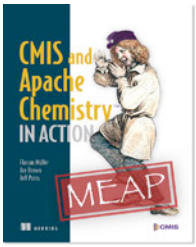


The Browser Binding with a CMIS Repository

By Florian Müller, Jay Brown, and Jeff Potts, authors of *CMIS and Apache Chemistry in Action*

A big part of the CMIS specification describes how the CMIS domain model is mapped to the bytes that are transferred. These mappings are called bindings. CMIS 1.0 defines two bindings, the Web Services binding and the AtomPub binding; and CMIS 1.1 adds a third, the Browser binding. In this article, based on chapter 11 of CMIS and Apache Chemistry in Action*, the authors discuss Browser binding.*

The objective of the Browser binding is to enable a JavaScript application in a web browser to access data in a CMIS repository. The binding only makes use of features that are available in the HTML and JavaScript specifications. It only uses the HTTP methods `GET` and `POST`: `GET` for requests that read data, and `POST` for requests that create, modify, or delete data. CMIS repositories return JSON responses as well as binary contents of documents. CMIS clients use URL parameters and HTML form data to communicate with the repository. Multipart messages are used to transport content from the client to the repository. A simple HTML form is sufficient to create a document in a CMIS repository.

The Browser binding covers the entire specification. There are no restrictions as in AtomPub, so in that respect the feature set is comparable with the Web Services binding. This includes the error handling. The Browser binding uses the same HTTP status codes as AtomPub but additionally sends a JSON response that contains the exception type and a message. HTTP status codes can be tricky in a browser application, but adding the parameter `suppressResponseCodes` with the value `true` to a URL can turn them off. The repository will then always return the HTTP status code 200.

The service document

In contrast to those in the AtomPub binding, the URLs of this binding are entirely predictable. That is, the specification defines URL patterns that work for all repositories. Similar to the AtomPub binding, the application must first get the service document. The service document contains information about all repositories available at this endpoint, two base URLs per repository. One URL is called the *repository URL* and the other one is called the *root folder URL*. The repository URL is used for all requests that are independent of the folder hierarchy, such as accessing or changing type definitions, performing a query, or getting content changes. A URL that is derived from the root folder URL always addresses an object, either by its ID or its path. The term *root folder URL* is a bit misleading because unfiled objects can also be addressed with this URL, so don't be confused. To select an object by path, the object's path is attached to the root folder URL.

To select an object by ID, the URL parameter `objectId` with the object's ID is attached to the root folder URL. If the parameter `objectId` is set, it takes precedence over the path.

Let's assume the root folder URL is http://example.com/repository/root. To select the document with the object ID 123456 and the path /myfolder/doc.txt, the following three URLs would work:

This one is by ID:

```
http://example.com/repository/root?objectId=123456
```

This one is by path:

```
http://example.com/repository/root/myfolder/doc.txt
```

In this one, the `objectId` parameter wins over the path:

```
http://example.com/repository/root/another/path?objectId=123456
```

Such URLs are called *object URLs*. If an object URL points to a document, the content of this document is returned by default. If it points to a folder, the children of this folder are the default return type. For all other base types, the object details are returned.

A client can specify which aspect of the object the repository should return by setting the parameter `cmisselector`. For example, a URL that gets the object details of a document could look like this:

```
http://example.com/repository/root?objectId=123456&cmisselector=object
```

And a URL to get the versions of a document could look like this:

```
http://example.com/repository/root/myfolder/doc.txt
          ?cmisselector=versions
```

Other URL parameters are similar to the parameters in the AtomPub URL templates. Property filters can be defined; allowable actions, ACLs, and policies can be turned on and off; a rendition filter can be set; and so on. For operations that return lists, such as `getChildren`, the offset, the length, the order of the list, and other things can be defined. The official CMIS 1.1 specification is the best complete reference for all the supported parameters and operations.

The succinct feature

A feature that is unique to the Browser binding is the `succinct` flag. All bindings transport for each property the property ID, the data type, the query name, the display name, the local name, and the value. That makes it easier for clients that don't know the type definition of the object to work with these properties. But if the client knows the type definition, this is extra weight. The `succinct` parameter with the value `true` can be attached to all Browser binding URLs that return objects. The repository then only sends the property ID and the value, which makes the response more compact.

Try capturing a URL from the CMIS Workbench, and open in it a web browser. The Workbench always sets the `succinct` flag. Remove that flag from the URL, and reload the URL in the web browser. You'll see that the response is now considerably larger.

CRUD operations

Operations that create, update, or delete objects use HTTP `POST` instead of HTTP `GET`. Data is transmitted in the same format that web browsers use to send HTML form data to a server. The form data must be URL encoded or sent as a multipart message. If content is attached to the request, then it *must* be a multipart request. (Thus, that only applies to the operations `createDocument`,`setContentStream`,`appendContentStream`, and `checkIn`.)

To indicate which operation should be invoked, the parameter `cmisaction` must be included. If a new document should be created, the value of `cmisaction` must be `createDocument`. To delete an object, `cmisaction` must be `delete`, and so on. The CMIS specification defines a `cmisaction` value for each operation as well as all other required and optional parameters.

Complex data structures are broken down to multiple parameters and parameter names with indexes. For example, to transmit the two base properties for creating a document (`cmis:name` and `cmis:objectTypeId`), the four properties shown in table 1 are required.

Table 1 Example parameters and values for **`createDocument`**

Parameter name	Parameter value
`propertyId[0]`	`cmis:name`
`propertyValue[0]`	`myNewDocument.txt`

`propertyId[1]`	`cmis:objectTypeId`
`propertyValue[1]`	`cmis:document`

You need a parameter pair (`propertyId` and `propertyValue`) to set one single-value property. The indexes indicate which parameters belong together, but the order doesn't matter. This pattern is used for all complex data structures and lists such as ACLs or lists of object IDs or change tokens.

> **TRANSPORTING PARAMETER NAMES AND VALUES** At first glance, it looks unnecessarily complex to split a property into two parameters. Wouldn't it be simpler to use the property ID as the parameter name?
>
> It would, but that could lead to ambiguities. Here's the reason why. The HTML specification says parameter names are case insensitive but property IDs are case sensitive. If a repository provided a type with two properties that differed only in the capitalization of the property ID, the client (such as a web browser) would normalize the parameter names, and it wouldn't be possible to identify which property the application meant. The chance of such a situation occurring isn't very high, but to prevent any kind of ambiguity, the Technical Committee fixed all parameter names.
>
> Apart from that, parameter names in multipart messages should only use 7bit ASCII characters because they're used in HTTP headers. Property IDs with characters outside this charset are very likely, so it's better to avoid compatibility issues and use fixed parameter names instead.

As with the other two bindings, you can add JSON structures that aren't defined in the CMIS specification. Clients that don't understand these should ignore them. The names of these extensions should be chosen carefully, though. In the other two bindings, the XML namespace can distinguish an extension tag from a CMIS or an Atom tag. Because JSON has no namespaces, the names should be as unique as possible. Future versions of CMIS may introduce more elements, and a name clash could lead to incompatibilities.

To learn more about the Browser binding and how to use it, see appendix D of *CMIS and Apache Chemistry in Action*. It shows how to build a JavaScript application that accesses a CMIS repository. It also demonstrates how to use JSON-P and callbacks to work with a repository that's hosted on a different server. Chapter 12 of *CMIS and Apache Chemistry in Action* covers user authentication and CSRF attack protection.

JSON create request examples

This section shows two examples of JSON requests. The first is an extremely simple example of `createFolder`:

```
cmisaction=createFolder&
  propertyId[0]=cmis%3AobjectTypeId&
  propertyValue[0]=cmis%3Afolder&
  propertyId[1]=cmis%3Aname&
  propertyValue[1]=myFolder& succinct=true
```

The second is a more complicated example of a `createDocument` multipart message:

```
--aPacHeCheMIStryoPEncmiS6a5a1a37createDocument13b766ab8a531763c9a
Content-Disposition: form-data; name="cmisaction"
Content-Type: text/plain; charset=utf-8

createDocument
--aPacHeCheMIStryoPEncmiS6a5a1a37createDocument13b766ab8a531763c9a
Content-Disposition: form-data; name="propertyId[0]"
Content-Type: text/plain; charset=utf-8

cmis:objectTypeId
--aPacHeCheMIStryoPEncmiS6a5a1a37createDocument13b766ab8a531763c9a
Content-Disposition: form-data; name="propertyValue[0]"
Content-Type: text/plain; charset=utf-8

cmis:document
--aPacHeCheMIStryoPEncmiS6a5a1a37createDocument13b766ab8a531763c9a
Content-Disposition: form-data; name="propertyId[1]"
Content-Type: text/plain; charset=utf-8
```

```
cmis:name
--aPacHeCheMIStryoPEncmiS6a5a1a37createDocument13b766ab8a531763c9a
Content-Disposition: form-data; name="propertyValue[1]"
Content-Type: text/plain; charset=utf-8

myDoc.txt


--aPacHeCheMIStryoPEncmiS6a5a1a37createDocument13b766ab8a531763c9a
Content-Disposition: form-data; name="versioningState"
Content-Type: text/plain; charset=utf-8

none
--aPacHeCheMIStryoPEncmiS6a5a1a37createDocument13b766ab8a531763c9a
Content-Disposition: form-data; name="content"; filename=myDoc.txt
Content-Type: text/plain
Content-Transfer-Encoding: binary

Hello World!
--aPacHeCheMIStryoPEncmiS6a5a1a37createDocument13b766ab8a531763c9a--
```

JSON object response example

Listing 1 shows a typical example of a response for requested CMIS object details via the Browser binding, with the `succinct` flag set to `true`.

Listing 1 JSON object response

```
{
    "allowableActions":{  #A
        "canGetACL":false,
        "canGetObjectRelationships":false,
        "canGetContentStream":true,
        "canCheckIn":false,
        "canApplyACL":false,
        "canRemoveObjectFromFolder":true,
        "canMoveObject":true,
        "canDeleteContentStream":true,
        "canGetProperties":true,
        "canGetAllVersions":true,
        "canApplyPolicy":false,
        "canGetObjectParents":true,
        "canSetContentStream":false,
        "canCreateRelationship":false,
        "canGetFolderTree":false,
        "canCheckOut":true,
        "canCreateDocument":false,
        "canCancelCheckOut":false,
        "canAddObjectToFolder":true,
        "canRemovePolicy":false,
        "canDeleteObject":true,
        "canGetDescendants":false,
        "canGetFolderParent":false,
        "canGetAppliedPolicies":false,
        "canDeleteTree":false,
        "canUpdateProperties":true,
        "canGetRenditions":false,
        "canCreateFolder":false,
        "canGetChildren":false
    },
    "acl":{ #B
        "aces":[
            {
                "isDirect":true,
                "principal":{
                    "principalId":"anyone"
                },
                "permissions":[
                    "cmis:all"
                ]
```

```
                }
            ]
        },
        "exactACL":true,
        "succinctProperties":{  #C
            "cmis:isLatestMajorVersion":true,
            "cmis:contentStreamLength":395,
            "cmis:contentStreamId":null,
            "cmis:objectTypeId":"cmisbook:text",
            "cmis:versionSeriesCheckedOutBy":null,
            "cmis:versionSeriesCheckedOutId":null,
            "cmisbook:author":null,
            "cmis:name":"welcome.txt",
            "cmis:contentStreamMimeType":"text/plain",
            "cmis:versionSeriesId":"162",
            "cmis:creationDate":1353180064169,
            "cmis:changeToken":"1353180064169",
            "cmis:isLatestVersion":true,
            "cmis:versionLabel":"V 1.0",
            "cmis:isVersionSeriesCheckedOut":false,
            "cmis:lastModifiedBy":"system",
            "cmis:createdBy":"system",
            "cmis:checkinComment":null,
            "cmis:objectId":"161",
            "cmis:isImmutable":false,
            "cmis:isMajorVersion":true,
            "cmis:baseTypeId":"cmis:document",
            "cmis:lastModificationDate":1353180064169,
            "cmis:contentStreamFileName":"welcome.txt"
        }
    }
```

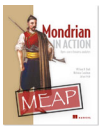
#A JSON responses: much smaller and easier to read than other XML responses
#B Passes ACLs
#C Properties: notice how concise they are

Summary

Knowing about the CMIS bindings will help you debug, tune, deploy, and, sometimes, develop CMIS applications. The chapter also described how to capture CMIS wire traffic, which is useful for learning about the bindings and for debugging your application.

Here are some other Manning titles you might be interested in:

Making Sense of NoSQL
Dan McCreary and Ann Kelly



Mondrian in Action
William D. Back, Nicholas Goodman, and Julian Hyde

Solr in Action
Trey Grainger and Timothy Potter

Last updated: July 3, 2013