A CMIS API library for Python, Part 1: Introducing cmislib

A client-side Content Management Interoperability Services API

Jeff Potts

25 March 2010
(First published 23 March 2010)

This is the first of a two-part series which will introduce you to cmislib, a client-side library for working with CMIS content libraries. Content Management Interoperability Services (CMIS) is a specification that provides a standard way to access content, regardless of the underlying repository implementation or the choice of the front-end programming language. In this article, learn about the cmislib API for Python using examples.

View more content in this series

25 Mar 2010 - Added links to Part 2 of the series in Introducing cmislib: A client-side CMIS API for Python, Conclusion, and Resources.

Introducing cmislib: A client-side CMIS API for Python

It's shaping up to be a busy spring for the Content Management Interoperability Services (CMIS) specification. OASIS is readying the spec for its 1.0 release, repository vendors are working hard to get their server-side implementations ready, and developers in the content management community are releasing clients and APIs that make it easier to explore and work with rich content repositories in a standard way.

Frequently used acronyms

- ACL: Access control list
- API: Application program interface
- HTTP: Hypertext Transfer Protocol
- OASIS: Organization for the Advancement of Structured Information Standards
- REST: Representational State Transfer
- SDK: Software development kit
- SQL: Structured Query Language
- URL: Uniform Resource Locator
- WSDL: Web Services Description Language

- XML: Extensible Markup Language

If you've ever built a content-centric application, you know that, typically, the first hurdle is figuring out how to talk to the underlying content repository. Your team first gets a crash course in the repository's SDK. Then you design the application including the integration between the presentation tier and the content services tier. Finally, you execute against the plan and celebrate your success at happy hour over cheese fries. The shame of it, though, other than the impact those happy hour carbs are having on your waist line, is that the process repeats itself for each new combination of front-end and back-end because every repository has its own unique API. And if your application talks to multiple repositories, as is often the case, you've got to learn and code to multiple interfaces.

Fortunately, this problem has been solved before. The pattern is the same that existed prior to the standardization of SQL. Relational databases created by IBM® and others began appearing in the early 1970s, but the first formal standards effort around the SQL query language didn't happen until 1986. Once that happened, especially after a major revision in 1992, developers could create front-end applications and be reasonably assured it could work against multiple relational back-ends. CMIS has the potential to do the same thing for content-centric applications that SQL did for relational database applications. It provides for a standard way of interacting with the back-end, regardless of the underlying repository implementation or the choice of front-end programming language. This time, instead of rows and columns, we're talking about unstructured and semi-structured content—usually files of some sort—that typically live in a hierarchical folder structure.

Figure 1. CMIS provides a common interface regardless of the front- or back-end containing an image

This article is about a client-side library for working with CMIS repositories from Python called *cmislib*. Now managed as part of the Apache Chemistry project, the goal of the library is to make it easy for Python developers to write content-centric applications that can work with any CMIS-compliant back-end. For many, the API can be an easy way to understand the power of CMIS first-hand.

Other articles in this series

- Part 2: Build real world ECM tools with Python and cmislib

Let's look at why the API was created, what it does for you, how it was developed (in case you want to write one in your favorite language, hint, hint), and a few brief examples of cmislib in action. The next article in the series will walk you through a real-world application of the library.

Motivation for creating the API

For a lot of reasons, the creation of a client-side API for CMIS in Python seemed like a good idea. Some of them are strategic and idealistic. Others are a little more tactical and selfish. Let's start with reasons filed under "Greater Good".

CMIS-compliant providers must offer both a Web services binding and a RESTful Atom Publishing Protocol (AtomPub) binding. Each binding has certain advantages over the other, but one difference is in how CMIS services are discovered and invoked across different servers. The Web services binding includes a WSDL file that can be used to automatically generate client code. With little more than your CMIS server's WSDL you can generate your own client-side API if you are willing to use the Web services binding.

The RESTful AtomPub binding, on the other hand, lacks a standard way to describe its services. Being restful, all of its services are accessed through URL, but the CMIS specification leaves the specific URLs up to each vendor. So if you want to write code that will work across all CMIS compliant providers using the RESTful AtomPub binding, you have a little more work to do on the client side. Instead of repeating that work across projects, it makes sense to have an open source project do it once for everyone.

The next reason is about adoption by both application developers and content repository vendors. Developers can watch webinars and blog posts and Twitter traffic, but until they get their hands on it and see the value first-hand, CMIS is just another over-hyped buzzword. If software still came in boxes, you could imagine the brightly-colored "explosion" graphic loudly proclaiming, "Now with CMIS!". Python—clean, productive, easy-to-install—seems like a great choice for those looking to get beyond the hype to see what CMIS can actually do for them and their applications. The cmislib API shields developers from implementation details and provides an intuitive, object-oriented, interoperable way of working. Hopefully, developers will take it for a spin, like what they see, and begin to use CMIS as the standard way for custom content-centric applications to interoperate with rich content repositories, whether that client is built in Python or some other language.

If only one or two vendors adopt CMIS, the whole purpose will be defeated. So it definitely makes sense for those who benefit from a standard to do what you can to push vendor adoption. The cmislib distribution includes unit tests simply as a development best practice. The test suite is great for helping to ensure that functionality doesn't get regressed as the API is developed and is also a handy way to validate interoperability in a repeatable way. What is really cool, though, is that the unit tests function as a test suite for vendors. IBM, Alfresco™, OpenText, and Nuxeo have all taken advantage of cmislib to uncover issues with their implementations. This hasn't been limited to cmislib—vendors have used all sorts of CMIS tools and clients built by the community to validate their work and that's a very good thing.

All-for-one-and-one-for-all is great motivation, but it rarely gets a line of code written. Every open source project starts with an itch a developer needed to scratch. In this case, the itch started with an intranet project that Optaros™ did for a client to provide an integration between Django®, a Python-based Web application development framework, and Alfresco, an open source content management platform. Developed when CMIS was barely an idea at OASIS, the integration relies

on Alfresco Web Scripts (a framework for rolling your own RESTful API to Alfresco) on the server side to move XML back-and-forth over HTTP between the Django-based presentation tier and the Alfresco repository. It works great, but it's Alfresco-specific. Refactoring the Django integration to leverage CMIS seemed like a good idea. But rather than make it Django-specific, we chose to first roll out cmislib as a lower-level Python API. The benefit is that in addition to Django, other Python projects, like Zope® and Plone®, as well as custom Python applications, can more easily integrate with CMIS repositories by leveraging cmislib.

The last selfish reason is about developer productivity. Most enterprises don't have the luxury of dealing with only a single repository. And solutions often either can't anticipate the specific repository that will be in play or at least need the option to switch at some point down the road. The CMIS standard obviously helps with those problems, but to actually get work done productively, client-side libraries are needed. Other projects are already underway to provide Java™- and PHP-based client libraries for CMIS. But Python is also very prevalent on the presentation tier, so a Python-based client library for CMIS is important.

What the API does

The goal of cmislib is to abstract away the underlying implementation details of CMIS. Developers don't want or need to learn how CMIS works in order to build solutions on top of CMIS repositories. Instead, cmislib provides an easy-to-grok object model that will be instantly familiar to anyone who works with a content repository or reads the CMIS spec. Instead of collections, and entries, and feeds, developers work with natural content management concepts like Repositories, Folders, Documents, and ACLs.

As mentioned, cmislib uses the RESTful AtomPub binding to communicate with CMIS servers. An important development consideration was to make sure that cmislib has no vendor-specific knowledge about the back-end repositories it is hitting—the library treats a CMIS provider like a black box. When you use cmislib to connect to a CMIS provider, you give it the CMIS provider's entry point, or, service URL, and some credentials. The library figures out how to further interact with the server by interrogating the server responses.

For example, suppose you want to get a list of documents that are currently checked out. The CMIS specification tells you that:

- A collection of checked out documents exists.
- `repositoryId` is a required argument when you invoke the `getCheckedOutDocs` service.
- Several optional arguments can be passed in, mostly having to do with paging the result set.
- The response from the service will be an Atom feed.

What the specification does not tell you, however, is the exact URL to retrieve the collection. That's left up to the vendor. One of the things cmislib handles for you is figuring out what that URL is and how to convert the response into Python objects you can work with in a Pythonic way. Listing 1 shows how this interaction looks from the Python shell:

Listing 1. Listing the checked out documents in a repository

```
>>> rs = repo.getCheckedOutDocs()

>>> len(rs)
2
>>> for doc in rs:
...     doc.title
...
u'Response XML (Working Copy)'
u'd1 (Working Copy)'
```

Similarly, suppose you want to do a checkout on a document. The specification tells you that to perform a checkout you need to post an Atom entry to the checkedout collection and the repository will return an Atom entry representing the Private Working Copy (PWC) of the object you just checked out. If you use cmislib, you don't have to worry about determining the collection, building and posting the Atom entry XML, or handling the XML response. Instead, you just call the `checkout` method on the object and cmislib gives you back a Document object representing the PWC. Listing 2 shows this interaction:

Listing 2. Checking out a document

```
>>> doc.title
u'testdoc'
>>> pwc = doc.checkout()
>>> pwc.title
u'testdoc (Working Copy)'
>>> doc.checkedOut
True
```

Development and testing approach

It was important that cmislib be written as close to the specification as possible. So the first step was to stub out the classes and methods with Eclipse and the specification open side-by-side. I added cross-references to the specification to in-line comments so I quickly found the applicable spot in the specification later when I came back to implement the method.

I set up my build process, documentation, and source code control right from the start. It was important to get those established early so additional developers could join the project and get ramped up quickly.

The code evolved iteratively. Each iteration started with writing unit tests for the new functionality and continued with actually coding the methods until the unit tests passed. I started with the basics like querying the repository for its capabilities and retrieving objects and object properties to validate the general approach. From there I moved on to full write operations, checkout/checkin, relationships, and ACLs.

Testing was a little tough initially because no reference implementation existed (the AtomPub binding in Apache Chemistry's reference implementation was read-only at the time) and because vendors are still working on their implementations. Alfresco, an early adopter of CMIS, had the most mature implementation, so I started there. As soon as most of the unit tests were finishing

cleanly against Alfresco, I began to test against additional vendors who had publicly available CMIS implementations. IBM graciously volunteered to make their implementation available. Adding that second implementation was an eye-opener, but a great exercise for cmislib and all of the vendors involved. We found issues on both the client- and server-side that might not have otherwise been discovered as quickly without this kind of interoperability testing.

If you are developing CMIS tools or APIs like cmislib, it is critical that you test against as many different servers as you possibly can. The specification is brand new and vendor implementations are still maturing, even from those that claim full compliance to the draft specification. The most common issues uncovered fell into three buckets:

- **Incomplete implementations**. CMIS is still very new. It's common to find missing services (ACL, relationships, policies, and change logs seem to be the least-supported at the moment), mandatory features that aren't yet supported (for example, mandatory collections and links that aren't provided), and hardcoded values.
- **Differing interpretations of the specification**. The CMIS specification is a well-written and easy-to-read document, but some things are still left open to interpretation. For example, prior to draft 6, the contents of the checkedout collection were vague. Was it supposed to contain Private Working Copies (PWCs) of the checked out objects or the objects themselves? Different vendors interpreted it differently and implemented according to their interpretation. That specific issue has since been cleaned up (PWCs is the answer, if you are curious) but you can see how that might make writing an interoperable client difficult.
- **Bad assumptions**. Sometimes a vendor's specific extension to the specification is obvious and sometimes it isn't. If you code your API against one server and treat it as if it is the reference implementation, you've made an assumption that other implementations are going to operate the same way. The challenge right now is that there isn't a CMIS reference implementation with an AtomPub binding that is fully functionally complete and 100% compliant to the specification.

A few examples

The next article in this series will show a real application of the library by walking through a Python script that you can use to bulk load a CMIS repository with digital assets and metadata. The basic examples below are taken from the cmislib documentation and show how to perform common operations with the API from the Python interactive shell. The operations include getting information about the repository, working with Folders and Documents, and finding objects by CMIS query, path, object ID, or relationship.

Getting a repository object

The CmisClient and Repository objects are common starting places for just about everything you'll want to do with the CMIS repository. Getting an instance is simple—all you need to know is the service URL and credentials of the repository. Here's how:

1. From the command-line, start the Python shell by typing python then click enter.
2. Import the CmisClient:

```
>>> from cmislib.model import CmisClient
```

3. Point the CmisClient at the repository's service URL:

```
>>> client = CmisClient('http://cmis.alfresco.com/s/cmis', 'admin', 'admin')
```

4. Repositories have an ID—if you know it, you can get the repository by ID as well. In this case, we'll ask the client for the default repository.

```
>>> repo = client.defaultRepository
>>> repo.id u'83beb297-a6fa-4ac5-844b-98c871c0eea9'
```

5. Now you can get the properties of the repository. This for-loop spits out everything cmislib knows about the repository. (I truncated the listing for the sake of brevity).

```
>>> repo.name u'Main Repository'
>>> info = repo.info
>>> for k,v in info.items():
    ...     print "%s:%s" % (k,v)
    ...
    cmisSpecificationTitle:Version 1.0 Committee Draft 04
    cmisVersionSupported:1.0
    repositoryDescription:None
    productVersion:3.2.0 (r2 2440)
    rootFolderId:workspace://SpacesStore/aa1ecedf-9551-49c5-831a-0502bb43f348
    repositoryId:83beb297-a6fa-4ac5-844b-98c871c0eea9
    repositoryName:Main Repository
    vendorName:Alfresco
    productName:Alfresco Repository (Community)
```

Working with Folders and Documents

Once you have the Repository object you can start to work with objects in the repository like Folders and Documents.

1. Create a new folder in the root. If you are actually following along, name your folder something unique if you test against a public repository.

```
>>> root = repo.rootFolder
>>> someFolder = root.createFolder('someFolder')
>>> someFolder.id
u'workspace://SpacesStore/91f344ef-84e7-43d8-b379-959c0be7e8fc'
```

2. Then, you can create some content:

```
>>> someFile = open('test.txt', 'r')
>>> someDoc = someFolder.createDocument('Test Document', contentFile=someFile)
```

3. And, if you want, you can dump the properties of the newly-created document (this is a partial listing):

```
>>> props = someDoc.properties
>>> for k,v in props.items():
...     print '%s:%s' % (k,v)
...
cmis:contentStreamMimeType:text/plain
cmis:creationDate:2009-12-18T10:59:26.667-06:00
cmis:baseTypeId:cmis:document
cmis:isLatestMajorVersion:false
cmis:isImmutable:false
cmis:isMajorVersion:false
cmis:objectId:workspace://SpacesStore/2cf36ad5-92b0-4731-94a4-9f3fef25b479
```

Retrieving objects

You can grab an object several different ways:

- You can run a CMIS query.
- You can ask the repository to give you one for a specific path or object ID.
- You can traverse the repository using the children and/or descendants of a folder.
- You can get the source and target objects tied together by a relationship.

Here are some examples that show these options at work:

1. Find the doc created in the previous section with a full-text search.
```
>>> results = repo.query("select * from cmis:document where contains('test')")
>>> for result in results:
...     print result.name
...
Test Document2
Example test script.js
```
2. Alternatively, you can also get objects by their path, like this:
```
>>> someDoc = repo.getObjectByPath('/someFolder/Test Document')
>>> someDoc.id
u'workspace://SpacesStore/2cf36ad5-92b0-4731-94a4-9f3fef25b479'
```
3. Or you can retrieve one using the object ID, like this:
```
>>> someDoc = repo.getObject('workspace://SpacesStore/2cf36ad5-94a4-9f3fef25b479')
>>> someDoc.name
u'Test Document'
```
4. Folder objects have getChildren() and getDescendants() methods that will return a pageable result set:
```
>>> children= someFolder.getChildren()
>>> for child in children:
...     print child.name
...
Test Document
Test Document2
```
5. Folders and Documents have a getRelationships() method that returns a set of Relationship objects. Relationship objects can give you the source and target object on each end of the relationship.
```
>>> rel = testDoc.getRelationships(includeSubRelationshipTypes='true')[0]
>>> rel.source.name
'testDoc1'
>>> rel.target.name
'testDoc2'
>>> rel.properties['cmis:objectTypeId']
'R:sc:relatedDocuments'
```

You'll see how to work with other parts of the API, including the ability to retrieve object type definitions, in the next article in the series.

Conclusion

Other articles in this series

- Part 2: Build real world ECM tools with Python and cmislib

This article has given you a brief overview of cmislib, how it came to be, what it does, and a few basic examples. Hopefully, you're now inspired to look into CMIS further. If you're interested in Python, check out cmislib. For a link, see Resources. If you aren't, explore some of the other tools

and client libraries that are out there. Last, the CMIS community needs you. You can help out in many ways:

- If there isn't a client library written in your favorite language, create one as an open source project.
- Help your repository vendor test their CMIS implementation.
- Write integrations for your favorite portal or presentation framework to enable it to easily work with CMIS repositories.
- Pitch in on existing open source CMIS projects like cmislib and Apache Chemistry.
- Join the CMIS specification effort by getting involved with the OASIS Technical Committee.