.1 Reading Documents

Every document object is represented by a single JSON object.

#### 5.4.1.1 Document Properties

The properties of a CMIS document object are added as members to the reserved *cmis:properties* JSON object (see Reading Properties).

#### 5.4.1.2 Content Stream

If a CMIS document objects provides a Content Stream[1] the following rules apply to the JSON representation: The binary itself is not included in the JSON object. Instead the implementation must provide the *ContentStreamUri*[2] property, which is included along with the other document properties.

In order to read the binary the consumer of the JSON uses the JSON value (i.e. the URI of the Content Stream) to access the value itself. The MIME Media Type of the Content Stream and it's length are exposed both as JSON property and as appropriate HTTP header upon requesting the Content Stream using its URI. If the implementation provides a separate display name (*ContentStreamFilename*) it can be retrieved from the property list of the corresponding CMIS document object.

The *contentStreamAllowed* attribute of the document object type informs about the ability of a document to have a content stream altogether. This information can be retrieved along with the attributes of the object-type definition (see Reading Object-Types).

### 5.4.2 Reading Folders

Every folder object is represented by a single JSON object.

#### 5.4.2.1 Folder Properties

The properties of a CMIS folder object are added as members to the reserved *cmis:properties* JSON object (see Reading Properties).

#### 5.4.2.2 Listing Child Objects

The list of child objects present within a given CMIS folder are added as JSON members to the JSON object representing the folder itself:

The CMIS *Name* property acts as JSON key (see also Invalid JSON Characters), while the corresponding value consists of a JSON object. The JSON object can either be empty (see Deep Reading) or contain the JSON representation of the child object. The latter depends on the object type of each individual child.

*Example:*

```
JSON representation of child objects is empty.
```

```
{
"cmis:properties" : { … },
"child1" : {},
"child2" : {}
}
```

*Example:*

```
JSON representation of child objects included.
```

```
{
```

[1] See 2.3.1 Content Stream

[2] See 2.7.4.1.2 Document Object-Type Property Definitions

```
"cmis:properties" : { … },
"child1" :
  {
  "cmis:properties" : { … },
  …

  },
"child2" :
  {
  "cmis:properties" : { … },
  "grandchild" : { … },
  …

  }
}
```

#### 5.4.2.3 Descendants of a Folder

Descendant objects other than the direct children (see Listing Child Objects) may or may not be part of the JSON representation of a folder. This behavior is controlled by implicit default depth or the depth explicitly specified upon requesting the JSON representation of a folder object (see Deep Reading for details).

The descendants of a folder object is included in the JSON in accordance to the behavior described for child objects: They are added as JSON members to the JSON representation of their direct parent (i.e. the folder which they are children of).

#### 5.4.2.4 Conflicts Between Child Objects

The child objects of an object may collide in the JSON representation as the *Name* of a child object is not required to be unique among the children of a single folder.

In this case the conflicting JSON key (or keys) (aka *Name* property) will get an index appended. The index is enclosed by brackets in order to avoid collisions with any trailing numbers contained in the name. The index of the first JSON key is optional and may be omitted. The index is an integer greater than or equal to 1.

*Example:*

```
{
"cmis:properties" : { … },
"child" : { … },
"child[2]" : { … },
"child[3]" : { … },
"anotherC" : { … },
…

}
```

### 5.4.3 Reading Relationship Objects

CMIS relationship objects are defined as dependent objects that should not be file-able. In order to make the available in the hierarchical access to the repository they are appended to the JSON representation of their source object.

Therefore a reserved *cmis:relationships* JSON object is created that collects the relationships of the specified source object. Each relationship is represented by a member of the *cmis:relationships*

JSON object using the relationship's *ObjectId* as key. It's value is a new JSON object that lists the properties of the relationship object as simple name-value pairs.

#### 5.4.3.1 Relationship Properties

The properties of a CMIS relationship object are reflected as simple JSON members of the JSON object associated with the corresponding relationship key (*ObjectId*).

*Example:*

```
{

"cmis:properties" : { … },
"cmis:relationships" :
  {
  "myRelationshipId" :
    {
    "CreatedBy" : "uncled",
    …

    "TargetId"  : "b410b572-9363-4f9c-9f70-04593652179b"
    }
  },
"child" : { … },
…

}
```

## 5.5 Reading Properties

The properties of all independent CMIS objects are included as members of a reserved JSON object whose key is *cmis:properties* that is automatically added as member to the JSON object representing a given CMIS object. The CMIS properties in turn are reflected as simple name-value members of the *cmis:properties* entry, where

- the property *Name* attribute acts as key,
- the property value(s) are included as JSON value of type string, number, boolean or in case of a multi-valued property as an JSON array of the JSON values mentioned (Single-Valued Properties, Multi-Valued Properties).

### 5.5.1 Property Types

The following property types defined by CMIS can be mapped to the types defined by JSON:

| CMIS Property Type | JSON Property Value | Comment |
|---|---|---|
| *String* | *string* | |
| *Decimal* | *number* | TODO |
| *Integer* | *number* | TODO |
| *Boolean* | *true, false* | |

The remaining CMIS property types

| | | |
|---|---|---|
| *DateTime* | - | |
| *URI* | - | |
| *ID* | - | |

| | | |
|---|---|---|
| *XML* | - | |
| *HTML* | - | |

have no corresponding type in JSON. Instead their string representation properly escaped according to the requirements will be used as JSON value. The type information of these properties can be determined by reading the property definition (see Property Definition6).

### 5.5.2 Property Value

#### 5.5.2.1 Single-Valued Properties

The value of single-value CMIS properties are directly included in the JSON. The conversion from CMIS property type to JSON follows the rules described in section Property Types.

*Example:*

```
Response containing a String properties.
```

```
{
"cmis:properties" : { … },
"mydocument" :
  {
  "cmis:properties" :
    {
    "CreatedBy" : "uncled",
    "Name" : "mydocument",
    …

    }
  }
}
```

*Example:*

```
Response containing a Boolean property.
```

```
{
"cmis:properties" : { … },
"mydocument" :
  {
  "cmis:properties" :
    {
    "IsImmutable" : true,
    …

    }
  }
}
```

*Example:*

```
Response containing an Integer property.
```

```
{
"cmis:properties" : { … },
"mydocument" :
  {
  "cmis:properties" :
    {
    "ContentStreamLength" : 34497,
    …
    }
  }
}
```

*Example:*

```
Response containing other property types. In this example DateTime, ID and Uri.
The remaining types (XML, HTML) are dealt with accordingly.
```

```
{
"cmis:properties" : { … },
"mydocument" :
  {
  "cmis:properties" :
    {
    "CreationDate" : "2009-05-19T11:37:55.431Z",
    "ObjectId" : "50d9317a-3a95-401a-9638-333a0dbf04bb",
    "Uri" : "http://localhost:4302/cmis/repo/myfolder/mydocument.doc",
    …

    }
  }
}
```

#### 5.5.2.2 Multi-Valued Properties

The JSON value of a multi-value CMIS property consists of a JSON non-nested JSON array. The individual values themselves are transformed to JSON values as described in section Property Types. This also applies if the value array only contains a single element.

*Example:*

```
   Response containing a multi-valued property with multiple values
```

```
{
"cmis:properties" :
  {
  "AllowedChildObjectTypeIds" : { "typeId1","typeId2","typeId3" },
  …

  }
}
```

*Example:*

```
   Response containing a multi-valued property with a single value
```

```
{
"cmis:properties" :
  {
  "AllowedChildObjectTypeIds" : { "typeId1" },
  …

  }
}
```

#### 5.5.2.3 "Value not set" state

Properties that are in a "value not set" state[3] are not included in the JSON. The same applies for empty value lists if a property is multi-valued, which is not allowed within the scope of CMIS.

[3] See 2.2.1 Property

### 5.5.3 Conflicts between Properties

The properties of a single CMIS object will never conflict as “within an object, each property is uniquely identified by its name”[4].

## *5.6 Reading Object-Types*

The object-type of a CMIS object can be determied from the *ObjectTypeId* property that is exposed in the JSON representation along with the other properties below *cmis:properties*. Object-type definitions are mapped as members of a reserved *cmis:object-types* resource. The JSON representation of the object-type definition can then be accessed by *displayName* or by *typeId*.

*Example:*

```
Request to retrieve the definition of the 'myObjectType' object-type.
```

```
GET /cmis:object-types/myObjectType.json HTTP/1.1
```

The complete set of object-type definitions can be obtained by inserting the *infinity* depth selector upon requesting the *cmis:object-types* resource.

*Example:*

```
Request to retrieve all object-type definitions.
```

```
GET /cmis:object-types.infinity.json HTTP/1.1
```

### 5.6.1 Object-Type Properties Definitions

The property definitions of CMIS a object-type are exposed as members of the JSON representation of the object-type below a reserved *cmis:properties* entry. The property *Name* will be used as key, while the property attributes are inserted as key-value paired members below.

*Example:*

```
Request:
```

```
GET /cmis:object-types/myObjectType.json HTTP/1.1
```

```
Response:
```

```
HTTP/1.1 200 OK
Content-Type: text/plain; charset="utf-8"
Content-Length: xxxx

{
"cmis:properties" :
  {
  "Name" :
    {
    "name" : "Name",
    "propertyType" : "String",
    "description" : "A description of the Name property.",
    "cardinality" : "Single",
    "Inherited" : false,
    …
    },
  "myProperty" :
```

4 2.2 Object

```
    {
    "propertyType" : "Integer",
    "defaultValue" : 13,
    "minValue" : 1,
    "maxValue" : 50,
    …
    }
  },
  …

}
```

### 5.6.2 Object-Type Attributes

The attributes of CMIS object-types are exposed as members of the JSON representation of the object type. The attribute name acts as key, while the attribute value acts as JSON value.

*Example:*

`Request:`

```
GET /cmis:object-types/myObjectType.json HTTP/1.1
```

`Response:`

```
HTTP/1.1 200 OK
Content-Type: text/plain; charset="utf-8"
Content-Length: xxxx

{
"displayName": "myObjectType",
"baseType" : "Document",
"creatable" : true,
"description" : "Object-type derived from 'Document'.",
"fileable" : true,
"queryable" : true,
…

"cmis:properties" : { … }
}
```

#### 5.6.2.1 Conflicts between Attributes

The set of attributes present with a CMIS object are defined by the repository. While CMIS does not explicitly require attribute names  to be unique among the attributes of a given object, it is expected that conflicts don't occur as attributes are not inherited to derived object. Handling unexpected conflicts is therefore left to the implementation.

### 5.6.3 Property Definition[5]

The characteristics of a property of a CMIS object is define by it's property definition, which forms part of the corresponding object-type definition. The property definition consists of a set of attributes, which are exposed as part of the object-type definition. Similar to the attributes of the object-type itself, that attributes are listed with each property definition.

[5] See  2.7.3 Object-Type Property Definitions

##### 5.6.3.1 Conflicts between Attributes

The set of attributes present with CMIS property are defined by the repository. While CMIS does not explicitly require attribute names  to be unique among the attributes of a given property, it is expected that the repository is able to detect and handle conflicts arising from inheritance of properties between object-types.

### 5.7 Reading Hierarchy Information

#### 5.7.1 Hierarchical Structure of a CMIS Repository

Hierarchical relationship between two CMIS objects can be determined through the navigation services[6]. With the structure defined by the browser binding simple ancestor/descendant relationships can easily be determined both from the request URL and from the JSON objects returned upon reading the repository.

In contrast unfiled content as well as object-type definitions that don't follow the parent-child relationship present with folders and documents are exposed under the corresponding reserved locations (see Access Unfiled Content and Reading Object-Types, respectively).

TODO: define how to determine multiple parents if present.

# 6 Writing

### 6.1 Overview

TODO

### 6.2 Format

The browser binding defines two variants of writing to a CMIS repository:

- Simple writing
- Batched writing.

Both are relying on standard functionality defined by HTTP 1.1 POST request. The batched writing in addition defines a json-like diff format and is used to cover multiple or complex operations within a single requests.

Simple writing is covered in sections Create CMIS Objects, Delete CMIS Objects and Writing Properties. The details of batched writing are described in section Batched Writing.

### 6.3 Writing CMIS Objects

#### 6.3.1 Create CMIS Objects

To create CMIS objects the URL of the form POST points to a parent Object and appends a trailing slash. This indicates that a new CMIS object should be created as opposed to the omission of the trailing slash which would constitute an update of an existing object identified in the URL.

Properties defined for the object-type to be created may be specified during the create request by adding corresponding request parameters, whose name identifies the object property, while the desired object value is passed as parameter value.

This behavior also allows to explicitly define the desire object-type for the CMIS object to be created. If an *ObjectTypeId* parameter is present in the request the implementation must try to create

[6] See 4.2 Navigation Services

an object of that type. If the specified object-type isn't valid or not allowed at the given location the request must fail with an appropriate error.

If however the *ObjectTypeId* parameter is omitted from the request, the following default behavior should be adopted:

- If the request contains a file parameter the default object-type is 'Document' and a new document should be created.
- If the request doesn't contain a file parameter the default object-type is 'Folder'.

#### 6.3.1.1 Create Document Objects

Creating new document objects follows the general rules defined for creating objects. In addition the following special cases must be respected:

- If the request contains a file parameter it's value must be used to set the content stream of the new document.
- If the Name parameter is missing and the implementation allows or requires documents to have a Name property, the 'filename' attribute of the file parameter should be used as default.

*Example:*

```
Request to create a new document without setting the content stream.
```

```
POST /myfolder/ HTTP/1.1
Content-Type: application/x-www-form-urlencoded
Content-Length: xxxxxx

Name=myDocument&ObjectTypeId=Document
```

*Example:*

```
Request to create a new document and setting it's content stream. Note that
the ObjectTypeId parameter can be omitted in this case.
```

```
POST /myfolder/ HTTP/1.1
Content-Type: multipart/form-data; boundary=---------------------------
21447684891610979728262467120
Content-Length: xxxxx

---------------------------21447684891610979728262467120
Content-Disposition: form-data; name="cmis:contentstream";
filename="small.gif"
Content-Type: image/gif

GIF89a...................!.......,............s...f.;
---------------------------21447684891610979728262467120
Content-Disposition: form-data; name="Name"
Content-Type: text/plain

myDocument
---------------------------21447684891610979728262467120--
```

*Example:*

```
Request to create a new document and setting it's content stream. Note that
the Name property of the document is retrieved from filename of the uploaded
file
```

```
POST /myfolder/ HTTP/1.1
Content-Type: multipart/form-data; boundary=---------------------------
```

```
21447684891610979728262467120
Content-Length: xxxxx

---------------------------21447684891610979728262467120
Content-Disposition: form-data; name="cmis:contentstream";
filename="small.gif"
Content-Type: image/gif

GIF89a...................!.......,...........s...f.;
---------------------------21447684891610979728262467120--
```

### 6.3.1.2 Create Folder Objects

Creating new folder objects follows the general rules defined for creating objects.

*Example:*

    Request to create a new folder object

```
POST /myfolder HTTP/1.1
Content-Type: application/x-www-form-urlencoded
Content-Length: xxxxxx

Name=child&ObjectTypeId=Folder
```

    Request to create a new folder object. As no file parameter is present the
    ObjectTypeId parameter can be omitted.

```
POST /myfolder HTTP/1.1
Content-Type: application/x-www-form-urlencoded
Content-Length: xxxxxx

Name=child
```

### 6.3.1.3 Create Objects with a Custom Object-Type

Creating CMIS objects with a custom object-type follows the general rules defined for creating objects. In contrast to documents and folder the object-type must explicitly specified.

*Example:*

    Request

```
POST /myfolder HTTP/1.1
Content-Type: application/x-www-form-urlencoded
Content-Length: xxxxxx

Name=child&ObjectTypeId=myObjectType
```

### 6.3.1.4 Create Relationship Objects

TODO

*Example:*

    Request to create a new relationship.

```
TODO
```

## 6.3.2 Delete CMIS Objects

In order to be able to delete CMIS objects with a simple HTML form the reserved *cmis:delete* parameter name is defined. The parameter value identifies the target object that needs to be deleted. Similar to the read operations the target object my either be identified by its *ObjectId* or by hierarchical access.

### 6.3.2.1 Delete Document Objects

Deleting document objects follows the rules defined for CMIS objects in general.

*Example:*

`Request to delete the mydocument child of myfolder.`

```
POST /myfolder HTTP/1.1
Content-Type: application/x-www-form-urlencoded
Content-Length: xxxxxx

cmis:delete=mydocument                         NOTE: url-encoding omitted
```

*Example:*

`Request to delete the mydocument by its ObjectId.`

```
POST /myfolder HTTP/1.1
Content-Type: application/x-www-form-urlencoded
Content-Length: xxxxxx

cmis:delete=[50d9317a-3a95-401a-9638-333a0dbf04bb]
                                               NOTE: url-encoding omitted
```

### 6.3.2.2 Delete Folder Objects

Deleting folder objects follows the rules defined for CMIS objects in general.

*Example:*

`Request to delete myfolder.`

```
POST / HTTP/1.1
Content-Type: application/x-www-form-urlencoded
Content-Length: xxxxxx

cmis:delete=myfolder                           NOTE: url-encoding omitted
```

### 6.3.2.3 Delete Relationship Objects

TODO

*Example:*

`Request to delete a relationship.`

```
TODO
```

## 6.4 Writing Properties

This section deals with setting, updating and removing of properties present with existing CMIS objects. Generally the behavior described below only applies to updatable CMIS properties. Whether a property can be updated or not depends on the corresponding property definition present with the object-type defined for the containing CMIS object.

### 6.4.1 Updating Properties

Modification of an existing CMIS property is achieved by submitting a simple HTML form containing name-value pairs for the properties to be updated. The parameter name either contains the CMIS *id* of the property to be updated or the *name* of the property. In the latter case the CMIS object whose properties (or property) needs to be modified is identified by the request URL.

*Example:*

```
Request:
```

```
POST /myfolder/myobject HTTP/1.1
Content-Type: application/x-www-form-urlencoded
Content-Length: xxxxxx

myStringProperty=anyValue&myIntegerProperty=15
```

### 6.4.2 Adding Properties

CMIS properties that are not required according to their definition, may be missing in the JSON representation of their containing object (see “Value not set” state). Setting it's value (i.e. adding the property) works the same as updating an existing property with the exception that the property wasn't previously present in the JSON representation of the containing object and the user must be aware of the object-type and the set of possible, updatable properties.

As CMIS doesn't support undefined property definitions the type of the resulting property is always defined by the corresponding property definition.

### 6.4.3 Removing Properties

CMIS properties may be "removed" by their value(s) thus changing the property state to "value not set" state. Any subsequent request to the containing CMIS object will result in a JSON object that does not list the removed property within *cmis:properties*. Similarly any attempt to address the removed property directly (e.g. the content stream) will fail with an appropriate HTTP status error.

In order to avoid ambiguity between change the property value to be an empty string and changing the status to "value not set" the approach used upon objected deletion is adopted: the reserved *cmis:delete* parameter name is used, followed by propertyName.

*Example:*

```
Request to remove the myStringProperty and at the same time update the
myIntegerProperty.
```

```
POST /myfolder/myobject HTTP/1.1
Content-Type: application/x-www-form-urlencoded
Content-Length: xxxxxx

cmis:delete=myStringProperty&myIntegerProperty=26
                                                  NOTE: url-encoding omitted
```

Note: If both a child object and a property exist with the same name, the *cmis:delete* parameter will always affect the property. TODO: how to remove the child object in this case. By ObjectId?

### 6.4.4 Handling Special Properties

#### 6.4.4.1 Setting Content Streams

While the content stream of a CMIS document is exposed through various properties defined for the document, the content stream manipulation is covered by separate methods in the document model. For the browser-binding however, it makes sense to cover content stream modification along with the general property handling. Note however, that the content type of the POST request should be adjusted accordingly.

*Example:*

```
Request: Setting the content stream of an document object and update another
custom property.
```

```
POST /myfolder/mydocument HTTP/1.1
Content-Type: multipart/form-data; boundary=---------------------------
21447684891610979728262467120
Content-Length: xxxxx

---------------------------21447684891610979728262467120
Content-Disposition: form-data; name="cmis:contentstream";
filename="small.gif"
Content-Type: image/gif

GIF89a...................!.......,...........s...f.;
---------------------------21447684891610979728262467120
Content-Disposition: form-data; name="customProperty"
Content-Type: text/plain

some other document property
---------------------------21447684891610979728262467120--
                                                NOTE: url-encoding omitted
```

#### 6.4.4.2 Deleting Content Streams

Deleting the content stream of a document object follows the same rules as defined for regular properties. Note that requesting the deletion of the reserved cmis:contentstream pseudoproperty must have the same effect as *deleteContentStream*. This particularly applies to all content stream related properties defined with the document object-type.

*Example:*

```
Request: Deleting the content stream of an document object
```

```
POST /myfolder/mydocument HTTP/1.1
Content-Type: application/x-www-form-urlencoded
Content-Length: xxxxxx

cmis:delete=cmis:contentstream                  NOTE: url-encoding omitted
```

## 6.5 Batched Writing

In order to be able to cover multiple and complex write operations with a single HTTP request, batched writing should be supported by the implementation.

The main difference to the simple writing consists of the definition of a *cmis:diff* parameter instead of multiple name-value pairs. The *cmis:diff* itself lists the set of operations to be executed in the parameter value. Similarly to the simple writing operations the batched writing is achieved by POST requests.

### 6.5.1 Diff Format

The *cmis:diff* parameter is defined to consist of one or multiple JSON like key-value pair(s). The JSON key must start with a character that identifies the desired operation followed by the identification of the target object. The operation characters are any of "+", "^", "-" or ">" representing the CMIS operations as follows:

- "+" : *createObject, createFolder, createDocument, createPolicy, addObjectToFolder*
- "^" : *updateProperties*, *setContentStream*, *deleteContentStream*
- "-" : *deleteObject*, *deleteTree*, *removeObjectFromFolder*
- ">" : *moveObject*
- *TODO createRelationship ?*
- *TODO deleteObject for Relationship?*

The key must be separated from the value by a ":" surrounded by whitespace and the command must be completed by line ending. The nature of the value itself depends on the operation (see below).

```
diff        ::= members
members     ::= pair | pairs
pair        ::= key " : " value
pairs       ::= pair line-end pair | pair line-end pairs
line-end    ::= "\r\n" | "\n" | "\r"
key         ::= opchar id
opchar      ::= "+" | "^" | "-" | ">"
id          ::= * identification of the object either by the ObjectId itself
                  or through hierarchical access, where the request URL
                  either identifies the target object itself or any of its
                  ancestors *
value       ::= value+ | value- | value^ | value>
value+      ::= * a JSON object *
value-      ::= ""
value^      ::= * any JSON value except JSON object *
value>      ::= id
```

#### 6.5.1.1 Object Identification within the cmis:diff Parameter

The *id*s present as keys part in the *cmis:diff* parameter are used to identify the CMIS object (or property that is affected by the desired operation.

The *id* either represents the *ObjectId* of the object itself or – alternatively is any other string that can be resolved to the target object. In the latter case the object is obtained through hierarchical access (see Hierarchical Access) where the objected addressed by the request URL is used as starting point.

*Example:*

```
Request: diff with object identification by ObjectID
```

```
POST /myfolder/mydocument HTTP/1.1
Content-Type: application/x-www-form-urlencoded
Content-Length: xxxxxx

cmis:diff=-50d9317a-3a95-401a-9638-333a0dbf04bb :
                                              NOTE: url-encoding omitted
```

*Example:*

```
Request: diff with hierarchical object identification. 'mydocument' is child
of 'myfolder'.
```

```
POST /myfolder HTTP/1.1
Content-Type: application/x-www-form-urlencoded
Content-Length: xxxxxx

cmis:diff=-mydocument :                               NOTE: url-encoding omitted
```

```
Request: diff with hierarchical object identification. 'myFolder' is the
requested resouce as well as the target of the operation.
```

```
POST /myfolder HTTP/1.1
Content-Type: application/x-www-form-urlencoded
Content-Length: xxxxxx

cmis:diff=-.                                          NOTE: url-encoding omitted
```

### 6.5.2 Create Objects

Object creation is triggered by the '+' operation character at the beginning of a new line. If the client wishes to explicitly indicate the desired object-type, the corresponding property must be present in the diff. Otherwise some implementation specific default should be used.

The value of the diff entry consists of a JSON object whose format corresponds to the serialization of CMIS objects upon read access. This includes the possibility to not only create a single child object but to import a complete object tree including their properties and relationships.

In order to form valid JSON keys and values, all CMIS identifiers, properties and attributes and their corresponding values must be properly escaped. See http://www.ietf.org/rfc/rfc4627.txt.

*Example:*

```
Request to create a new document.
```

```
POST /myfolder HTTP/1.1
Content-Type: application/x-www-form-urlencoded
Content-Length: xxxxxx

cmis:diff=+mydocument : { "cmis:properties" :  { "ObjectTypId" :
"myObjectType", "myProperty" : 24, … } }
                                NOTE: url-encoding omitted for readability
```

```
Request to import an object tree.
```

```
POST /myfolder HTTP/1.1
Content-Type: application/x-www-form-urlencoded
Content-Length: xxxxxx

cmis:diff=+child : {
         "cmis:properties" : { "ObjectTypId" : "Folder", … },
         "grandchild" :
           {
           "cmis:properties" : { "ObjectTypId" : "Folder", … },
           "mydocument :
             {
             "cmis:properties" : { "ObjectTypeId : "myObjectType",
                                   "myProperty" : 24, … }
             }
           }
```

```
        }                              NOTE: url-encoding omitted for readability
```

### 6.5.3 Delete Objects

The '-' operation character is used to delete the CMIS object identified by the following id. Note however that the delete operation line must not specify any value.

*Example:*

Request to delete the document 'mydocument'.

```
POST /myfolder HTTP/1.1
Content-Type: application/x-www-form-urlencoded
Content-Length: xxxxxx

cmis:diff=-mydocument :            NOTE: url-encoding omitted for readability
```

Alternative solution to delete the same document:

```
POST /myfolder/mydocument HTTP/1.1
Content-Type: application/x-www-form-urlencoded
Content-Length: xxxxxx

cmis:diff=-. :                                    NOTE: url-encoding omitted
```

… or by *ObjectId*:

```
POST /[b410b572-9363-4f9c-9f70-04593652179b] HTTP/1.1
Content-Type: application/x-www-form-urlencoded
Content-Length: xxxxxx

cmis:diff=-. :                                    NOTE: url-encoding omitted
```

### 6.5.4 Move Objects

Moving objects is achieved by starting the line with the move operation character ('>'), followed by the identification of the object to be moved. The parameter value contains the identification of the target. In contrast to the domain model description this may not only be the *ObjectId* of the target parent folder. TODO review again

*Example:*

Request to move a document.

```
POST /myfolder HTTP/1.1
Content-Type: application/x-www-form-urlencoded
Content-Length: xxxxxx

cmis:diff=>mydocument : /otherfolder/mydocument
                                                  NOTE: url-encoding omitted
```

### 6.5.5 Create and Delete Relationship Objects

TODO

*Example:*

    Request to create a new relationship.

```
TODO
```

*Example:*

    Request to delete a relationship.

```
TODO
```

## 6.5.6 Write Properies

### 6.5.6.1 Update Properties

The '^' character at the beginning of a new line in the diff parameter indicates that the subsequent id refers to a property that needs to modified or created. The value following the :-separator must specifiy the desired property value. The format of the value corresponds to the property serialization upon read-access (see Property Value). This also includes the special treatment of the various property types (see Property Types).

*Example:*

    Request to update a property.

```
POST /myfolder/mydocument HTTP/1.1
Content-Type: application/x-www-form-urlencoded
Content-Length: xxxxxx

cmis:diff=^customProperty : "the property value"
                                              NOTE: url-encoding omitted
```

*Example:*

    Request to update a multi-valued numeric property.

```
POST /myfolder/mydocument HTTP/1.1
Content-Type: application/x-www-form-urlencoded
Content-Length: xxxxxx

cmis:diff=^customProperty : [ 250975, 14, 109873 ]
                                              NOTE: url-encoding omitted
```

### 6.5.6.2 Change Property State to "value not set"

In order to change the state of a given non-required property to "value not set" the same diff entry is created as if the property was updated. However, the value part is omitted. TODO review again. Eventually change to use the '-' opchar in order to avoid ambiguity between empty-string an no-value. See also the simple content editing...

*Example:*

    Request to change the state of a property to "value not set".

```
POST /myfolder/mydocument HTTP/1.1
Content-Type: application/x-www-form-urlencoded
Content-Length: xxxxxx

cmis:diff=^customProperty :                   NOTE: url-encoding omitted
```

#### 6.5.6.3 Update Binary Content

Please note the special handling of setting the binary content stream of a CMIS document: The diff treats the content stream as a regular property. However, due to it's binary nature however the JSON format it not suite for the data serialization and the request has the following characteristics:

- the Content-Type of the request is multipart/form-data,
- as CMIS doesn't define an separate identifier for the content stream, *cmis:contentstream* is used as placeholder.
- the value of the *cmis:diff* entry is expected to be missing
- a separate multipart parameter is added to request. It's name corresponds to the name of the *cmis::diff* entry, while the parameter value contains the binary data.
- Note: Upon modification of the content stream of a CMIS document the related properties must be adjust as well. This includes *ContentStreamLength*, *ContentStreamMimeType*, *ContentStreamFilename*.

*Example:*

```
Request to set the content stream of a document.
```

```
POST /myfolder/mydocument HTTP/1.1
Content-Type: multipart/form-data; boundary=---------------------------
21447684891610979728262467120
Content-Length: xxxxx

---------------------------21447684891610979728262467120
Content-Disposition: form-data; name="cmis:contentstream"
Content-Type: image/gif

GIF89a...................!.......,............s...f.;
---------------------------21447684891610979728262467120
Content-Disposition: form-data; name="cmis:diff"
Content-Type: text/plain

^cmis:contentstream :
---------------------------21447684891610979728262467120--
```

# 7 Search

TODO

# 8 Appendix

## 8.1 Reserved Names and "Selectors"

### 8.1.1 Reserved Names

#### 8.1.1.1 cmis:properties

TODO

#### 8.1.1.2 cmis: relationships

TODO

#### 8.1.1.3 cmis:object-types

TODO

#### 8.1.1.4 cmis:unfiled

TODO

#### 8.1.1.5 cmis:delete

TODO

#### 8.1.1.6 cmis:diff

TODO

#### 8.1.1.7 cmis:contentstream

TODO

### 8.1.2 "Selectors"

#### 8.1.2.1 query

TODO

#### 8.1.2.2 depth

TODO

#### 8.1.2.3 none

TODO