OpenCMIS Server Development Guide (2ⁿᵈ Edition)

Building custom CMIS servers with the Apache Chemistry OpenCMIS Server Framework

Version History:

Version 1.0 November 6, 2013
Notes: First edition of Guide.

Version 1.1March 25, 2014
Notes: Added sections explaining new 2014 server extensions and new subproject with a reference extension.   FileBridge is updated to support extensions.

Version 2.0.1 October 27, 2014
Notes: Update all of guide and finished coverage of OpenCMIS server plugins.

Version 2.0.2 June 1, 2016 : Added ISBN number

The most current version of this document can be found at :
https://github.com/cmisdocs/ServerDevelopmentGuideV2
in the ./docs subdirectory.

Authors:
Jay Brown, ECM Architect, IBM
Florian Müller, ECM Architect, SAP

Table of Contents

Introduction

Overview for Parts 1 and 2

In part one of this tutorial you will be introduced to the Apache Chemistry project, its architecture, tools and APIs but from a perspective of a developer that needs to build a custom server. We will be focusing specifically on building our server using Java and even more specifically using the Apache Chemistry OpenCMIS Server Framework (version 0.11.0 or higher).

In part two, you will learn about the new (as of 2014) runtime OpenCMIS server plugins. Both how to support them from your server and how to build and deploy them as well.

A note about extensions versus plugins:

In the most recent version of the guide (version 1.1) we referred to OpenCMIS server 'plugins' as server 'extensions'. This caused some confusion with the OASIS CMIS specification extensions. So we have chosen to refer to them as plugins from here out to reduce confusion.   OpenCMIS Plugins are code that is added to a working CMIS server as described in part 2 of this guide.   OASIS CMIS extensions are specifications for additional server and client functionality from the OASIS CMIS technical committee. OpenCMIS plugins exist entirely in the domain of Apache Chemistry OpenCMIS server framework.

Prerequisites

- Experience with Java development using Eclipse and Maven.
- It is assumed the reader has a good familiarity with the CMIS specification and its purpose.
    - For a quick tour, the Apache Chemistry site has a "What is CMIS?" page with lots of useful links to get you acquainted here: http://chemistry.apache.org/project/cmis.html.
    - Also the first chapter of "CMIS and Apache Chemistry in Action" is a very good introduction and is available as a free PDF download at Manning's site here: http://www.manning.com/mueller/

- Finally it is always a good idea to keep a copy of the CMIS specification handy. (See resources section at end of document for links.)
- A familiarity with Apache Chemistry CMIS Workbench is helpful but not mandatory. Refer to the 10 minute video introducing this tool if you are not already familiar with it here: http://www.youtube.com/watch?v=akvCDVh03qs . Note that this video refers to a much older version of CMIS Workbench than is available today but the general concepts have not changed.

Goals of the tutorial

- Understand how to use the OpenCMIS server framework to build CMIS 1.0+1.1 compliant servers.
- Understand the scope of server side Apache Chemistry OpenCMIS and be able to build its dependencies locally.
- Understand the CMIS Workbench tool and become familiar with it by using it throughout this tutorial.
- (Part II) Understand how OpenCMIS server plugins work, how they are designed and how to deploy them to off the shelf servers like FileNet CMIS.

Tutorial task description

In this tutorial we will be using OpenCMIS APIs to build a CMIS 1.1 server on top of a local filesystem using the latest version of the Apache Chemistry OpenCMIS Server Framework. We will show all of the aspects of this development from the initial setup of a blank server template using a Maven archetype, through development, deployment and testing of the finished product. The purpose of this exercise is to demonstrate the server framework on top of a simple and universally understood back end (a typical filesystem). Once you understand the steps in this document you should be able to extrapolate this knowledge into what will be required to build a CMIS server on top of whatever server or legacy application that you wish to expose to the world of CMIS clients.

Initial setup of your developer environment

The tutorial assumes you have the following installed:
- Eclipse EE edition "Luna" (v 4.4.0) or later
  - This is the Java IDE that we will be using for editing / compiling and running our project.
- Apache Maven 3.0.5 or later
  - The command line version of Maven for building our project outside of the Eclipse IDE.
- SVN client, version 1.6.19 or later
  - A subversion client is needed in order to retrieve the project from our source repository. If you don't want to use SVN you can also go directly to the source location and download a zipped bundle of the source code.
- Java JDK version 1.7.0_25 or later
  - Any current JDK should work. Has also been tested with IBM's JDK 6 and above.
- Apache Tomcat v 7.x

- ○ The servlet container for hosting our server. Tomcat is not required but deploying this server to other JEE containers like WebSphere, JBoss and WebLogic is outside the scope of this tutorial.

All of the these tools we have chosen for this tutorial can be freely obtained and can be installed on Windows, OSX or Linux.

Getting and building the latest OpenCMIS libraries

Although all you need to perform these exercises is the latest version of the OpenCMIS libraries we are going to start off by downloading the source and compiling all of the libraries ourselves. This includes the Apache Chemistry Workbench CMIS client application that we will be using for testing our server. Once you see how easy this is (just a couple of commands) you will be a lot more comfortable doing this if you ever end up in a situation where you need a fix or a feature that is only in the latest snapshot. So lets get started!

Initial build of OpenCMIS

Our first step will be to download the latest version of the source code from the Apache Chemistry SVN repository. Create a directory where you would like the Apache Chemistry source tree to be located. This is not going to be the location of our project, but the build location of the libraries that our project will be using. It can be under the same project directory if you wish but this may be used by other projects later so a shared location may be preferable. When completed it will take about 250 MB of space after your first build is completed.

From a command prompt, cd into your new directory and enter:

```
svn checkout http://svn.apache.org/repos/asf/chemistry/opencmis/trunk
```

then press enter to start the checkout operation. This may take a few minutes depending on your connection speed.

Building OpenCMIS

Note: Refer to the "Initial setup of your developer environment" section before starting this section.

To build the entire tree we need to change directories to the top of the source tree where the top level Maven `pom.xml` file is located. CD into the `./trunk` directory if you are not already there.

Since the build is pretty large, let's increase the max Java heap size for Maven here:

```
export MAVEN_OPTS='-Xmx1024m -XX:MaxPermSize=512m'
```

Windows Note: use 'set' in place of 'export' for (and omit the quotes) throughout this document.
Then we will run the build while skipping the tests like this:
```
mvn clean install -Dmaven.test.skip=true
```

If you see out of memory errors, etc., try running a second time without the clean like this:

```
mvn install -Dmaven.test.skip=true
    -Dorg.apache.chemistry.opencmis.tck.test=false
```

This is also a much faster way to build if you are just refreshing after a few small changes. That last -D parameter will turn off the TCK tests during the build and reduce total build time and memory needed even further.

Output from a successful run will look something like this.

```
… <lots of build output stuff>
[INFO] Apache Chemistry OpenCMIS .......................... SUCCESS [12.223s]
[INFO] OpenCMIS Commons API .............................. SUCCESS [16.552s]
[INFO] OpenCMIS Commons Implementation ................... SUCCESS [29.901s]
[INFO] OpenCMIS Client API ............................... SUCCESS [1.520s]
[INFO] OpenCMIS Client Bindings Implementation ........... SUCCESS [9.572s]
[INFO] OpenCMIS Client Implementation .................... SUCCESS [6.003s]
[INFO] OpenCMIS Server Support ........................... SUCCESS [13.602s]
[INFO] OpenCMIS Server Implementation .................... SUCCESS [7.383s]
[INFO] OpenCMIS Server Implementation WAR packaging ...... SUCCESS [2.776s]
[INFO] OpenCMIS Test Utilities ........................... SUCCESS [1.613s]
[INFO] OpenCMIS InMemory Server WAR packaging ............ SUCCESS [5.783s]
[INFO] OpenCMIS FileShare Server Implementation .......... SUCCESS [2.234s]
[INFO] OpenCMIS Server Archetype ......................... SUCCESS [9.044s]
[INFO] OpenCMIS Server Extension Archetype ............... SUCCESS [0.582s]
[INFO] OpenCMIS Bridge WAR packaging ..................... SUCCESS [2.488s]
[INFO] OpenCMIS Test Compatibility Kit ................... SUCCESS [4.782s]
[INFO] OpenCMIS Full Integration Tests ................... SUCCESS [19.733s]
[INFO] OpenCMIS Tools .................................... SUCCESS [4.266s]
[INFO] OpenCMIS Browser .................................. SUCCESS [0.967s]
[INFO] OpenCMIS Browser Application ...................... SUCCESS [1.002s]
[INFO] OpenCMIS Workbench ................................ SUCCESS [44.898s]
[INFO] OpenCMIS OSGi Client Wrapper ...................... SUCCESS [1.642s]
[INFO] OpenCMIS Android Client ........................... SUCCESS [30.801s]
[INFO] OpenCMIS Assemblies Distribution .................. SUCCESS [0.503s]
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time: 3:55.429s
[INFO] Finished at: Sat Sep 06 13:54:38 CEST 2014
[INFO] Final Memory: 142M/465M
[INFO] ------------------------------------------------------------------------
[INFO] Shutdown hook executing
[INFO] Shutdown hook complete
```

Getting the project source from GitHub

All of the source code for this project is publicly available on GitHub and is Apache 2 licensed so that you are free to copy it, reuse it, sell it or whatever you want. Just remember this code is not supported for production use. (legal disclaimer)

Create a directory to contain the source code tree for the project. For consistency throughout the tutorial let's hypothetically use:

```
/project.code
```

In a terminal window cd to your new project directory and execute the following command to checkout the project source tree.

```
svn checkout https://github.com/cmisdocs/ServerDevelopmentGuideV2.git
```

Note: You may also use a git client of your choice including the one that is available for Eclipse which is what we will be using in the Hands on Lab.

After this completes, take a moment to examine the structure of the project before we move on to building it for the first time.

The illustration below shows the directory once it has been extracted or checked out.

Name	Size	Type
▼ ServerDevelopmentGuideV2.git	2 items	folder
▶ branches	0 items	folder
▼ trunk	5 items	folder
▼ src	2 items	folder
▼ main	2 items	folder
▼ java	1 item	folder
▼ org	2 items	folder
▼ example	1 item	folder
▼ cmis	1 item	folder
▶ server	9 items	folder
▶ foo	2 items	folder
▶ webapp	2 items	folder
▼ test	1 item	folder
▼ java	1 item	folder
readme.txt	6 bytes	plain text document
extra.credit.txt	929 bytes	plain text document
LICENSE	11.3 kB	plain text document
pom.xml	5.9 kB	XML document
README.md	97 bytes	Markdown document

The contents are as follows:

- trunk: Contains the actual working version of the FileBridge sample CMIS server application from the guide as well as the code from part 2.
- doc: (not shown here) Contains the various renderings of this document (PDF, etc)
- pom.xml (the Maven POM file for the project used in this guide)

Building the solution from the command line

Throughout this project we will be using Eclipse to edit, compile, debug and run our project; but before we get to Eclipse it is always good to know how to build the project from the command line. Perhaps you prefer Vi, Emacs, Netbeans or even Notepad (no judging here). In all these cases we would change to the directory that contains your pom.xml file. In the case of our hypothetical tutorial directory we chose earlier that will be the

```
/project.code/cmisFileBridge-master
```

directory and execute the following command:
(Note this must be executed from the same location as the project pom.xml file)
```
mvn clean install -Dmaven.test.skip=true
```

(does that command look familiar?)

After a few seconds, you should see something like this at the tail end of a lot of output.

```
[INFO] Webapp assembled in [764 msecs]
[INFO] Building war: /project.code/cmisFileBridge/trunk/target/server-1.0-SNAPSHOT.war
[INFO]
[INFO] --- maven-install-plugin:2.3.1:install (default-install) @ server ---
[INFO] Installing /project.code/cmisFileBridge/trunk/target/server-1.0-SNAPSHOT.war to
    /root/.m2/repository/org/example/cmis/server/1.0-SNAPSHOT/server-1.0-SNAPSHOT.war
[INFO] Installing /project.code/cmisFileBridge/trunk/pom.xml to
    /root/.m2/repository/org/example/cmis/server/1.0-SNAPSHOT/server-1.0-SNAPSHOT.pom
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time: 15.560s
[INFO] Finished at: Wed Aug 28 18:02:01 EDT 2013
[INFO] Final Memory: 16M/55M
[INFO] ------------------------------------------------------------------------
```

If you want to deploy this WAR file manually to your own Tomcat (or other container) have a look at the target directory under trunk (shown in illustration below) and you will see the xxx-SNAPSHOT.war file (shown highlighted).
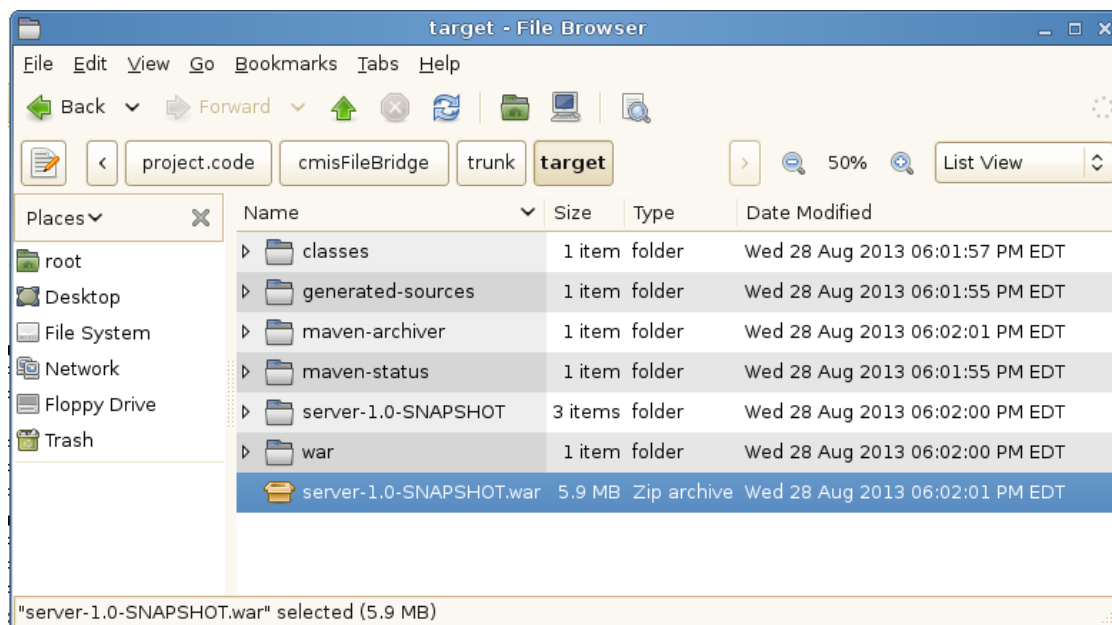
Going forward we will be letting eclipse handle all of our deployments but again its nice to know how to do this from the command line in case you need to automate your build later.

Building and running from Eclipse

Please start your copy of Eclipse for the next part. From here out we will be using absolute paths for locations of files so just compute the difference for your local env based on our initial hypothetical root path.

Importing the OpenCMIS code formatter settings into Eclipse

Optionally, if you would like to use the OpenCMIS code formatter options that we use in the rest of the code you can import them into your Eclipse workspace by going to Window / Preferences / Java / Code Style / Formatter / Import... and selecting the _dev/Formatting_presets.xml file from the OpenCMIS trunk that we just updated.

Importing the project into Eclipse

From the file menu select 'import' and you will see the dialog in the illustration below:
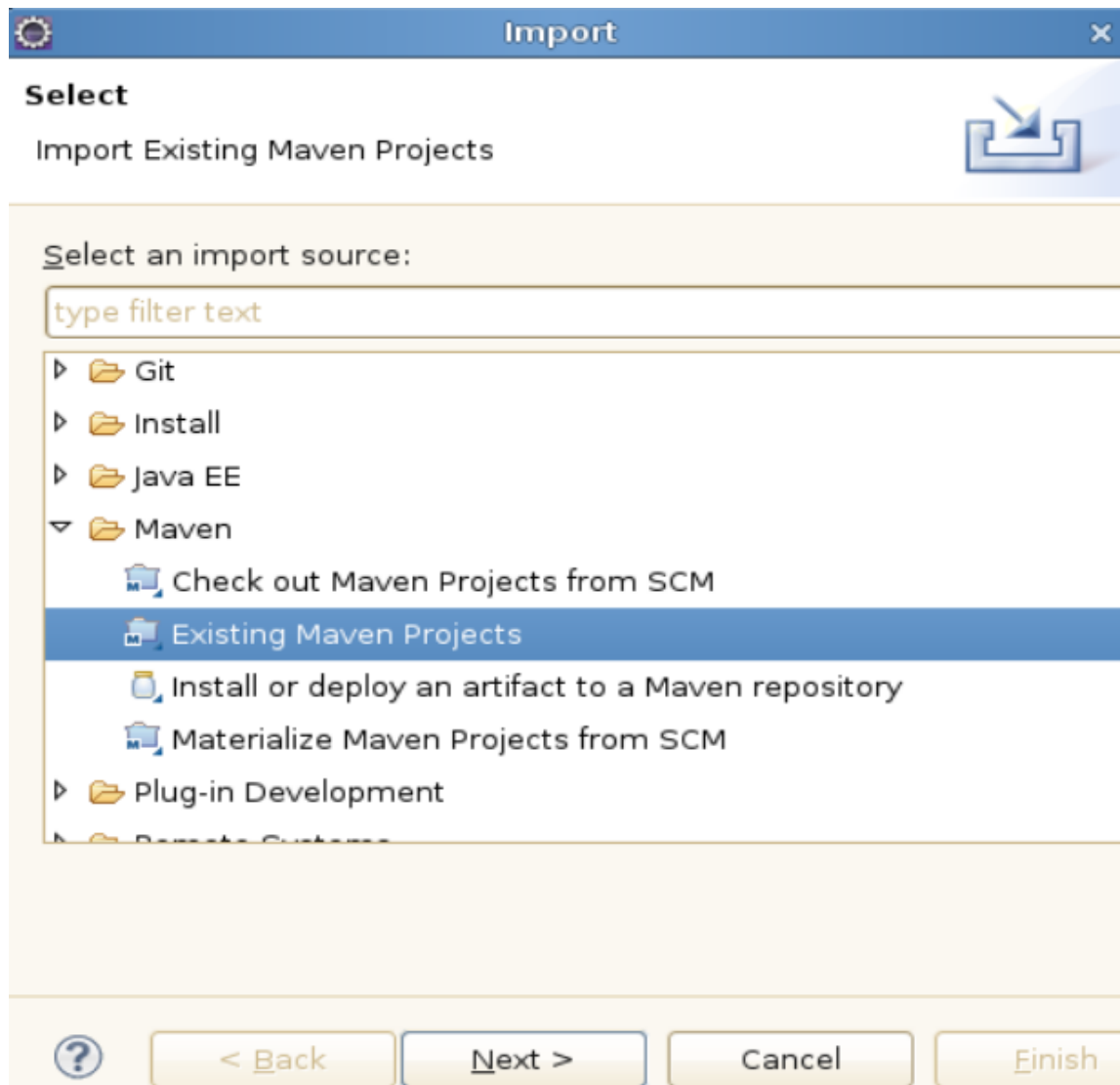
Open up the 'Maven' submenu and select 'Existing Maven Projects' then click 'Next'.
On the next dialog select the 'browse' button to the right of the 'root directory' field then navigate to your trunk directory where the project `pom.xml` file it located. For the our tutorial environment the location is

`/project.code/ServerDevelopmentGuideV2.git/trunk`

After selecting the directory, Eclipse will do a bit of processing (reading the `pom.xml` file) and then display an entry for your project (pre checked) as shown in the illustration below:
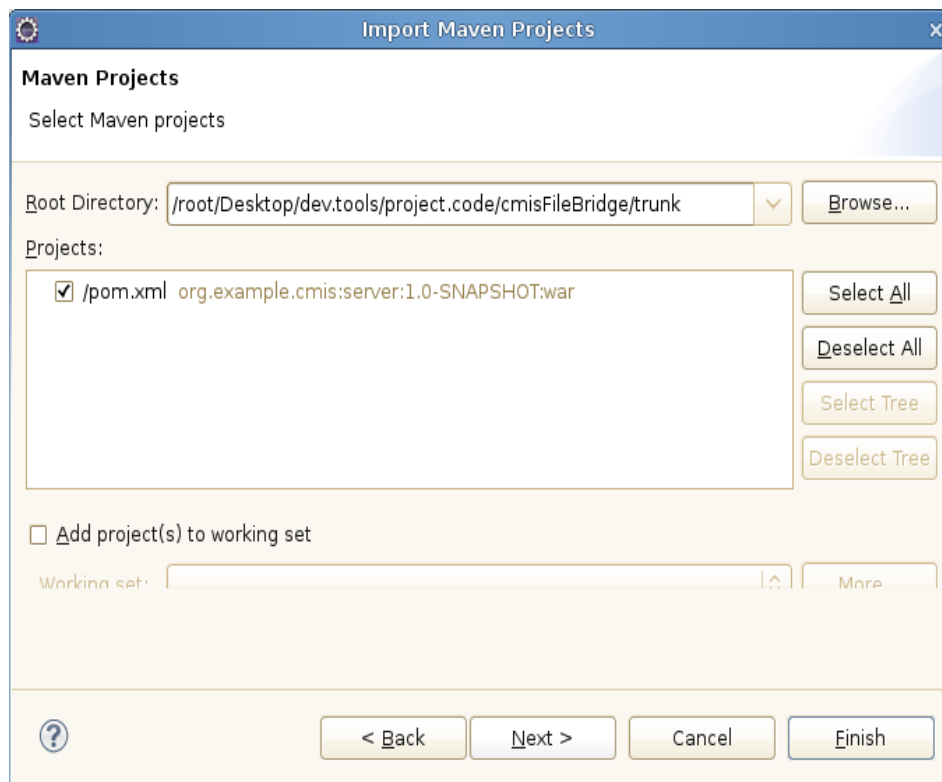
Select 'Finish' to finish the import and go back to your workspace to see the newly imported CMIS server project. The following illustration shows the new project explorer view of the project showing the files in the `org.example.cmis.server` package.
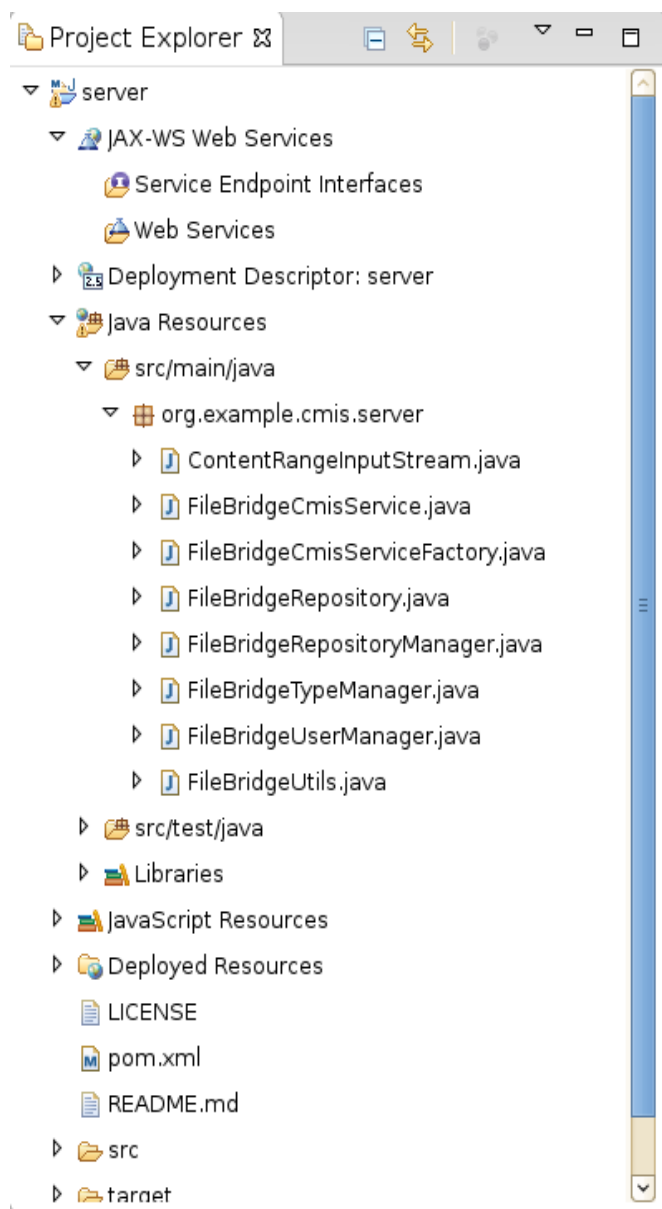
Now that we have the project successfully imported we will setup an embedded Tomcat instance so that Eclipse will have a container target for deployments.

Setting up a Tomcat server target in Eclipse

Download Apache Tomcat 7 from any of the standard mirrors. (e.g. http://tomcat.apache.org/download-70.cgi) and extract the archive to a working location where you want to keep tools for your project. On our hypothetical tutorial image we place the .tar.gz file in the

```
/root/Desktop/dev.tools/to.install
```

directory.

Extract this file in place which will create a new

```
/root/Desktop/dev.tools/to.install/apache-tomcat-7.0.42
```

directory. Make a note of wherever you place this directory since you will need it in the next step.

Next start your Eclipse again and look at the bottom of your Eclipse workspace at the lower most row of tabs. There you will find a 'Server' tab. Select it and then click on the blue link to create a new server.
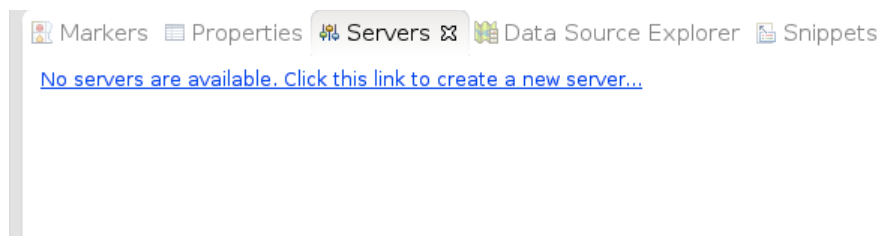
Illustration 1: Server tab showing no servers currently configured

Then you will be prompted with the 'Define a new server' dialog. Open up the Apache dropdown and select 'Tomcat v7.0 server' then select the 'Next' button at the bottom.
In the 'Tomcat Installation Directory' field enter in the path where your Apache Tomcat is located (for the tutorial we are using `/root/Desktop/dev.tools/to.install/apache-tomcat-7.0.42`) then click 'Next'.
On this last dialog (Illustration below) you will select your server app on the left and press the 'Add' button which will move it over into the 'Configured' side like shown in the illustration below:
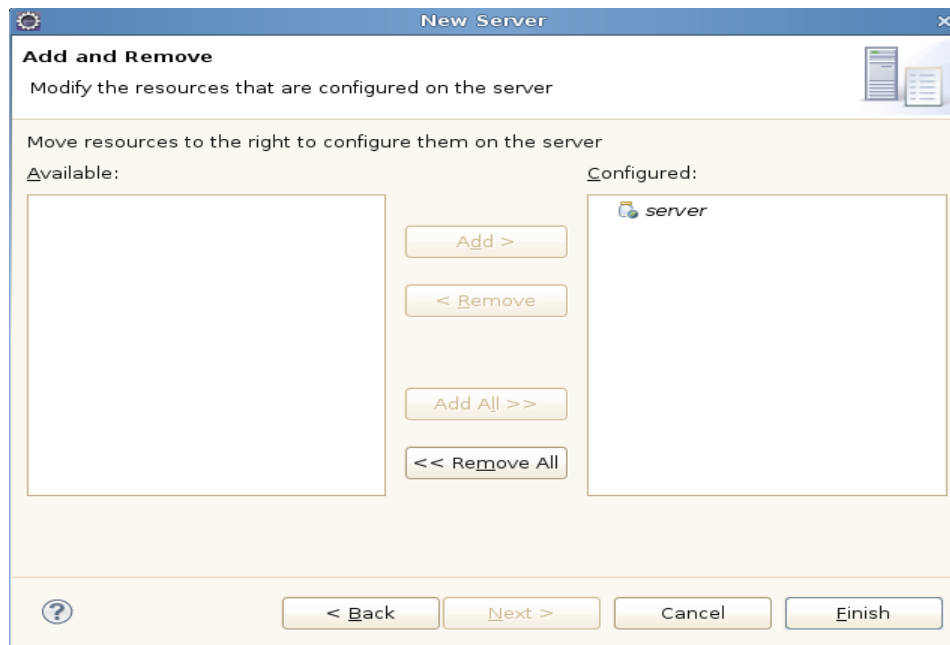
Ilustration 2: New server dialog showing our server app configured

Press 'Finish' to complete the configuration.

A note about running from Windows

If you are running from Windows you will need to change the `repository.properties` file (we will talk more about this file later).
Edit the file (see illustration below for location) and change the line

```
repository.test = /
```

to point to a more Windows friendly path like

```
repository.test = c:\\myrootdirectory
```

and make sure whatever directory you indicate actually exists and has some files and directories already populated.

The relative location of this properties file is shown in the illustration below:
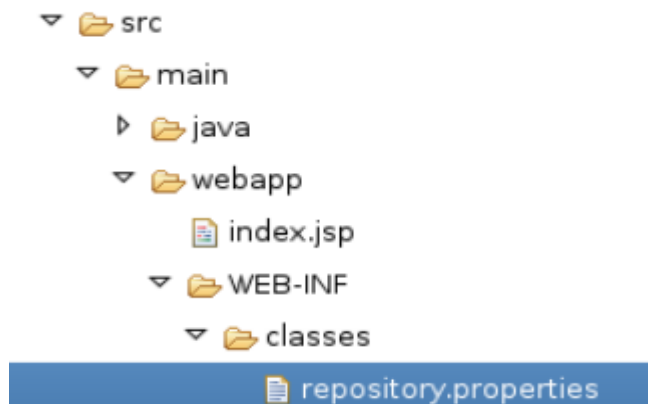
Running and debugging from Eclipse

Now all that is left is to tell Eclipse to run our project. Select your new Tomcat server instance in the Servers tab and press the green 'start' button as shown in the illustration below:
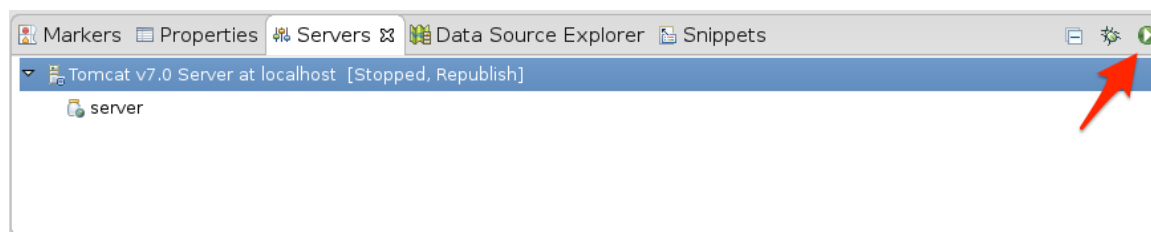
You will know when the server has started when you see an info message in your 'Console' tab showing successful startup like this:

```
INFO: Server startup in 14322 ms
```

A note about startup timeouts

If you are running on a slower machine it is possible that the server will not start up within the default timeout window that is defined by Eclipse. If you notice that your server is timing out you can increase the timeout value by double clicking on the server entry in your 'Servers' view. This will open up the Tomcat server properties page (shown in illustration below). On that page you will find a 'Timeouts' section where you can increase the value (from 45 seconds) to whatever is appropriate for your machine (perhaps 90 seconds).
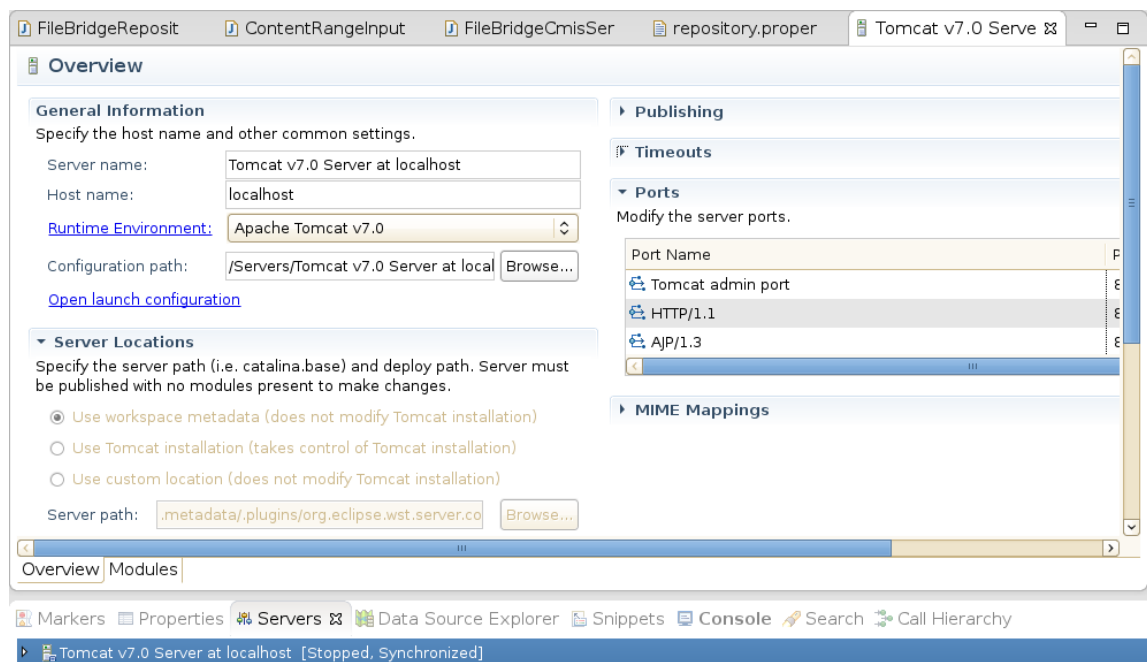
At this point lets bring up a web browser to see our new server's landing page and make sure it really did startup. Go to http://localhost:8080/server/ and you should see a page like the following illustration.
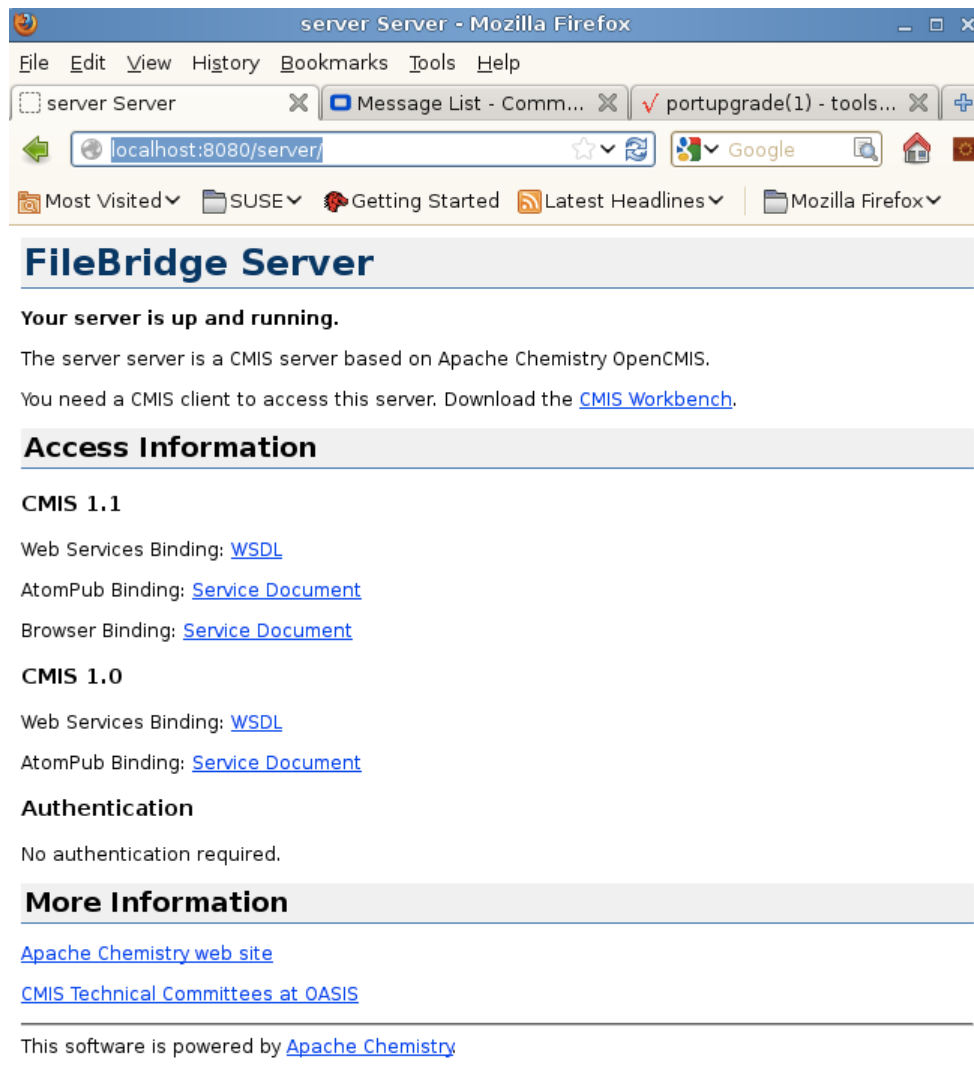
Illustration 3: Firefox showing the landing page for the FileBridge CMIS server

Now we are ready to connect CMIS Workbench and poke around to see what this little bit of code can really do.

Connecting CMIS Workbench to our local server

Ordinarily to run CMIS Workbench you would just download the latest binary from the download page but since we have just did a local build of the entire OpenCMIS tree lets go ahead and use the one that we have just built instead.

Go back to your chemistry.trunk directory and navigate down into

```
/trunk/chemistry-opencmis-workbench/chemistry-opencmis-
workbench/target
```

Here you will grab the file that ends with `-SNAPSHOT-full.zip` as shown in the illustration below:
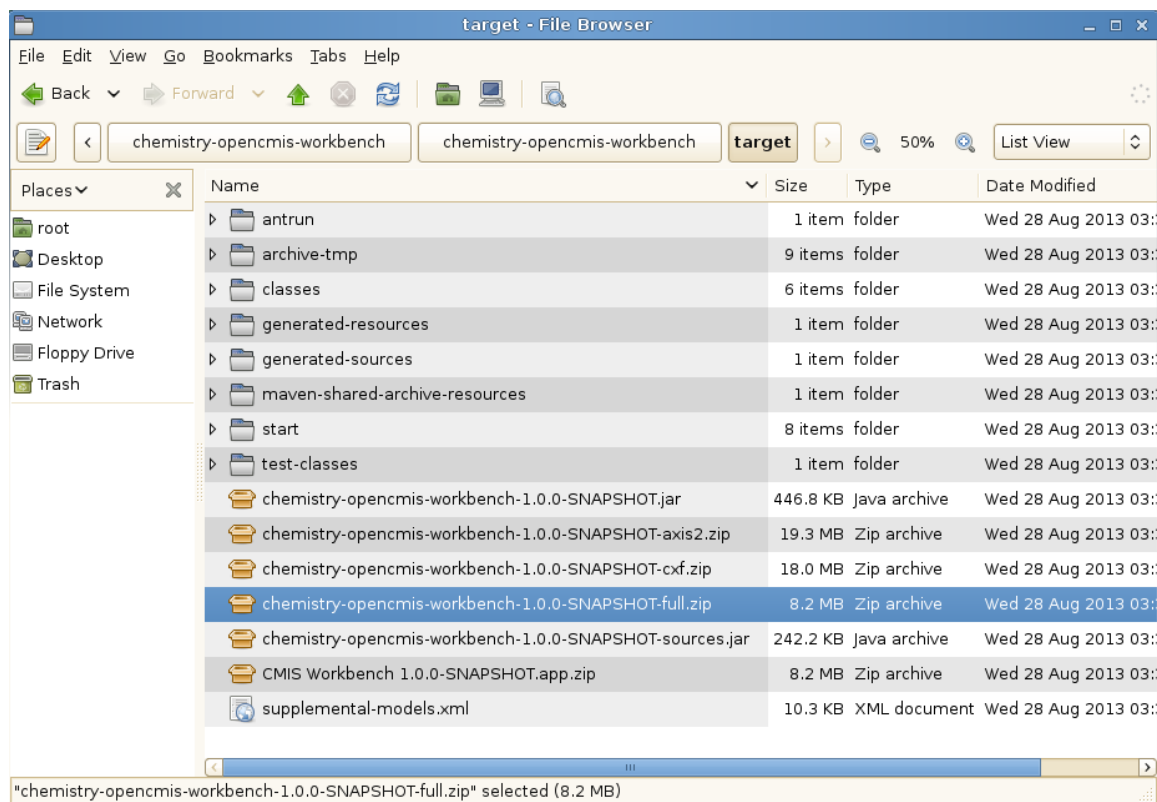
Copy the zip file up to a shared dev.tools directory (in our tutorial we will use ../desktop/dev.tools). There create a new sub-directory named Workbench. Once there uncompress the file.

Starting CMIS Workbench

Running the workbench is now just a matter of running `workbench.sh` (if on Linux or Mac – you may have to 'chmod 777 workbench.sh' first) or `workbench.bat` (if Windows). After running this command you will see the following dialog (illustration below) asking you for connection information to your server.

Since our server is still running all we need to do is enter in the connection info and we will be off and running.

Select the 'Expert' tab and paste in the following info:

```
org.apache.chemistry.opencmis.binding.spi.type=browser
org.apache.chemistry.opencmis.binding.browser.url=http://localhost:8080/server/browser
org.apache.chemistry.opencmis.user=test
org.apache.chemistry.opencmis.password=test
org.apache.chemistry.opencmis.binding.compression=true
org.apache.chemistry.opencmis.binding.cookies=true
```

Note: You don't have to use the expert tab. You can also separately enter in the URL, user and password values then click on the 'Browser' radio button. Same result. Its just less steps to use the expert page (e.g. a single paste).

Select the 'Load Repositories' button then click on 'Login' and you should almost immediately see the root directory displayed like the illustration below:
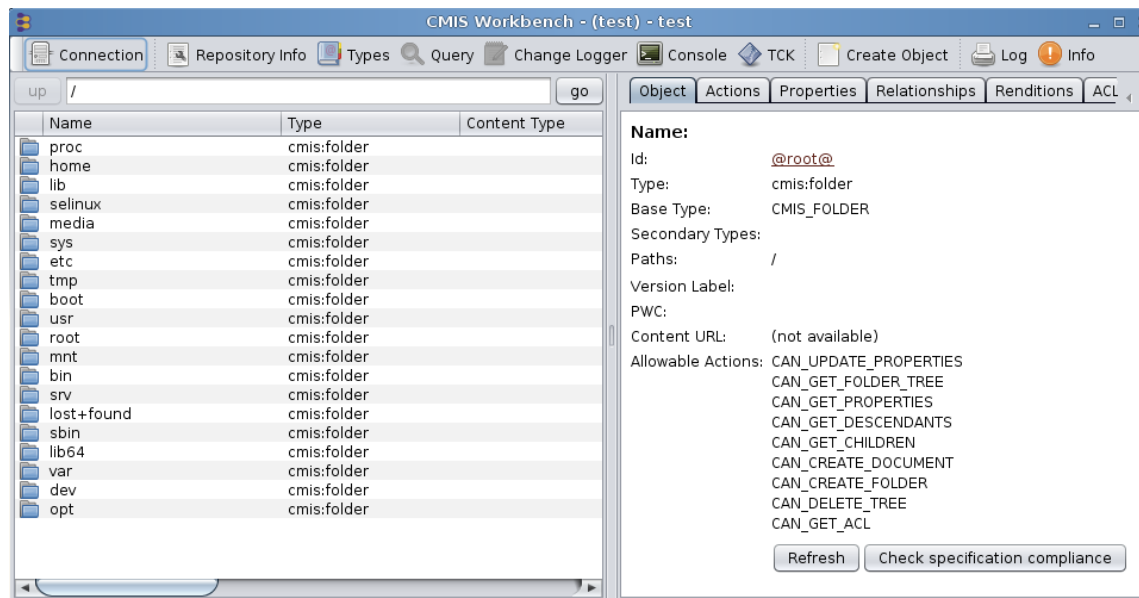
Ilustration 4: First screen you see in Workbench after login and selecting a repository

Creating a new server project from scratch (Optional)

Now that you have seen how to build and run the solution let's get down to how to build a project like this from scratch. The text below shows the command line options to pass to Maven (mvn) to generate our server from the OpenCMIS archetype. This is the command that we used to generate FileBridge originally. If you want to see this in action create a temp directory and give it a shot now.

```
mvn archetype:generate \
-DgroupId=org.example.cmis \
-DartifactId=server \
-Dversion=1.0-SNAPSHOT \
-Dpackage=org.example.cmis.server \
-DprojectPrefix=FileBridge \
-DarchetypeGroupId=org.apache.chemistry.opencmis \
-DarchetypeArtifactId=chemistry-opencmis-server-archetype \
-DarchetypeVersion=1.0.0-SNAPSHOT \
-DinteractiveMode=false
```

Note: This archetype generation step can also be done GUI-style, with the latest version of Eclipse (with Maven integration). We are leaving that out of the tutorial to save space but feel free to use that instead (as an extra exercise) if you prefer.

Some highlights to note here concerning these values:

- **groupId, artifactId, and version**: These are the Maven 'coordinates' for the code you are generating. (see: http://maven.apache.org/pom.html#Maven_Coordinates for more info on these)
- **package**: The Java package for the code.
- **ProjectPrefix**: Prefix for all the classes that will be generated. For example, the prefix FileBridge generates the classes `FileBridgeCmisService` and `FileBridgeCmisServiceFactory`.
- **archetypeGroupId, archetypeArtifactId, and archetypeVersion**: OpenCMIS archetype and OpenCMIS version that should be used. The archetype OpenCMIS 1.0 may be available by the time you read this. This version also defines the runtime OpenCMIS Server Framework version.
- **InteractiveMode**: If false, Maven won't prompt you for confirmations during the generation process.

A cooks tour of the FileBridge project

Before we explain the different components and classes of the FileBridge, you have to understand first how the OpenCMIS Server Framework works at a more generic level. The following material is a condensed version of the material provided in the initial lecture we did on this subject at IBM's Insight conference in 2014 prior to the lab (based on this tutorial). It is provided as a convenience for the folks who did not attend the lab in person. For a more complete explanation of these issues please refer to chapter 14 in the 'CMIS and Apache Chemistry in Action' book (see References). As we move through and discuss the different key classes and interfaces, take a moment to examine the corresponding code in your solution project.

OpenCMIS Server Framework Interfaces

The OpenCMIS Server Framework is a web application that runs on top of a servlet engine such as Tomcat, Jetty, or an application server like WebSphere (shown in the illustration below). It handles all CMIS requests and responses and does all the XML and JSON processing. It also hides the details of CMIS bindings from the developer by turning incoming data into Java objects and outgoing data from Java objects into XML or JSON.
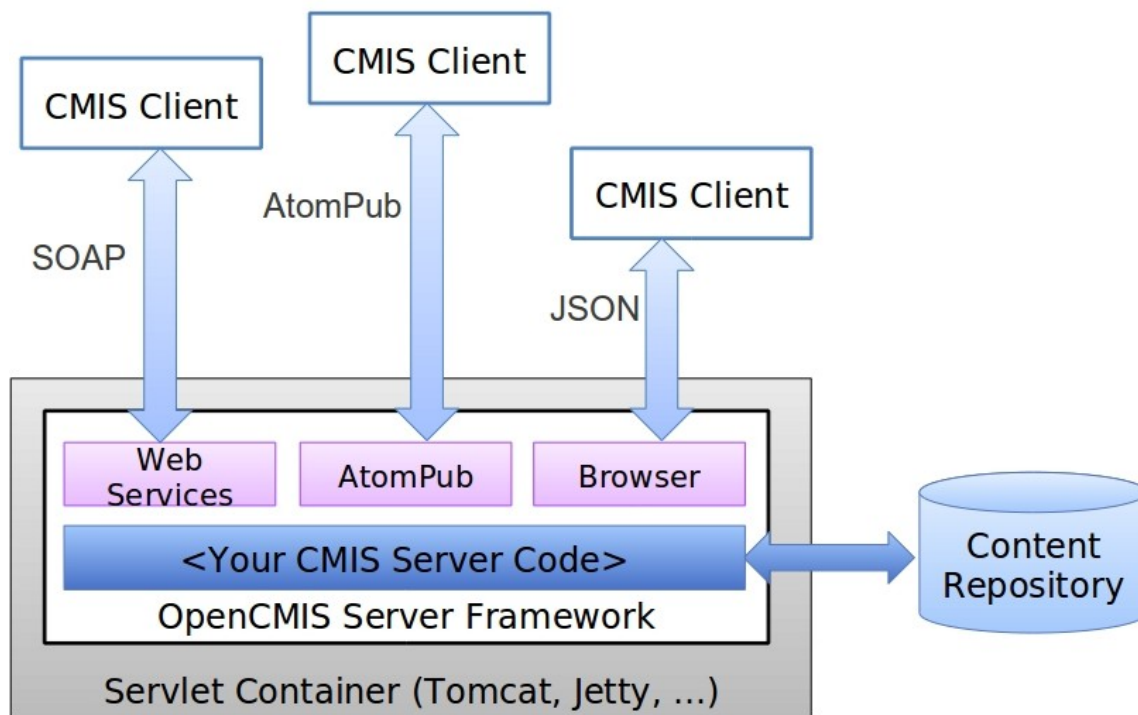
To connect the server framework to the content repository, you have to implement two Java interfaces: `CmisService` and `CmisServiceFactory`.

CmisService

The `CmisService` interface aggregates all CMIS 1.0 and CMIS 1.1 operations plus a few additional methods. There are over 50 methods in total that can be implemented. The methods and method parameters are named after the operations that are described in the "Services" section of the CMIS specification. The implementation of the `CmisService` interface is supposed to behave as defined in the specification. That includes throwing the exceptions documented there.
Implementing all these methods sounds tedious doesn't it? Luckily, OpenCMIS provides the abstract class `AbstractCmisService`, which implements the `CmisService` interface and provides convenience implementations for most methods. It reduces the number of required methods to just six. Providing implementations for these six methods doesn't make the CMIS connector specification compliant, but it is sufficient for many CMIS clients (like the CMIS Workbench) to navigate through the folder structure. It's recommended to extend this class instead of implementing the interface directly. You'll see an example later when we discuss the `FileBridgeCmisService` class.

CmisServiceFactory

The main task of an implementation of the `CmisServiceFactory` interface is to provide
`CmisService` objects. Whenever the server framework receives a request, it asks the
`CmisServiceFactory` for a `CmisService` object, which is then used to process the request.
There is only one `CmisServiceFactory` object per web application. This object also manages the
initialization and shut down of the CMIS connector and provides some configuration values for the
server framework. The `CmisServiceFactory` object must be thread-safe but the `CmisService`
objects it produces don't need to be because they are only used for one request.
OpenCMIS provides the abstract class `AbstractServiceFactory`, which should be used to build
a service factory because it sets some sensible default values among other details. The
`FileBridgeCmisServiceFactory` class, which we will walk through later, also extends the
AbstractServiceFactory class.

Next, let's see how the server framework uses these objects.

OpenCMIS Server Framework Operation

The server framework web application consists of five servlets and two context listeners (shown in
illustration below). It provides two CMIS 1.0 endpoints (for the AtomPub and the Web Services
binding) and three CMIS 1.1 endpoints (for the AtomPub, the Web Services, and the Browser binding).
Each endpoint can be disabled if necessary by editing the `web.xml` of the web application.
One of the context listeners sets up the `CmisServiceFactory` object and calls it's init() method
when the web application starts up.
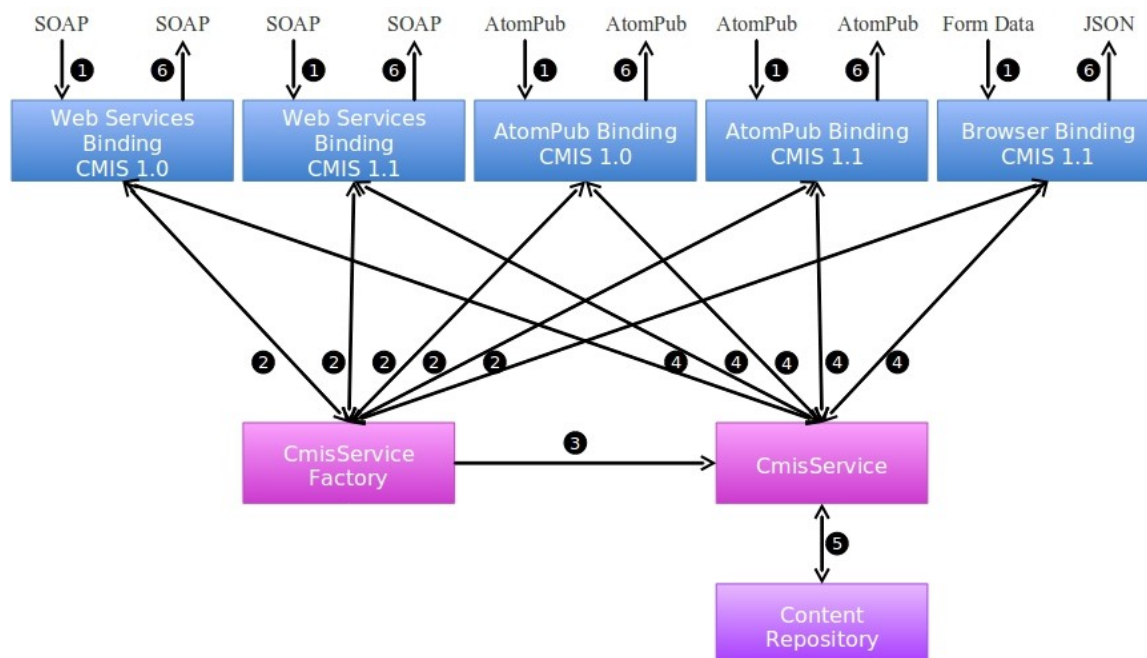
Incoming CMIS requests are processed by the servlets. The requests are parsed, checked for syntactical correctness, and the data is converted into Java objects. The framework then requests a `CmisService` object from `CmisServiceFactory` and calls the suitable method with the received data. The data returned by the method is converted into XML or JSON and sent back to the client.

Now you should have a rough understanding how the server framework works.
Next lets look into the FileBridge implementation – class by class.

FileBridgeCmisServiceFactory

As you might figure out from the name, the `FileBridgeCmisServiceFactory` is the service factory class for the `FileBridgeService`. Its `init()` method gets the repository configuration and sets up all necessary objects. The method receives a map of configuration parameters. This map represents the content of the `repository.properties` file, which resides in the classpath. The `repository.properties` file is used by the server framework to identify the service factory class and must at least contain an entry with the key "class" and the fully qualified classname of the service factory as the value. The `repository.properties` file can also contain any other configuration. In case of the FileBridge, we use it to configure the repositories, their root paths on disc, and the logins. The `readConfiguration()` method iterates through the map, collects all the repository and login details and stores them in the repository manager (`FileBridgeRepositoryManager`) and the user manager (`FileBridgeUserManager`). As you will see in Part II, this file is also used to hook in your server plugin classes.

The factory's main task, though, is to serve `FileBridgeCmisService` objects. The framework calls the `getService()` method whenever it needs one. The framework provides a `CallContext` object, which contains all kinds of details about the incoming call. That includes the user name and the password that the client sent. Before the `FileBridgeCmisServiceFactory` returns a `FileBridgeCmisService` object, it hands the `CallContext` object over to the user manager (`FileBridgeUserManager`) to authenticate the user. If the authentication fails, it throws a `CmisPermissionDeniedException`.

Now it's time to serve a `FileBridgeCmisService` object. There are multiple ways to manage those objects. The easiest, but most inefficient way would be to create a new object every time. Here, we decided to use a ThreadLocal. Over time each thread will have it's own object that is reused when a subsequent request hits that thread.
`FileBridgeCmisService` objects are very lightweight and are only proxies for repository objects (FileBridgeRepository), which we'll discuss in a moment. For heavyweight `CmisService` objects or `CmisService` objects that are expensive to create, a pool of objects might be a better option.

`FileBridgeCmisServiceFactory` does not return a `FileBridgeCmisService` object directly, but a `FileBridgeCmisService` object that is wrapped by one or more service wrappers. Part 2 explains in detail how service wrappers work and how to implement and configure them. By default, a `FileBridgeCmisService` object is wrapped by at least the

`ConformanceCmisServiceWrapper`, which is set up in the init method of the factory. The `ConformanceCmisServiceWrapper` is an optional class provided by OpenCMIS. The wrapper checks a request for specification conformance before the request is forwarded to the wrapped service object. If a client sends an invalid request, for example if it didn't provide a mandatory parameter, the wrapper throws the appropriate exception without bothering the service object. The wrapper also sets parameter values for parameters the client didn't provide but the specification defines default values for. Using this wrapper is recommended because it helps building more robust CMIS servers, however it is not required.

Once the `FileBridgeCmisService` object has been created or retrieved, the `CallContext` object has to be handed over to the object. Here we use a custom `CallContext` object that wraps the one originally provided by OpenCMIS. There is more information on `CallContext` objects in part 2. Eventually, the service factory returns the object to the server framework.

FileBridgeCmisService

The `FileBridgeCmisService` class implements the `CmisService` interface and therefore has to provide more than 50 method implementations. By extending the `AbstractCmisService` class, we can focus here on just the methods that we want to (and can) implement.
The `FileBridgeCmisService` only contains a tiny amount of logic since its task is merely to forward the call to the appropriate repository instance.

FileBridgeRepositoryManager

The repository manager maps repository IDs to `FileBridgeRepository` object, which in turn contains the repository logic. The repository manager is set up when the `FileBridgeCmisServiceFactory` starts up and is used by the `FileBridgeCmisService` to find the right `FileBridgeRepository` object.

FileBridgeRepository

The `FileBridgeRepository` class contains the main logic of our CMIS server. There is one instance per repository at runtime. **It has to be thread-safe because multiple threads could access the same repository at the same time.**
The `FileBridgeRepository` class maps CMIS operations to file system operations. Most of the code is straightforward. Creating a document or folder maps to creating a file or directory on the file system. The CMIS operations `getObject` and `getContentStream` provide metadata and content respectively of a file (or folder). The `getChildren` and `getDescendants` operations return a list or tree of children of a directory. And so on.
We skip the details here because the implementation of this class is the main topic of this tutorial's exercises.

ContentRangeInputStream

The `getContentStream` operation allows a client to request an excerpt from a document content by

proving an offset and a length. This class is a simple wrapper around a Java `InputStream` that takes the offset and length into account. The `FileBridgeRepository` uses it when `getContentStream` is called and the client has requested a content excerpt.

FileBridgeUserManager

The `FileBridgeUserManager` manages user logins and passwords. It is kept very simple and acts a placeholder for a real user management system that you would be connecting to. In FileBridge, logins are necessary to demonstrate the Allowable Actions and ACLs features. The `FileBridgeRepository` distinguishes between read-only and read-write users, which affects the Allowable Actions and ACLs.

FileBridgeUtils

This class provides a set of static helper methods. Most of them deal with extracting values from a set of properties and are mainly used in `FileBridgeRepository`.

FileBridgeTypeManager

This class manages the CMIS type system. All repositories share the same type system in this implementation and therefore there is just one instance of this class at runtime.
The type manager is as simple as it can be. It only manages the two base types for documents and folders. It provides access methods to these type definitions that are similar to the CMIS operations. This class makes use of the `TypeDefinitionFactory`, which is an OpenCMIS helper class that provides methods to create and transform type and property definitions in a CMIS compliant way. It should work for all servers with a straight forward type hierarchy and provides spec conforming base type definitions. This is helpful to developers starting with CMIS setting up a spec compliant type system quickly which can otherwise be somewhat tedious.
Note: In your own custom repository you may need to have a separate set of type definitions per repository.

Tutorial exercises

For all of the exercises that follow, we will be using the same project as the one that you downloaded at the start of this tutorial here: https://github.com/cmisdocs/ServerDevelopmentGuideV2.git
For this guide we will have only one final working version of the source code that includes the server from part 1 and the plugins from part 2.   Our exercises will guide you through specific areas of the code. First explaining what is going on (often using code comments) then encouraging you to optionally make any changes you wish and observe the modified output / behavior of the server.   This allows you to stop the tutorial at any time and debug the working server to try changes as they occur to you.

Exercise 1: Filling out the RepositoryInfo structures

For our first exercise we will cover one of the first things you will code up when you create your own server. That is populating the `RepositoryInfo` structure. Now that we have CMIS 1.1 clients to accommodate (in addition to CMIS 1.0) there are a few finer points that you will need to look out for. Navigate to the `FileBridgeRepository` class in your lab copy of the project (rather than the solution). Then proceed down to the

```
private RepositoryInfo createRepositoryInfo(CmisVersion cmisVersion)
```

method.

Lets take a moment to look at the context of how this is first called. If you set a breakpoint set at the top of this method you will see the following when we startup the server.   (below) Note this is getting called before any client has tried to connect. This is part of the first time initialization and is only called at startup.

The red arrow in the illustration above shows the call to the `createServiceFactory`. There the `init()` method is called, which after setting up its 3 manager classes, calls `readConfiguration()` passing it the parameters that were read from the `repository.properties` file.

At the tail end of `readConfiguration()`, the constructor for `FileBridgeRepository` is called which calls `createRepositoryInfo` two times. Once with `Version.`*`CMIS_1_0`* passed in and once with `CmisVersion.`*`CMIS_1_1`*. You may wonder why do we need two of these?

Recall during the cooks tour in the 'OpenCMIS Server Framework Operation' section, we talked about how there are 5 servlets setup by the framework. These are divided into two groups, 2 of the endpoints for CMIS 1.0 and 3 for 1.1 We are pre-populating a unique repository info versioned for each of these. So if a CMIS 1.0 client comes in on either of the two 1.0 bindings they will get a CMIS 1.0 compliant repository info structure. Likewise for a CMIS 1.1 client on any of the three CMIS 1.1 bindings.

The structures are the same except for the code that is contained in the block starting with this test:

```
if (cmisVersion != CmisVersion.CMIS_1_0) {
```

Please have a look at that code now. You will see these are mostly settings having to do with the new CMIS 1.1 type mutability features.

Armed with this we now should have a better idea what to do for the
  `repositoryInfo.setCmisVersionSupported(...)`
method. All we need to pass in here is the .value() of the cmisVersion object that was passed to us for this method.

Exercise 1.1 Setting the CMIS supported version

Find the line marked with the `exercise 1.1` comment. Remember FileBridge must work for both 1.0 and 1.1 clients. So we will set the version to match as the framework calls us for all of the supported versions. Place a breakpoint here and start the server in debug mode. Note the value of the cmisVersion object each of the two times this method gets called.

Exercise 1.2 Setting product, version and vendor

Find the lines marked with `exercise 1.2`. This is how we set the product, version and vendor name respectively that gets returned to the client. Change these values to something unique that you will recognize later. These values can be anything you want since they are descriptive aspects of the repository info.

Exercise 1.3 Setting the root folder ID

Find the line marked as `exercise 1.3`. To a client the root folder ID is an opaque string so in your own repository you can make this any unique value (unique for this repository) that you wish. In the case of the fileBridge we have created a static string:

```
private static final String ROOT_ID = "@root@";
```

Note that you are free to use whatever makes sense in your repository here. It may be a GUID that the underlying repository uses to identify the root folder, or it can be a special static value. It all depends on your implementation.

Choosing an ID for your root object is a nice stepping off point to the bigger subject of choosing IDs for all your objects which we will cover in the next exercise.

Exercise 2: Computing CMIS IDs for your objects

Since CMIS IDs are completely opaque to all CMIS clients, often when you build a server you have lots of choices concerning how you could map your internal objects to your external CMIS IDs. A couple of points to remember:

All IDs expressed to clients MUST be unique per repository.
There must be a strict 1:1, bi-directional ** mapping between your external IDs and your internal objects. Any ambiguity even for extreme edge cases is going to cause you pain later.
** So for example you could not use a one way hash value (not bi directional and may have collisions).
The format of your IDs should be compatible with the CMIS transports XML, JSON and the associated URLs so stay clear of special characters like < >/ \ $? * & %, etc.

With these in mind how would you go about mapping all filesystem objects to a set of unique IDs? You could use a file's iNode number as its ID. That would certainly be unique and URL friendly but it would limit us to only Unix / Linux filesystems, so let's skip that one.
We could use the fully qualified path (from the root) of the object. That would be unique for sure, but then we have the problem of all of those non URL friendly characters like spaces and '/'s not to mention all of the non-Latin characters that could be in filenames. So what we have done in this sample is base64 encode the path to get the ID and unencode to get the path. This produces IDs that are URL friendly and guaranteed to be 1:1 and bi directional. E.g. /home/test → 'aG9tZS90ZXN0' and then back again.

A note about Base 64 encoding / decoding: *OpenCMIS comes with a handy Base64 class (package org.apache.chemistry.opencmis.commons.impl) Take note of it! This class not only implements base64 in the most efficient manner possible but it also supports base64 encoded streams which you will have to deal with here on the server side.*

Spot the design problem

It turns out that there is an edge case where the model we have used for FileBridge for mapping ID's to objects violates the specification. Can you guess where that is?
Hint: A CMIS object's ID must not change even if its name changes.   This is ok for our little test server but would not be ok for a real server unless names and paths were immutable.

Exercise 2.1 Handle null and root when computing IDs

In this example we are going to be focused one small method, that being

```
private String fileToId(File file)
```

in `FileBridgeRepository`. This is the one place that gets called when the repository need to obtain a CMIS ID for a given file.

Please go there now and have a look at the code. Have a quick look over this important method. Place a break point here if you want to see what the raw paths come out looking like after they have been encoded.   Note how in the case of the root folder we simply return our constant that we discussed in exercise 1.3.

Exercise 3: Returning an Object

Almost all CMIS read operations return object data. The operations `getObject` and `getObjectByPath`, for example, return the data of a single object. The same structure is used by list of objects operations like `getChildren`, `getObjectParents`, `getCheckedOutDocs`, `getObjectRelationships`, and `getAllVersions`. The operations `getDescendants` and `getFolderTree` return a tree of the same object data structure.

In this exercise we assemble such a data structure. The metadata of an object consists of the object properties, Allowable Actions, ACLs, policies, relations, renditions, and so on. We are focusing on the properties for this exercise.

Navigate to the `FileBridgeRepository` class and find the `compileObjectData` method. It returns an ObjectData object, which the OpenCMIS server framework needs to compose a response for the client. If you look through the code, you see that this method is used in many places.

The method `compileObjectData` calls the method `compileProperties`, which you will observe in the following exercises.

Exercise 3.1 Getting the File or Folder

Navigate to the `compileProperties` method and look over the code. Before we can return something useful later in the method we need to sure that file or folder exists. If it doesn't, we need to throw the right CMIS exception.   Knowing which exception is correct is something that eventually becomes clear but for each case you can always go back to the specification (see references) and see what it says there if you don't already know.

A note about exceptions: Check the CMIS specification and the OpenCMIS JavaDoc (URL's for both in Resources section) for exception definitions. Another good starting place to look is here (http://docs.oasis-open.org/cmis/CMIS/v1.1/os/CMIS-v1.1-os.html#x1-1630001), section 2.2.1.4.1 General Exceptions from the specification itself (the source is always definitive).

Also to note:
If the object exists, then we need to identify if it is a file or folder (nothing to change for this one just be

aware what is happening). We need to know this because both types have a slightly different set of properties.

Exercise 3.2 Identify all of the Properties - Research

Consult the CMIS specification for all the document and folder properties. There are quite a few standard properties that have to be returned. Also s*ee the 'CMIS Cheatsheet' in the Resources section for a complete condensed list of the version 1.0 properties.*
The properties are of different data types. OpenCMIS provides an implementation class for each CMIS data type named `Property<type>Impl`. Find them in the OpenCMIS JavaDoc (references section).
For each property mentioned in the specification we must create a `Property` object with a suitable value. Some values are just constants. For example, the file system doesn't support versioning. So, the versioning related properties could be hard coded. Other values can be derived from the Java File object such as the name or last modification date.

Hints:
The class `PropertyIds` has constants for the property IDs and indicates when which property has been introduced. *Make sure that you don't return CMIS 1.1 properties to a CMIS 1.0 client.*
Documents and folders share a set of base properties but also have type specific properties. Make sure that you take this into account.
Make sure you treat the root folder correctly. The root folder is unique in that it has no parent folder.
OpenCMIS provides the helper class `MimeTypes` that guesses the MIME type of a file.
You can decide how you want to handle empty files. You can return them as documents without content (and no length and no MIME type) or as an empty document (with length 0 and a MIME type).

Exercise 3.3 Return the Properties

The `compileProperties` method is supposed to return an object that implements the `Properties` interface. OpenCMIS provides a simple implementation class called `PropertiesImpl`. Find the marker for `exercise 1.1`. Here we create an instance of this class and add the Property objects that you have been looking at in the previous exercise.

Once the properties collection object is created the first thing we need to do is compute the ID for the object and add it as a property using our `addProperty<type>()` method mentioned in the previous exercise. Note that the `addProperty<type>()` methods are part of FileBridge not OpenCMIS. More on that soon. Have a look at some of the other completed property types for examples on how to handle the ID.

Exercise 3.4 Honoring the Property Filter - Research

As you already know, clients may provide a property filter when requesting objects.
If you were to look up the property filter description in the CMIS specification you would find they are quite simple. They consist of a comma delimited list of property definition query names. (Go ahead and have a look just to get more familiar with the spec layout.)

Our code that added all of the properties used the constant values to identify each property we were adding. (e.g. `PropertyIds.BASE_TYPE_ID`) So we must be able to convert between those property IDs and the query names used in filters. To understand how this is done, have a look at one of the `addProperty<type>()` methods to see how it calls `checkAddProperty()` to see if the property should be added to the response. Next look at the `checkAddProperty()` method to see how this is implemented. Pay special attention here to how the `TypeManager` is used to convert from the typeId to the Query name so that it can be compared to the filter.

Exercise 4: getContentStream

Now that we have gotten this far things are going to start looking pretty simple. For this exercise examine the implementation of `getContentStream`; and in doing so, see some details on how to handle Offset + Range even if your underlying repository does not support them. The illustration below shows what the top of the stack will look like after a call to `getContentStream`.

Exercise 4.1 Offset and Range

Navigate over to the

```
    public ContentStream getContentStream
```

method in the `FileBridgeRepository` class and set a breakpoint here. Then retrieve a document in Workbench to catch it.
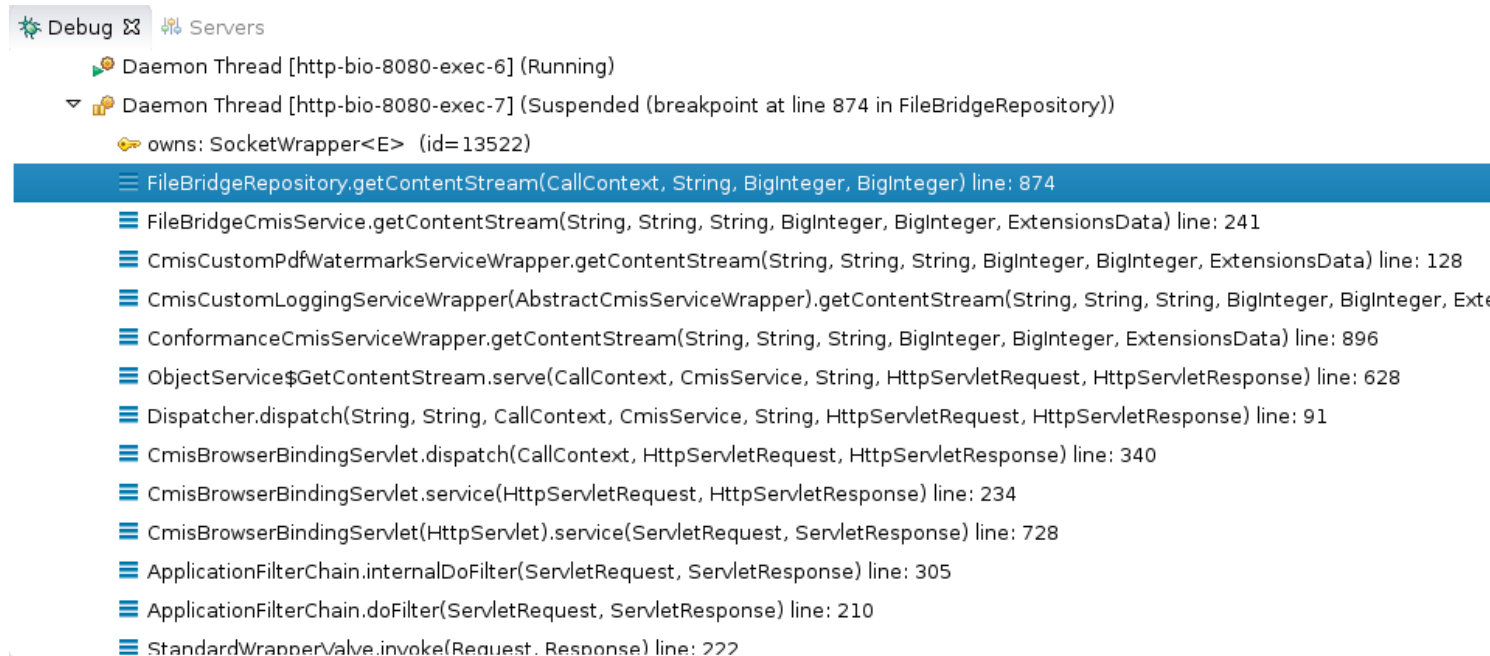
Those of you that are really paying attention have noticed something odd here. Where is that call to `CmisCustomPdfWatermarkServiceWrapper` *coming from?   We are going to cover that in part 2 of this guide. For now just know it's supposed to be here.*

Go ahead and step through the method paying attention to how we would handle offset and range if they were present. They won't be unless you have built your own client to test this.   Workbench just defaults to retrieving the entire file.

Extra Credit: Construct a groovy script to run inside of Workbench that will retrieve a test text document using the range and offset parameters to see if our implementation really works.

After inspection you will see that a lot of the work is done in this line

```
stream = new BufferedInputStream(new FileInputStream(file), 64 * 1024);
```

where we initialize a `BufferedInputStream` with the stream from our local file.

Exercise 5: Adding logging and tracing to your server

Logging is not in the scope of the OpenCMIS server framework so you may use whatever technology or library you prefer for this purpose. Note that the framework uses SLF4J (http://www.slf4j.org/) for logging internally. So if you want framework log messages in your log files, you may want to configure SLF4J to use your same logging backend.
In this exercise we are going to show you how to do just that.

Unlike the solution project, the lab version of your project represents a generically generated project from Maven. In order to add logging we are going to have to modify that project in the following ways:
Modify the `pom.xml` to include whatever dependencies we need for our logging framework of choice.
Add a `log4j.properties` file.

In order to add tracing, we have manually added a new `src/main/webapp/WEB-INF/web.xml` file to define our http trace logging filter. Note that HTTP tracing and logging are not related or dependent on each other. You can use one or the other, both or neither. To emphasize this we are separating them into different sections of this exercise.

Exercise 5.1 Adding slf4j to our project for logging

In our sample FileBridge server we have decided to use SLF4J as our abstraction logging layer for our code and at deployment time we will be configuring log4j as our logging implementation. To make this change in our project we will have already modified the project's `pom.xml` as follows: (right before the closing `</dependencies>`)

```
<!-- enables OpenCMIS frame logging to log4j -->
<dependency>
      <groupId>org.slf4j</groupId>
      <artifactId>slf4j-log4j12</artifactId>
      <version>1.7.5</version>
</dependency>
```

Please go to your `pom.xml` now and verify you have this change.

Next we have added the required log4j.properties file.
Locate `log4j.properties` in your solution project's directory and quickly inspect the contents.
The file is located here:

```
/src/main/webapp/WEB-INF/classes
```

At this point we will are ready to add some logging code in the next exercise.

For more information about how slf4j can dynamically plug in logging frameworks at deployment time please consult the documentation at http://www.slf4j.org/.

Remember that anytime you change the `pom.xml` you must remember to do a (right click on the server project) / maven / update project
to have Maven make all of the necessary changes to your Eclipse project. After selecting this option you will see the dialog in the illustration below:
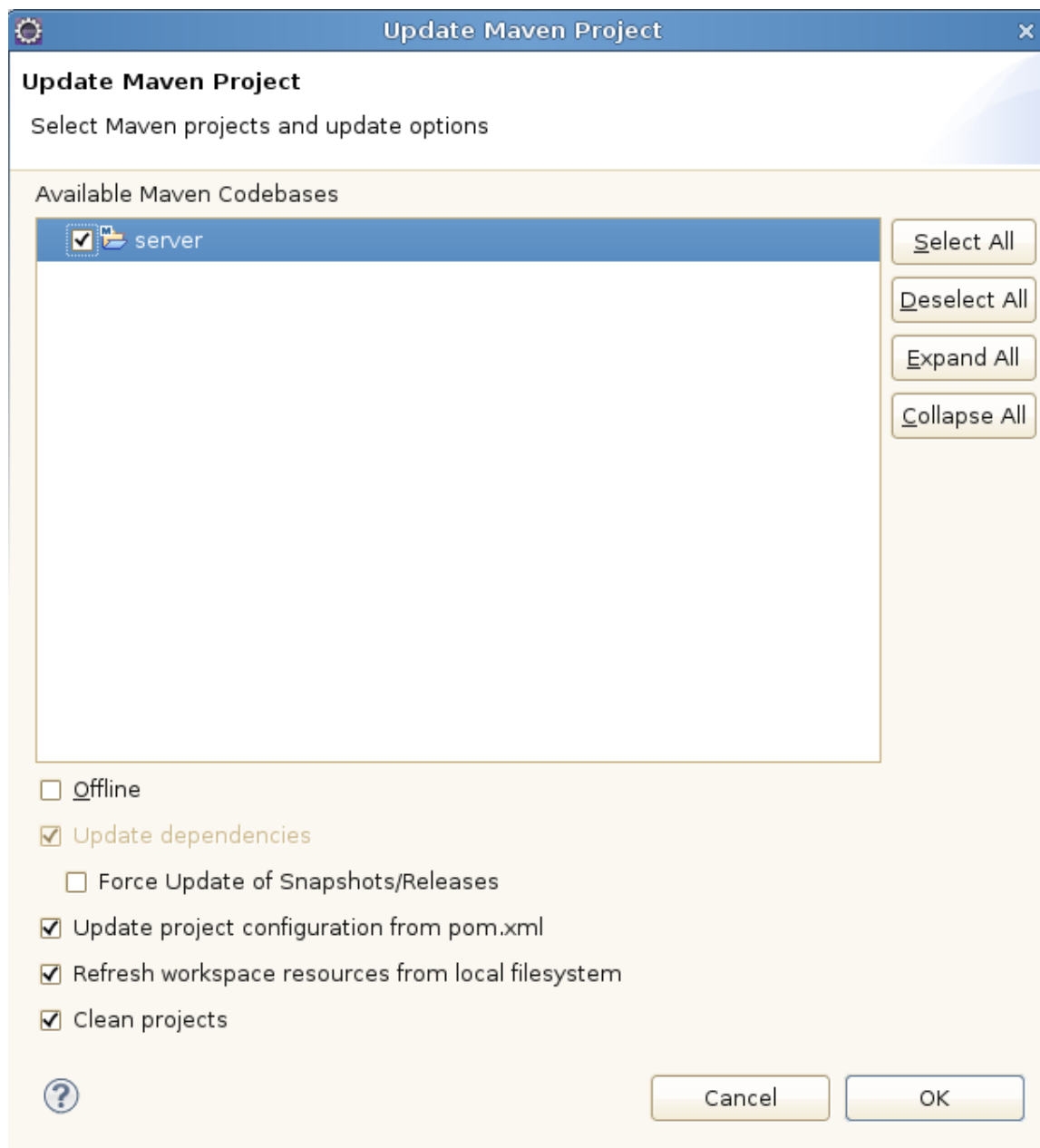
Select 'OK' to continue. After Maven has finished processing you can check the libraries view of your project (properties / Java Build Path / libraries / maven dependencies) and you will now see the project has been updated with any new dependencies (like log4j and slf4j jars associated).
(verify this now in your own project)

Exercise 5.2 Adding some logging code

Note that the final version of this code already has a server runtime plugin configured that does logging for a test method. So when you do this exercise you will also see logging output from that code as well for any `getFolderChildren` *calls. You can ignore this other logging output for now.*

Now that all of the dependencies are setup we are going to add a log entry for every time the framework calls the getRepositoryInfos() method on our service.

In `FileBridgeCmisService` you will need do add the following imports:

```
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
```

Then at the top of the class anywhere between

```
public class FileBridgeCmisService extends AbstractCmisService {
```

and the first method definition we will add

```
    private static final Logger logger =
LoggerFactory.getLogger(FileBridgeCmisService.class);
```

then on the first line of `getRepositoryInfos` we will add the logger (info) line:

```
logger.info("entering call to FileBridgeCmisService.getRepositoryInfos");
```

That's all there it to it! We are ready to see it run.

Exercise 5.3 Observe the logging output

Run your server and login with CMIS Workbench as usual. At this point you should see some output in your Eclipse console window like this:

```
2013-09-13 19:15:55,728 INFO  [http-bio-8080-exec-3]
org.example.cmis.server.FileBridgeCmisService: entering call to
FileBridgeCmisService.getRepositoryInfos
```

At this point if you wish to see how this all works in a normal Tomcat container you may want to build from the command line to produce a war file in the /target directory.
Recall how we did this earlier in the lab by running

```
mvn clean install -Dmaven.test.skip=true
```

at the top of our project directory (where the `pom.xml` is located).
Rename the war file (to whatever you wish, e.g. `filebridge.war`) and deploy to Tomcat as usual.

After running again you should see your logging output in the `catalina.out` file.

Armed with the knowledge of how to add logging, we should be ready to start running some unit tests in the next exercise.

Exercise 5.4 Overwriting the web.xml file to enable HTTP tracing

In this exercise we are going to enable the
`org.apache.chemistry.opencmis.server.support.filter.LoggingFilter`
HTTP tracing filter. In order to do so we will have to make some changes to the `web.xml` for the servlet.

Note: A generic OpenCMIS server gets its web.xml file from the framework so it will not normally appear in your Eclipse project. But since we need to make changes to it we have already added one and in the process overrode the default framework version. As you must do in your own projects.

Go to your solution project's `src/main/webapp/WEB-INF/` and open up the `web.xml` file. Once there uncomment out the following xml:

```xml
 <filter>
        <filter-name>LoggingFilter</filter-name>
        <filter-class>org.apache.chemistry.opencmis.server.support.filter.LoggingFilter</filter-class>
        <init-param>
            <param-name>LogDir</param-name>
            <param-value>/home</param-value>
        </init-param>
        <init-param>
            <param-name>PrettyPrint</param-name>
            <param-value>true</param-value>
        </init-param>
        <init-param>
            <param-name>LogHeader</param-name>
            <param-value>true</param-value>
        </init-param>
        <init-param>
            <param-name>Indent</param-name>
            <param-value>4</param-value>
        </init-param>
    </filter>

    <filter-mapping>
        <filter-name>LoggingFilter</filter-name>
        <servlet-name>cmisatom10</servlet-name>
    </filter-mapping>
    <filter-mapping>
        <filter-name>LoggingFilter</filter-name>
        <servlet-name>cmisatom11</servlet-name>
    </filter-mapping>
    <filter-mapping>
        <filter-name>LoggingFilter</filter-name>
        <servlet-name>cmisws10</servlet-name>
    </filter-mapping>
    <filter-mapping>
        <filter-name>LoggingFilter</filter-name>
        <servlet-name>cmisws11</servlet-name>
    </filter-mapping>
    <filter-mapping>
        <filter-name>LoggingFilter</filter-name>
        <servlet-name>cmisbrowser</servlet-name>
```

```
</filter-mapping>
```

Note the value of `LogDir` must be set to a valid directory. For the tutorial image we are using `/home`. So don't forget to set this value after you uncomment the xml. If you don't the output will go to your temp directory. e.g. `/tmp`.

Note about performance and tracing:
Use this tracing filter with care! It can generate a huge amount of files (as you will see) and will slow down performance significantly.

The screen shot in the illustration below is of the home directory after a CMIS Workbench login has taken place. As you can see there are a total of 8 round trips made in the process of getting the service document, retrieving the type definitions, getting the root folder children, etc.
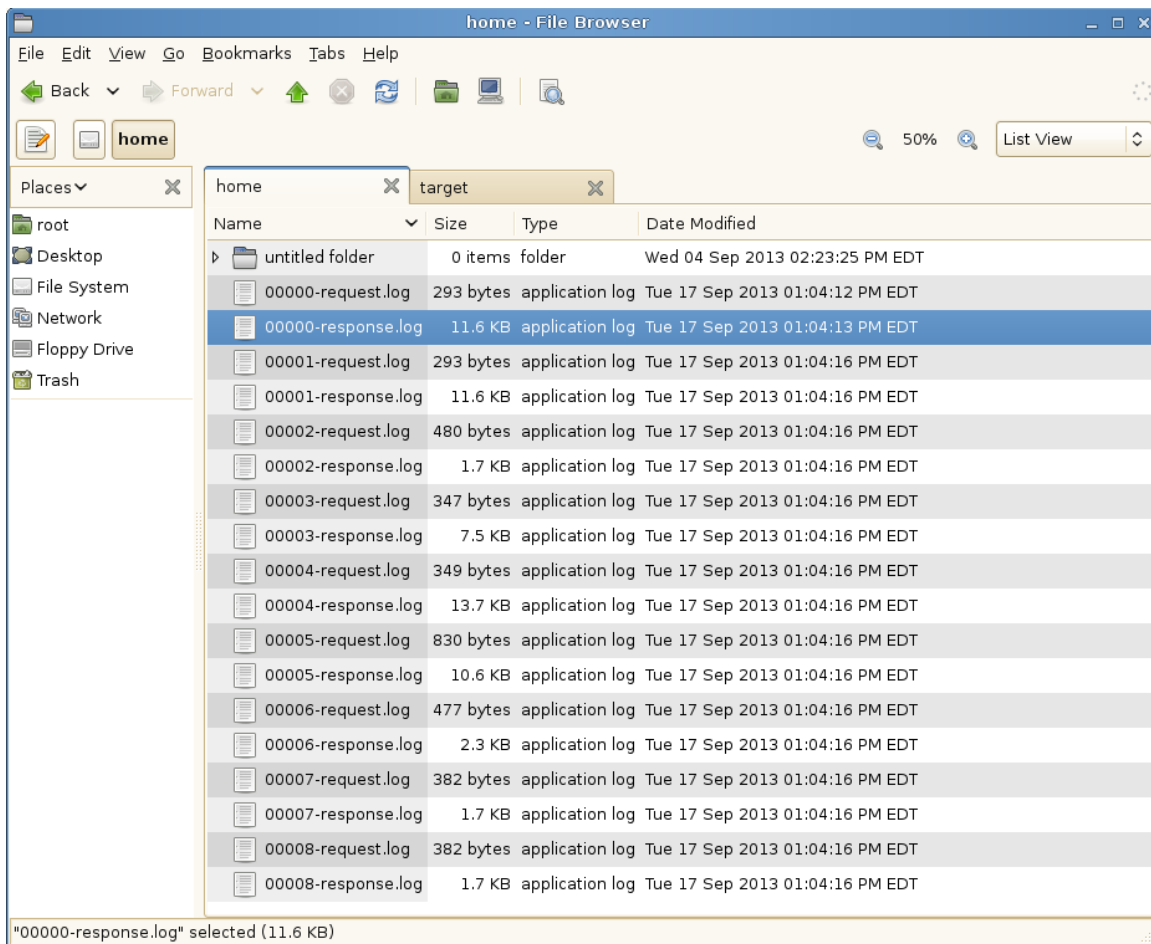
Next we will open up a couple of these and see what a typical browser trace looks like.

Examine the HTTP trace output

We are going to look as one of these requests so you can see what a typical trace looks like. Request 002 is a request to get the root folder object shown here:

```
GET /server/browser/test/root?objectId=%40root
%40&cmisselector=object&includeAllowableActions=true&includeRelationships=none&rend
itionFilter=cmis%3Anone&includePolicyIds=false&includeACL=false&succinct=true
HTTP/1.1
User-agent: Apache Chemistry OpenCMIS/1.0.0-SNAPSHOT
Authorization: Basic dGVzdDp0ZXN0
Accept-encoding: gzip,deflate
Cache-control: no-cache
Pragma: no-cache
Host: localhost:8080
Accept: text/html, image/gif, image/jpeg, *; q=.2, */*; q=.2
Connection: keep-alive
```

The response contains all of the properties for the folder formatted as JSON since we are using the browser binding. Here is the full text of the response:

```
HTTP/1.1 200
Content-Type: application/json;charset=UTF-8
Server: Apache-Chemistry-OpenCMIS/1.0.0-SNAPSHOT
Cache-Control: private, max-age=0

{
    "succinctProperties":{
        "cmis:objectId":"@root@",
        "cmis:name":"",
        "cmis:createdBy":"<unknown>",
        "cmis:lastModifiedBy":"<unknown>",
        "cmis:creationDate":1379287941000,
        "cmis:lastModificationDate":1379287941000,
        "cmis:changeToken":null,
        "cmis:description":null,
        "cmis:secondaryObjectTypeIds":null,
        "cmis:baseTypeId":"cmis:folder",
        "cmis:objectTypeId":"cmis:folder",
        "cmis:path":"\/",
        "cmis:parentId":null,
        "cmis:allowedChildObjectTypeIds":null
    },
    "allowableActions":{
        "canDeleteObject":false,
        "canUpdateProperties":true,
        "canGetFolderTree":true,
        "canGetProperties":true,
        "canGetObjectRelationships":false,
        "canGetObjectParents":false,
        "canGetFolderParent":false,
        "canGetDescendants":true,
```

```
        "canMoveObject":false,
        "canDeleteContentStream":false,
        "canCheckOut":false,
        "canCancelCheckOut":false,
        "canCheckIn":false,
        "canSetContentStream":false,
        "canGetAllVersions":false,
        "canAddObjectToFolder":false,
        "canRemoveObjectFromFolder":false,
        "canGetContentStream":false,
        "canApplyPolicy":false,
        "canGetAppliedPolicies":false,
        "canRemovePolicy":false,
        "canGetChildren":true,
        "canCreateDocument":true,
        "canCreateFolder":true,
        "canCreateRelationship":false,
        "canCreateItem":false,
        "canDeleteTree":true,
        "canGetRenditions":false,
        "canGetACL":true,
        "canApplyACL":false
    }
}
```

As you can imagine. Having this level of tracing detail can prove invaluable when you are doing interoperability tests with 3rd party CMIS clients.

Exercise 6: Testing your CMIS server

The objective of a CMIS server is to be able to serve any CMIS client. Compliance with the CMIS specification is key. To make this compliance easier, OpenCMIS provides a test utility called the Test Compatibility Kit (TCK), which makes a few hundred calls to the repository and checks if the repository reacts as defined in the CMIS specification. It covers most areas of the specification and is an essential testing tool. It cannot replace repository specific tests, though.

In this exercise we run the OpenCMIS TCK against the final version of the FileBridge Repository. The TCK is a library and can be triggered in many different ways. There is, for example, an Ant task to run the TCK and all TCK test are also JUnit tests. You can build your own TCK runner or just use the CMIS Workbench. In this exercise we do the latter.

Open the CMIS Workbench and connect to the FileBridge server. When you press the TCK button in the Workbench toolbar, a dialog should open that lets you pick the tests you want to run (shown in the illustration below)
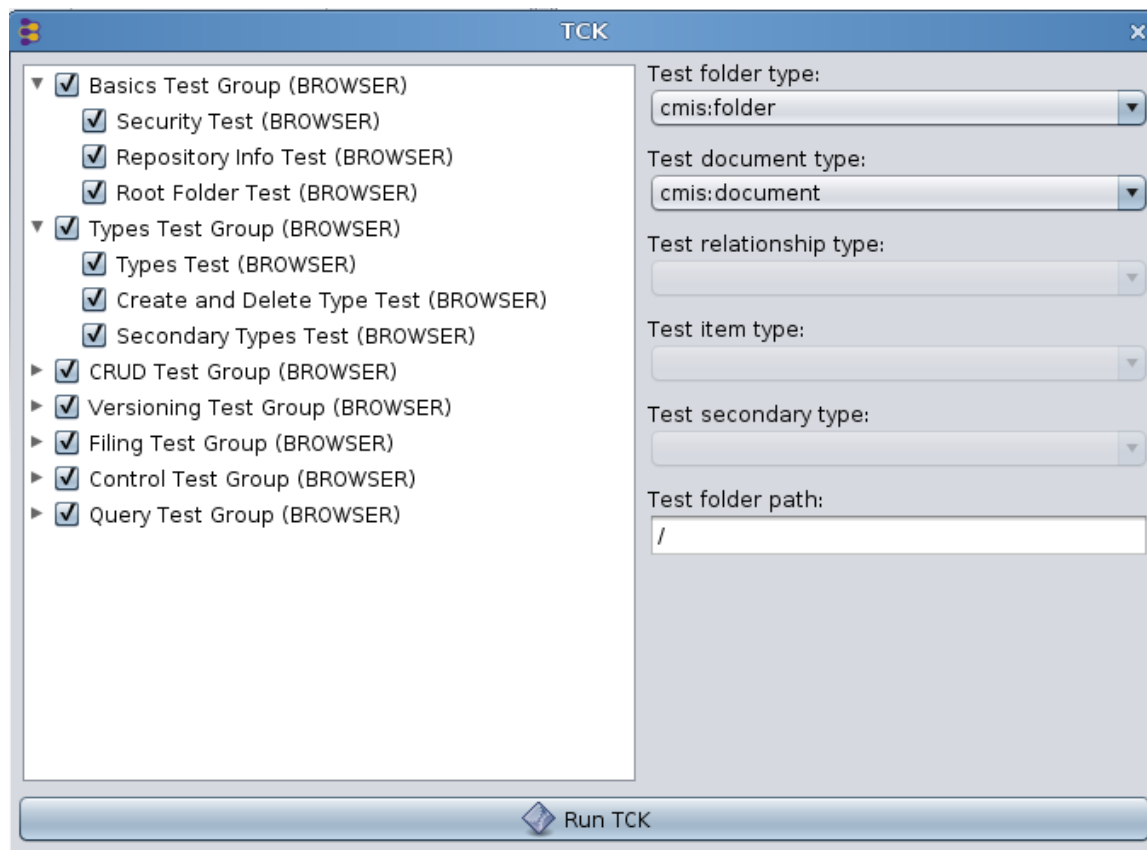
By default all tests are enabled.

Now run the TCK and wait for the results.

When the TCK run is done, a new window opens with the results. If you need more details, open the HTML report (at the bottom of the window). It contains more information and links to the test code. The illustration below shows the test report output dialog:
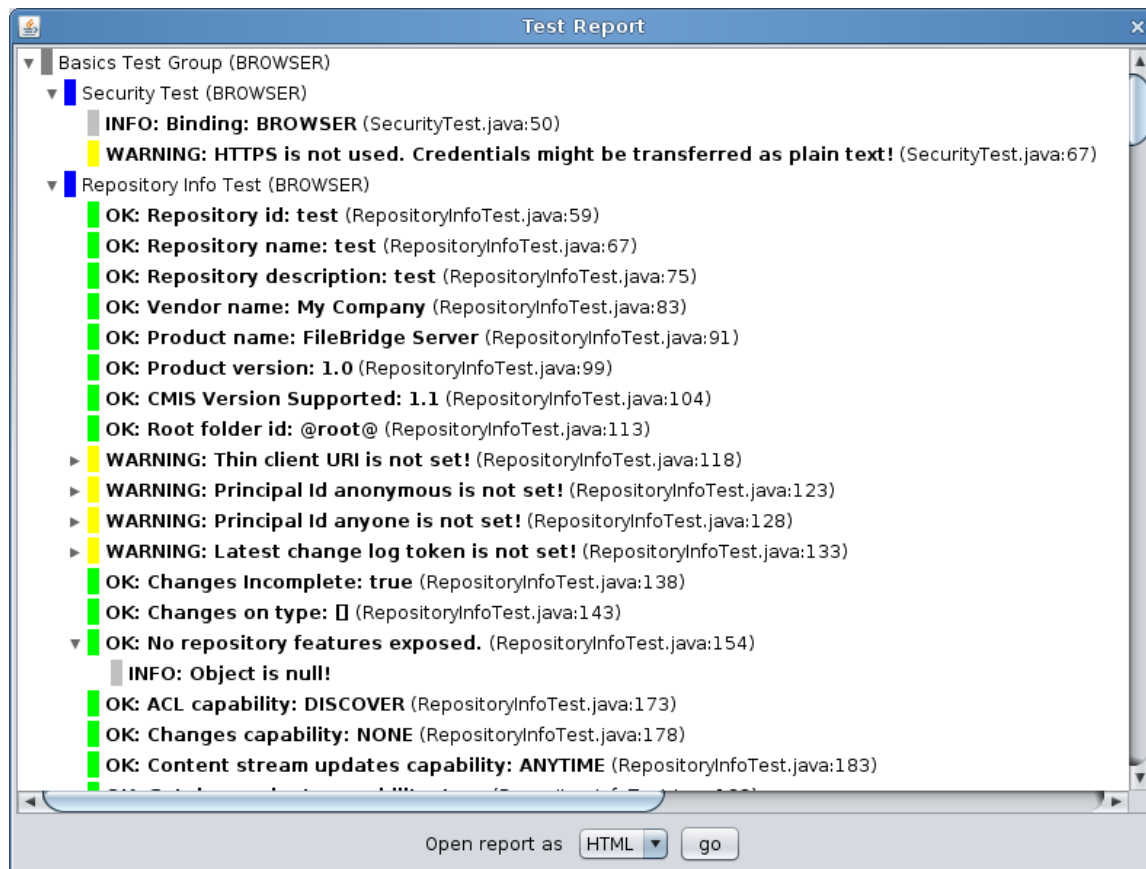
A TCK report can contain six different message types.

- **INFO**: Additional information about the test result.
- **SKIPPED**: The repository doesn't support the feature and the test was skipped.
- **OK**: The test was successful.
- **WARNING**: The test was somewhat successful. The repository didn't violate the specification, but there may be interoperability issues. You should analyze the cause of this warning and decide if you can ignore it.
- **FAILURE**: The TCK noticed a specification violation. There is something to fix.
- **UNEXPECTED EXCEPTION**: The TCK received an unexpected exception. There is probably a bug in the repository implementation.

Go through the report and check all warnings and failures. Why do you think the FileBridge server has failures?

Exercise 7: Supporting multiple repositories for your service

For this last exercise we have a look at the code that reads and parses the `repository.properties` file. Please navigate to the `FileBridgeCmisServiceFactory` class and then to the

private void readConfiguration(...)

method. Here take a moment to see how we are currently parsing the `.properties` file to get a list of the repositories that we will expose from our `getRepositories` implementation. Of course, this is entirely arbitrary how we are doing this, the important thing to note here is that we are maintaining a list of the repositories and their IDs. Even though your native service may only have one repository that does not mean that you cannot synthesize additional ones. For example you would have one repository that serves up the documents, and one that serves up the same documents with with watermarks.
After you have absorbed what is going on here (in this case the code is self explanatory), open up the `repository.properties` file for editing.
Recall the location of this file is shown in the illustration for the "A Note about Running From Windows" section.
(also note that this example will be for Linux. If you are using Windows please modify the file paths to something with a local drive letter and path as we did earlier.

There is already a section marked as exercise 7. If you uncomment that section you will have something like this:
Note you can ignore the stuff at the tail end of the file concerning server plugins for now. We will cover that in part 2.

```
class=org.example.cmis.server.FileBridgeCmisServiceFactory

login.1 = test:test
login.2 = reader:reader

repository.test = /
repository.test.readwrite = test
repository.test.readonly = reader

repository.test2 = /home
repository.test2.readwrite = test
repository.test2.readonly = reader
```

Then start your server and reconnect with Workbench. Once you have done a load repositories, you should see your new 'test2' repository as shown in the illustration below.
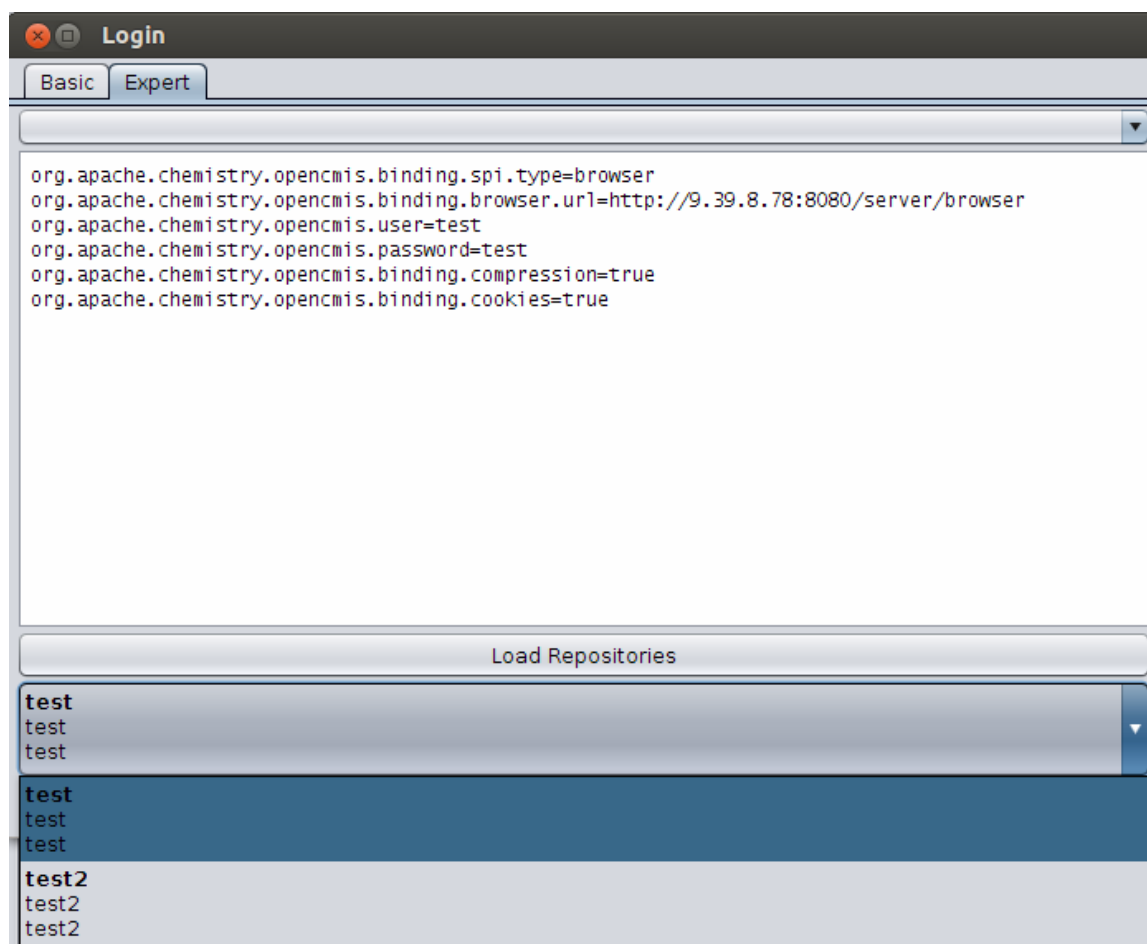
Another example would be to have the code dynamically discover all of the mounted filesystems (e.g. network drives) (for Windows use each drive letter) and expose each as its own repository (left as an exercise for the student).

Miscellany for Developers

IBM Content Navigator's CMIS client - minimum requirements

Different CMIS clients have different requirements. For example some may only work with with the AtomPub binding. While others may require support for Query.
To be fully functional with IBM Content Navigator's CMIS client, a CMIS Server must meet the following requirements:

- CMIS 1.0 compliant
- expose the optional capabilityQuery = metadataOnly (or greater)
  This is used to more efficiently get document folder children separately from folder children for the different pane displays.

Other than the extra query capability there are no other 'optional' features needed. In the book "CMIS and Apache Chemistry in Action", (see references section) chapter 14 covers the subject of building a query parser which we will not be able to cover in this short tutorial.
Instead for this tutorial we have added a tiny bit of code to fileBridge so that is can handle two specific queries namely:

```
    SELECT * from cmis:document WHERE in_folder('xx')
```
and
```
    SELECT * from cmis:folder WHERE in_folder('xx')
```

e.g. `SELECT * FROM cmis:folder WHERE in_folder('@root@')`
With this tiny bit of query code added (see FileBridgeRepository.query()) fileBridge is compatible with Navigator. See illustration below showing Navigator 2.0.2 browsing our local filesystem:
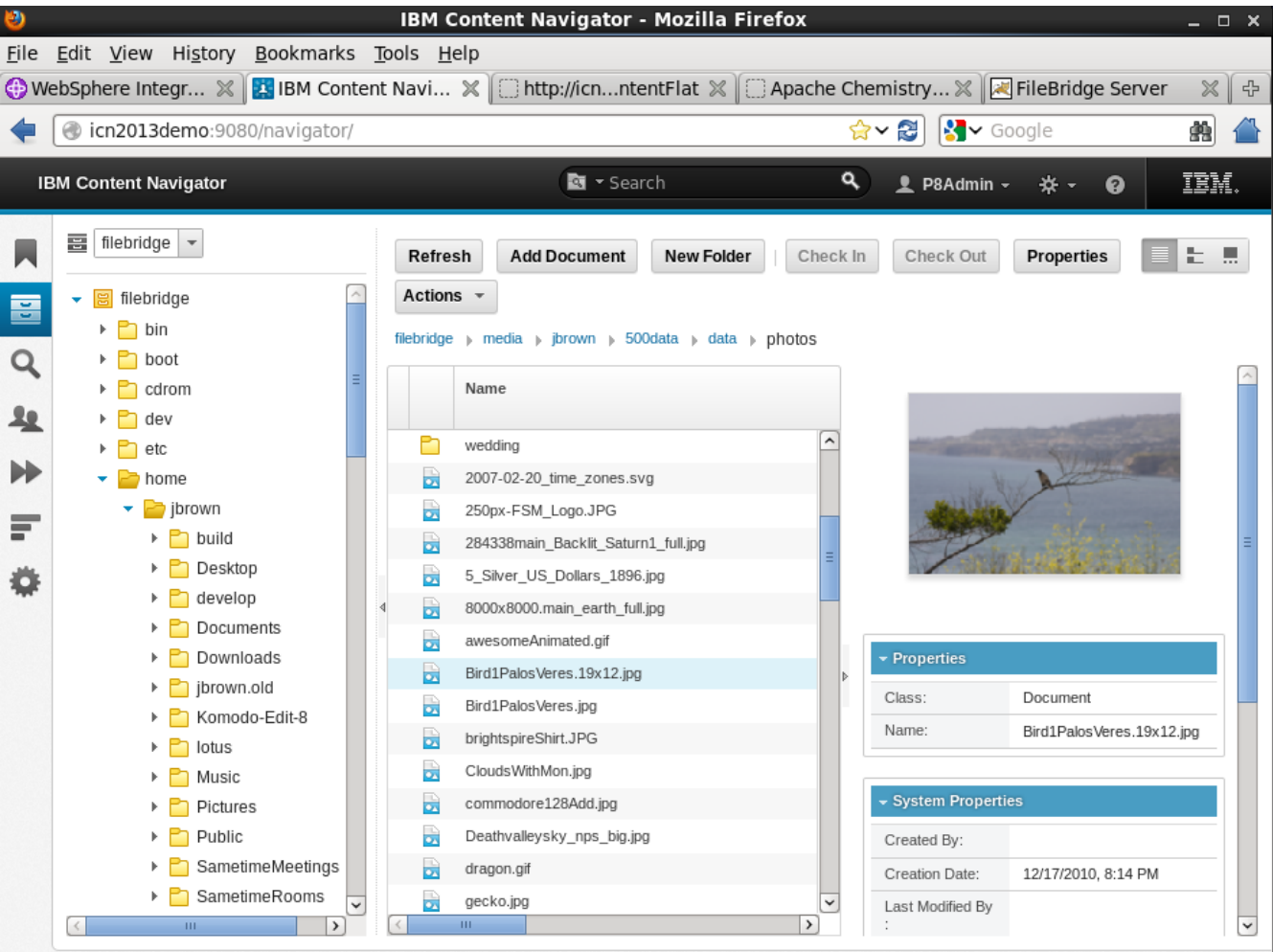
Illustration 5: IBM Content Navigator browsing our demo fileBridge repository.

Subversion clients for Windows

(GUI) TortoiseSVN - http://sourceforge.net/projects/tortoisesvn/
(command line) – These are listed in the Windows section toward the bottom of
http://subversion.apache.org/packages.html

Auto start CMIS Workbench connected to your server

If you would like your Workbench to start up every time connected to the same server with the same credentials. (As is often the case in developer environments)

Create a shell file to kick off Workbench along these lines (linux version):

```
#! /bin/sh
export CUSTOM_JAVA_OPTS="\
-Dcmis.workbench.binding=browser \
-Dcmis.workbench.url=http://localhost:8080/server/browser \
-Dcmis.workbench.user=reader \
-Dcmis.workbench.password=reader \
-Dcmis.workbench.compression=true \
-Dcmis.workbench.cookies=true"

# Now cd to your directory where you have Workbench
# then run the stock workbench.sh script.
./workbench.sh
```

Conclusion for Part I

This concludes this document. In this tutorial you have become familiar with the following:
* Building an OpenCMIS server and client library dependencies from the source.
* Setting up a typical OpenCMIS server build environment using Eclipse and Maven.
* Familiarity with the Chemistry Workbench CMIS client.
* Understanding key points necessary to build an OpenCMIS server with enough functionality to be browsable by the CMIS Workbench client (or the IBM Navigator Client).
* Understanding how to add additional library dependencies to your server as in the case for a logging framework.
* Understand how to hook up HTTP tracing for debugging and interoperability testing.

Part 2 – The OpenCMIS Server Plugins Framework

Part 2 will introduce you to the concept of runtime OpenCMIS server plugins. We will then show you how to update a pre 0.11.0 OpenCMIS server so that it will accept any of these plugins at runtime. Finally we will walk you through two sample plugin examples that you can add to any CMIS server that supports this new feature. This part is written as a brief intro into the new plugin framework for someone who is already familiar with the concepts of OpenCMIS server development.

Part 2 includes two main subparts:
- Server side considerations: Covers the high level design of the framework and changes needed to make your server support the new plugins. This includes a new version of FileBridge with plugins and a guided tour through the changes needed.
- Building a Server Plugin: This includes 2 simple reference plugin projects along with instructions for deploying them to your FileBridge (or any OpenCMIS) server.

What are Server Plugins?

OpenCMIS server plugins allow consumers of CMIS servers (customers and third parties) to treat a CMIS server as an open service platform they can extend and change as needed. Plugins are the OpenCMIS version of a feature that started out experimentally in IBM's line of CMIS servers in 2014. Note: *These are not to be confused with 'Repository Features' from section 1.5.2 of the CMIS 1.1 specification.*

How plugins came about

IBM was getting requests for custom CMIS features that would only be useful for a single customer or a single ECM related vertical. They needed a way for customers and partners to independently develop their own custom add-ons and plug them in at deploy time.
Sometimes these add-ons would be very simple, like a specialized audit log that needs to be created for certain CMIS calls. Others would add in entire optional parts of CMIS on top of the off the shelf version. For example a need to plug custom cmis:policy behaviors, rendition implementations or even integrate certain operations with business processes. Alternatively, instead of adding features some may want to tweak an existing behavior slightly. For example, a plugin that watermarks any PDF files that are retrieved for certain types of users, but leaves all the metadata unchanged. (This is one that we chose for one of our working examples.)

Supported versions of OpenCMIS

OpenCMIS server plugins are supported on versions 0.11.0 and later of the OpenCMIS server framework. If you need features from an unreleased version of the framework, you can grab the 1.0.0-snapshot and build it locally.

Engineering requirements

To ease the tasks of development and deployment we defined the following additional requirements:

- Plugins must be uncoupled from the server.
- A customer or partner can develop independent plugins and plug them into any (current) OpenCMIS based server at deploy time.
- A CMIS server will not need to know whether it has plugins installed (chaining must be handled entirely by the shared framework).
- The only dependencies on these plugins should be libraries already in use from Apache Chemistry (OpenCMIS). Nothing from a specific vendor required, unless the plugin is using a native library for its function.
- Developing these plugins should be like building a normal server so that there is no (additional) knowledge needed to make them for a developer already familiar with the OpenCMIS server framework.
- A CMIS server should support an unlimited number of plugins chained in the order that the customer specifies at deploy time.

The image below shows the typical lifecycle of these plugins at a customer site.
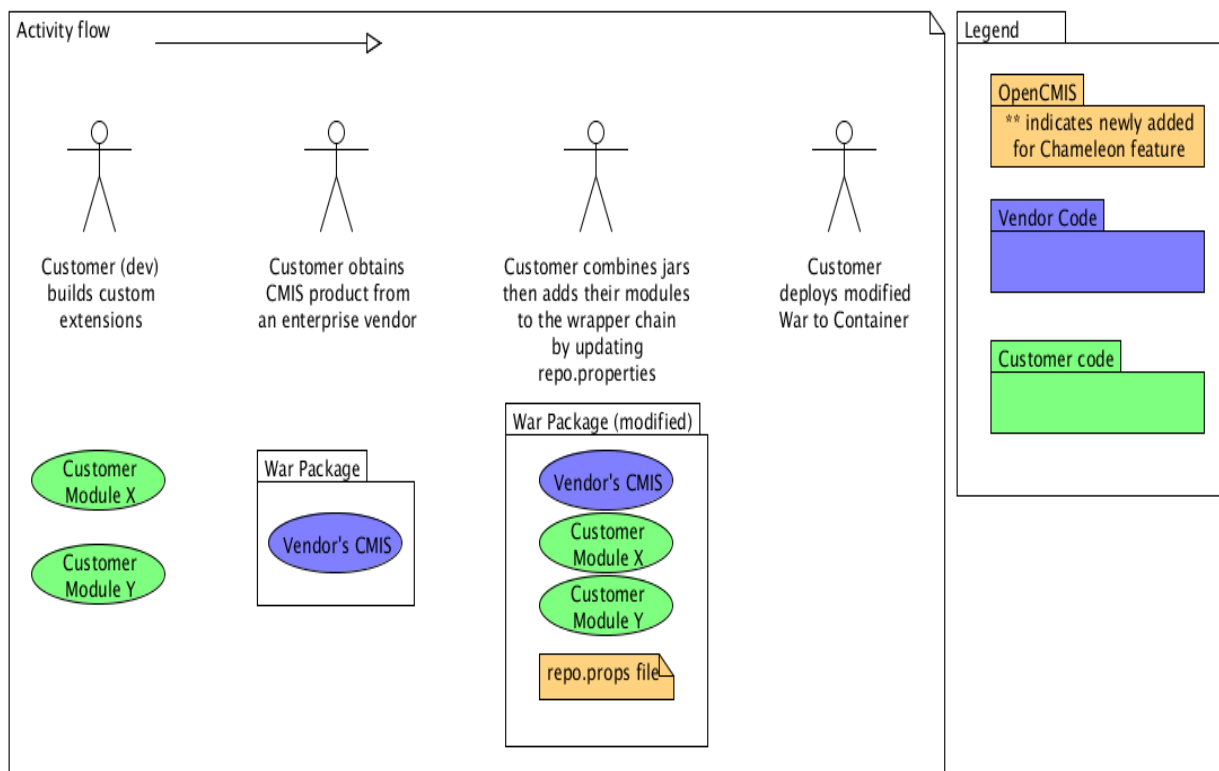
Note that in many cases the same custom plugin can be deployed to any number of vendor's CMIS implementations assuming it is not doing something vendor specific. For example, a watermarking, or detailed logging plugin would be vendor neutral.

Design and Discussion

The illustration below shows the overall design of the new server plugins broken down to three color categories. Blue showing the parts that the vendors ships to customers. Gold is the part that is from the OpenCMIS libraries, and the green is the part supplied by a consumer of the CMIS service.
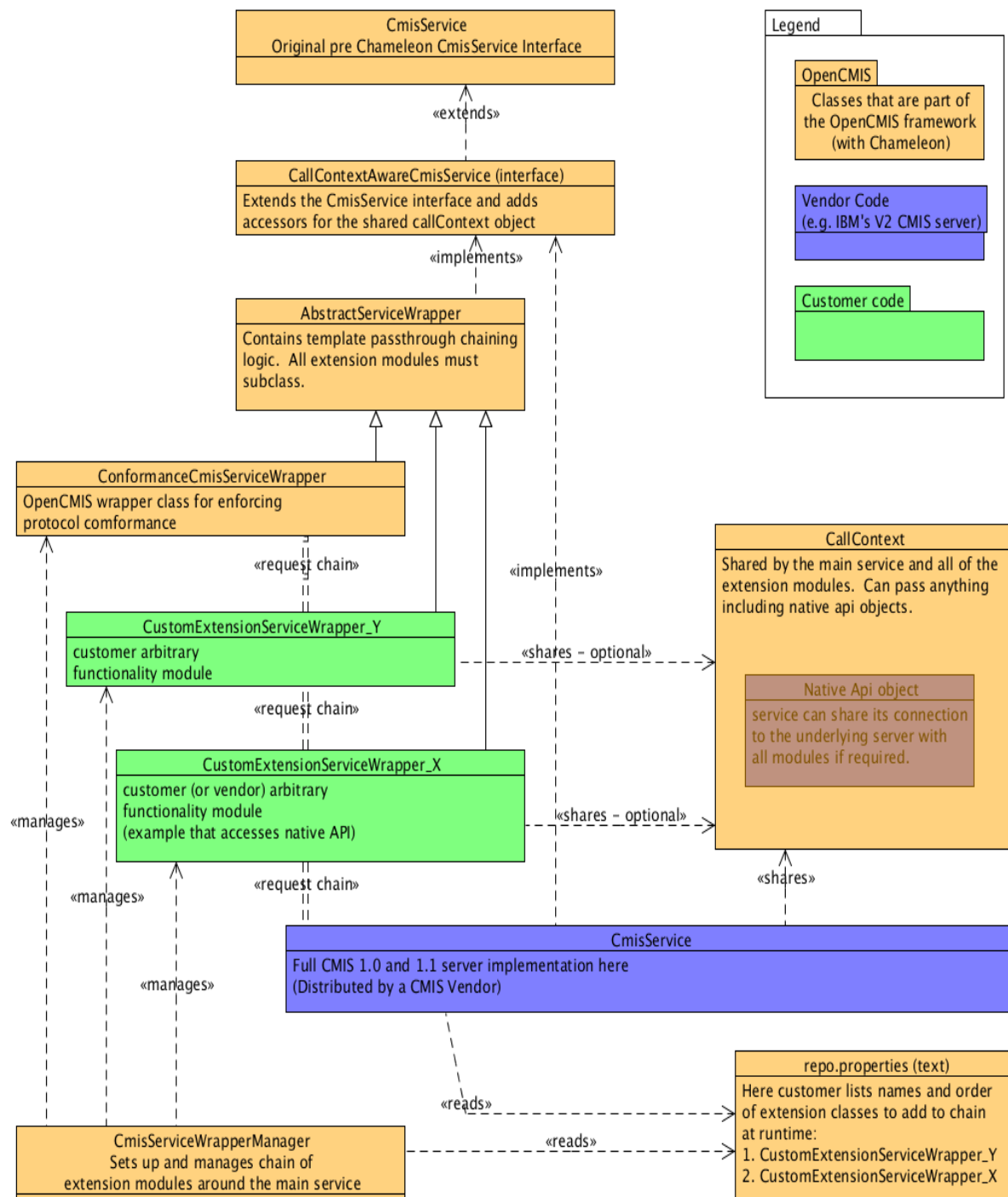
As you can see from the diagram above. Plugins (shown in green – xxxWrapper_...) end up implementing (via their extension of the `AbstraceServiceWrapper`) the same `CmisService` interface as all OpenCMIS servers. Any function they do not override will be handled by another extension further down the chain and eventually by the main CMIS service itself (shown in blue). The

`CmisServiceWrapper` handles reading in the registered plugins from the repository.properties file and sets up the service chain between them.   If the main `CmisService` adds any objects to the `CallContext` either in its `ServiceFactory` or while processing requests, all of the plugins on the request chain will be able to have access to that object. A common use case for this is as follows: The main CMIS service has already setup a connection with the underlying repository. If it shares that connection with the `CallContext` then plugins can share that (already authenticated for the call) connection to improve performance of the plugins. Any object can be shared in the `CallContext` including objects that are shared between plugins.   Just remember that the order that the plugins are registered will prevent one plugin from seeing objects added from another added later in the chain. And the order is reversed between the requests and responses.

ServerSide Changes to enable plugins

This section will walk you through the small adjustments you need to make to your current OpenCMIS based server (in this case FileBridge but the changes are the same in general).   We encourage you to follow along in the latest version of the FileBridge Server so see these changes with more context.

Changes to CmisService implementation (FileBridgeCmisService)

The changes to your service amount to a modification of one line (and the corresponding import). Where previously your service would extend the AbstractService like this:

```
public class FileBridgeCmisService extends AbstractCmisService
```

You will now add an `implements` for the `CallContextAwareCmisService` interface like this:

```
public class FileBridgeCmisService extends AbstractCmisService
implements CallContextAwareCmisService
```

which opens up the ability to get and set the internal `CallContext` object.   Those accessor functions are already present, so nothing else to do here.

Changes to your ServiceFactory (FileBridgeCmisServiceFactory)

The WrapperManager

As we have discussed in the intro section, the `CmisServiceWrapperManager` is the class that manages the chain of plugins so it makes sense that we would have to initialize it in the service factory. First off we will declare it as a private like this:

```
private CmisServiceWrapperManager wrapperManager;
```

Then we will allocate it, and initialize it in our `init()` method like this:

```
wrapperManager = new CmisServiceWrapperManager();
wrapperManager.addWrappersFromServiceFactoryParameters(parameters);
wrapperManager.addOuterWrapper(ConformanceCmisServiceWrapper.class,
     DEFAULT_MAX_ITEMS_TYPES,
     DEFAULT_DEPTH_TYPES,
     DEFAULT_MAX_ITEMS_OBJECTS,
     DEFAULT_DEPTH_OBJECTS);
```

The last line (in bold) sets up the `ConformanceCmisServiceWrapper`, which is a replacement of the old way we did this in `getService` where we would allocate and initialize a new `CmisServiceWrapper` and hand it our newly created service object.

In our `getService` method the main difference is that now that we have the new `wrapperManager`, we give it our newly created service implementation so that it can be added into the bottom of the chain of plugins (if there are any found at runtime).   In the case of the `fileShareService` the code looks like this:

```
service =
(CallContextAwareCmisService)wrapperManager.wrap(fileShareService);
threadLocalService.set(service);
```

The last line passes the fully setup wrapper manager to our

```
ThreadLocal<CallContextAwareCmisService>
```

Which is defined in place of the old

```
ThreadLocal<CmisServiceWrapper<service class name here>>
```

declaration.

That's all there is. All the rest of your server code will continue to work the same and it will be unaware if there are any plugins running on top.

Setting a custom CallContext (optional)

If your Service implementation needs to be able to share objects or data with plugins, they can be transported by a custom call context object.

For example: Imagine your service wants to share the connection it has to an underlying database or back end server so that plugins will not have to establish one of their own.

The `CallContext` interface does not provide means to add, change, or remove data. You can cast `CallContext` objects provided by the server framework to the `MutableCallContext` interface, which exposes `put()` and `remove()` methods in addition `get()` method already present in the parent. However, casting can fail if your implementation is used outside the server framework, for example in a test framework that provides dummy `CallContext` objects. It will also fail if you want add or change values within a service wrapper for subsequent wrappers in the wrapper chain and a previous service wrapper in the chain doesn't forward the original `CallContext` object but an immutable one.

A cleaner way is to create a custom `CallContext` class. OpenCMIS provides the `CallContextWrapper` class, which makes creating a custom context easy. The `FileBridgeCallContext` is a simple example. The key is override the `get()` method and add the logic for your own values. Additionally, you can also provide convenience and type-safe methods, if you want.

Building a Server Plugin – Example #1 - Logging

Note: For the exercises in part 2 you should be using the `reader` account (password = `reader`).

Under normal conditions you would be building a server plugin separately then adding its jars at runtime to the target CMIS server. However in this tutorial we are including the two plugins as part of our project so that you don't have to manually deploy them every time you want to test a new code change.

Below you will see a screenshot of the project explorer for our server with the `org.foo` namespace highlighted. These two files are the two server plugins that we will be building in the remaining exercises.
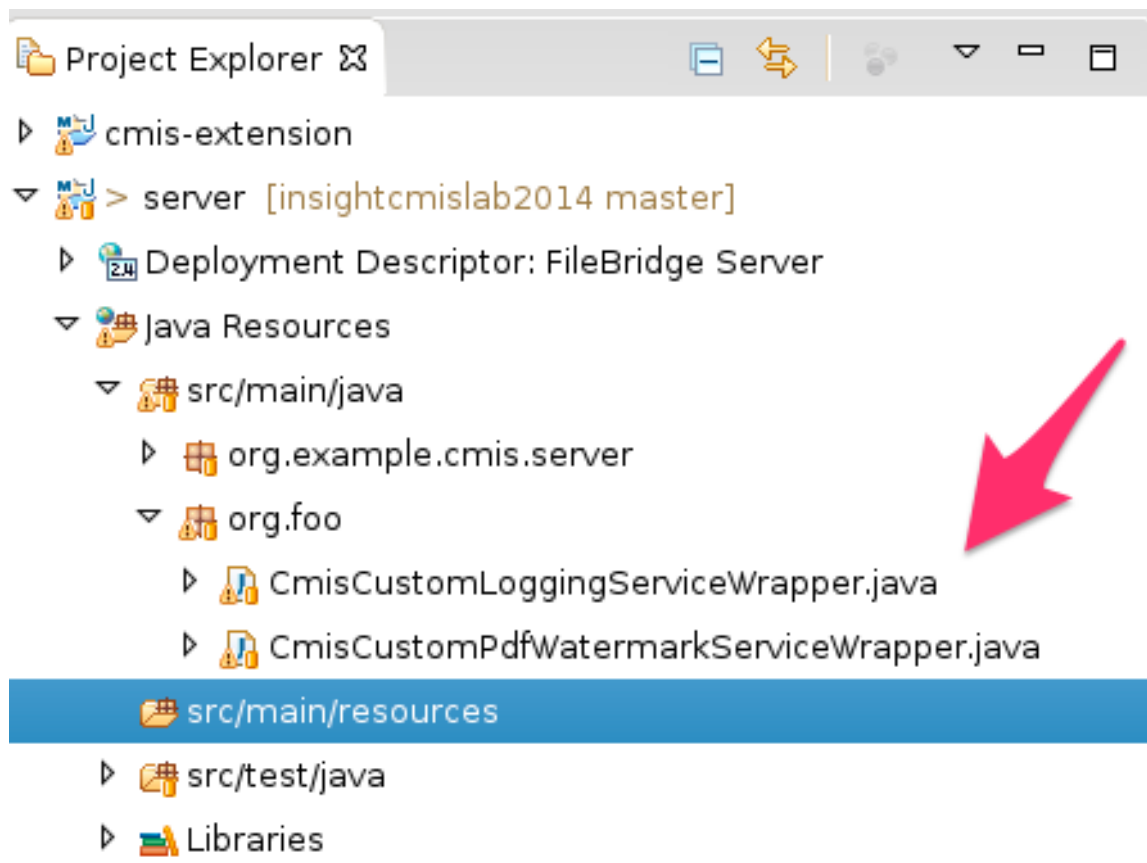
The first CMIS server plugin example is named `CmisCustomLoggingServiceWrapper`. This example is the simplest plugin you could make and still have it do something useful. In this case, the plugin will hook into any calls for (folder) `getChildren()` and log some details about the request and including the time it took (in milliseconds) for the CMIS service to process the request. When you look over this file you will see there is just one file needed for the entire plugin. That is all that is necessary since it inherits all of its default behaviors from its abstract superclass. Anything that we don't wish to override will be handled as a straight pass-through by the server framework.

The AbstractCmisServiceWrapper

As you can see by looking at the `CmisCustomLoggingServiceWrapper.java` file. The signatures of the methods that we override/modify are the same `CmisService` methods that we talked about in part I of this guide. If you know how to build CMIS servers with OpenCMIS then you already know what you need to build server plugins. It's the same interface.
Have a look at the `getChildren()` method now. Any code you place before the passthrough to the underlying service `getWrappedService().getChildren(...)` will be to modify the request. You can also choose not to call the underlying service at all and completely override with your own implementation.
Any code placed after the call to the base service will allow you to inspect and or modify the response from the vendor's CMIS implementation before you pass it on to the client.

Sidebar: Building a Server Plugin from scratch

Usually you want build an plugin as a separate project. OpenCMIS after version 0.12.0 provides a Maven archetype that generates an plugin project skeleton. In part 1 we explained how to use a Maven archetype to generate a server stub. The same rules apply here, only the archetype artifact ID is different.
Note: To run this you must have locally built the trunk of OpenCMIS. See the section titled 'Getting and building the latest OpenCMIS libraries' prior to attempting this.
To generate a plugin (aka extension) stub run:

```
mvn archetype:generate \
-DgroupId=org.example.cmis \
-DartifactId=server-extension \
-Dversion=1.0-SNAPSHOT \
-Dpackage=org.example.cmis.server.extension \
-DprojectPrefix=MyExtension \
-DarchetypeGroupId=org.apache.chemistry.opencmis \
-DarchetypeArtifactId=chemistry-opencmis-server-extension-archetype \
-DarchetypeVersion=1.0.0-SNAPSHOT \
-DinteractiveMode=false
```

Note: This archetype generation step can also be done GUI-style, with the latest version of Eclipse (with Maven integration). We are leaving that out of the tutorial to save space but feel free to use that instead (as an extra exercise) if you prefer.

Deploying the Plugin (if separate)

Since we are building our plugins as part of our server, all we need to do is compile and our server will already have the plugins included. But if we were building separately (the normal case) here is what you would have to do:
Compile your .jar file.
Take your .jar file from your target build directory and add it to the vendor's `\WEB-INF\lib\` in their WAR file.
In the default case your plugin will look something like this `cmis-extension-1.0-SNAPSHOT.jar.`

Note: Adding a plugin to a CMIS implementation that does not support 0.11.0 extensions will have no effect, so make sure your vendor has added support. If they have not, point them to this guide!

Now that your extension files are on the CMIS server's classpath, all that is left to do is hook them in by adding some instructions for the WrapperManager into the repository.properties file.   This is shown in the next section.

Registering Plugins

Again this part is already done for you in the sample but if you were doing this to a vendor's CMIS server this is how you would do it. Your plugins should be registered in the repository.properties file (located in `\WEB-INF\classes`) like this:

```
servicewrapper.1=com.example.my.SimpleWrapper
servicewrapper.2=com.example.my.AdvancedWrapper,1,cmis:document
servicewrapper.3=com.example.my.DebuggingWrapper,testRepositoryId
```

The number at the key is the position in the wrapper stack. Lower numbers are outer wrappers, higher numbers are inner wrappers closer to the vendor's service implementation.   So the wrapper with key 1 will be the first one executed when a request comes in and the last to see or change the response on the way out.
Wrappers can have parameters that can be attached as a comma separated list to the class name. They are provided to the wrapper implementation when the manager calls `AbstractCmisServiceWrapper.initialize().` The last two examples show these optional parameters.

In the case of our sample wrapper you will want to add the following line:

```
servicewrapper.1=org.foo.CmisCustomLoggingServiceWrapper
```

In our FileBridge example this line is already been added for you.

Running the Plugins

Now we are going to run the server and test the plugin. Start your server and login with the CMIS Workbench and browse a few folders. Every time we request folder children we see output in our console something like this:

```
2014-08-14 17:42:13,993 INFO  [http-bio-8080-exec-4]
org.foo.CmisCustomLoggingServiceWrapper: Operation: getChildren ,
Repository ID: test, Binding: browser, User Agent: Apache Chemistry
OpenCMIS/0.11.0

2014-08-14 17:42:13,998 INFO  [http-bio-8080-exec-4]
org.foo.CmisCustomLoggingServiceWrapper: [CmisCustomServiceWrapper]
Exiting method getChildren. time (ms):5
```

Now stop your server and set a breakpoint at the first line of the plugin's `getChildren` method. Re-start in debug mode and trace through the code as it is executed before and after the underling server `getChildren` method.

Building a Server Plugin - Example #2 - PDF Watermarking

Now that you have seen a very simple server plugin example.   We are going to build on what you have seen up till now and walk through a plugin that is a bit more useful. In this case we want to watermark any PDF (both graphically and in the metadata) that is retrieved from `FileBridge` only when the user is using the `reader` account.

Locate the `CmisCustomPdfWatermarkServiceWrapper` class in the `org.foo` package. This is the only file needed for this plugin however we are going to be sub contracting a lot of work out to the very powerful Apache PDFBox library. In order to tell Eclipse/maven that we have this dependency, we have added the following section to our `pom.xml` file.   Find this section in the POM and verify that you know where it should go.

Dependency section from our `pom.xml` showing the PDFBox requirement.
```
<dependency>
      <groupId>org.apache.pdfbox</groupId>
      <artifactId>pdfbox</artifactId>
      <version>1.8.6</version>
</dependency>
```

Handling Dynamic Parameters for your plugin

All plugins can have an arbitrary number of deploy-time supplied parameters. They take the form of name-value pairs. Navigate to your copy of the `repository.properties` file now and have a look at the `servicewrapper.2` line. Immediately after the full class name of our wrapper class you will see the first parameter, `userfilter` which is set to the value of `reader`.
Now lets go over to the initialize method on our `CmisCustomPdfWatermarkServiceWrapper` class and have a look at how this value is parsed and stored for later use for the wrapper.   Set a break point in this code and debug the application while logging in with Workbench.   How many times does it get called? Why so many?
Hint: Have a look at the code comments.
Finally we jump to the `getContentStream` method. Have a look here to see how we get the current user from the `sharedContext` and use it to toggle our watermark logic off and on.

OpenCMIS helper classes - ThresholdOutputStream

Often while doing client or server development with OpenCMIS you will encounter the need for helper classes. One such class is the `ThresholdOutputStream`.   Navigate to the `getContentStream` again and find the first occurrence of this class.   Rather than explain what this class does we will show you how to find out for yourself by inspecting the code in a really browser friendly way.   You may not be aware that the folks from Apache Chemistry keep their code mirrored in GitHub here:
https://github.com/apache/chemistry-opencmis

Go to this repository and do a search for `ThresholdOutputStream`. You will see a few references as well as the factory. If you look down the list eventually you will find the actual class. Note you could also just navigate directly to is via the package names.   Go to the source listing for the class now here:
https://github.com/apache/chemistry-opencmis/blob/9b821292e708c24827ed526d4226b4308eff91bf/chemistry-opencmis-server/chemistry-opencmis-server-bindings/src/main/java/org/apache/chemistry/opencmis/server/shared/ThresholdOutputStream.java

Have a look at the comments for the class. We will include part of them here in case you are reading this tutorial without an internet connection.

```
/**
 * An OutputStream that stores the data in main memory until it reaches a
 * threshold. If the threshold is passed the data is written to a temporary
 * file.
```

Take a bit of time to look through the file to see how it works.   Now next time you have a question about why something works the way it does in OpenCMIS you know how to find out!

CmisCustomPdfWatermarkServiceWrapper walkthrough

Since we have already covered the `initialize` method let's have a quick look over the rest of our wrapper version of `getContentStream`. We will deliberately not get into how to user pdfBox since that is well documented at the PDFBox web site.
(here: https://pdfbox.apache.org/)

The call to `getCallContext` is an important one to note. Generally most of the sorts of data about the current request will be found here since you can add whatever custom data you wish to this object.

Note: See the getService call in the `FileBridgeCmisServiceFactor` class for an example of setting a custom name value pair in the call context.

In this case all our work takes place after the call to
`getWrappedService().getContentStream()` since we need the underlying service to
retrieve the raw content stream for us from the repository.   Spend a couple minutes looking all this
over before we run it. If necessary place a break point here and step through a retrieve of a pdf
document.

The last change we will make (in addition to the watermark) is the tweak the PDF metadata with some
static text.   This will serve as an example for later if you want to stamp the document with the user's
login name or other identifing information like their ip address.   Navigate to the watermarkPDF
towards the bottom of the method.   See the line:

`pdf.getDocumentInformation().setTitle("Modified by fileBridge");`

Have a look at the documentation for PDFBox and see what other metadata you can change this way.
The screenshot in the exercise below shows the result of this static metadata change.
Optional extra: Pick a piece of dynamic information about this transaction (e.g. the time of day, the ip
address of the user, the user's id, etc.) and change some other PDF metadata fields to reflect these
values.

Running the watermark example

Now that we have a basic understanding of how this all fits together let's see it in action. Start your
server again, and login with Workbench. **This time be sure to use the `reader` account.** Navigate to
a directory where you have some multipage PDF files and load one of them by double clicking on it in
the left pane.   The screen shot below shows a PDF loaded with the fileBridge with the reader user.
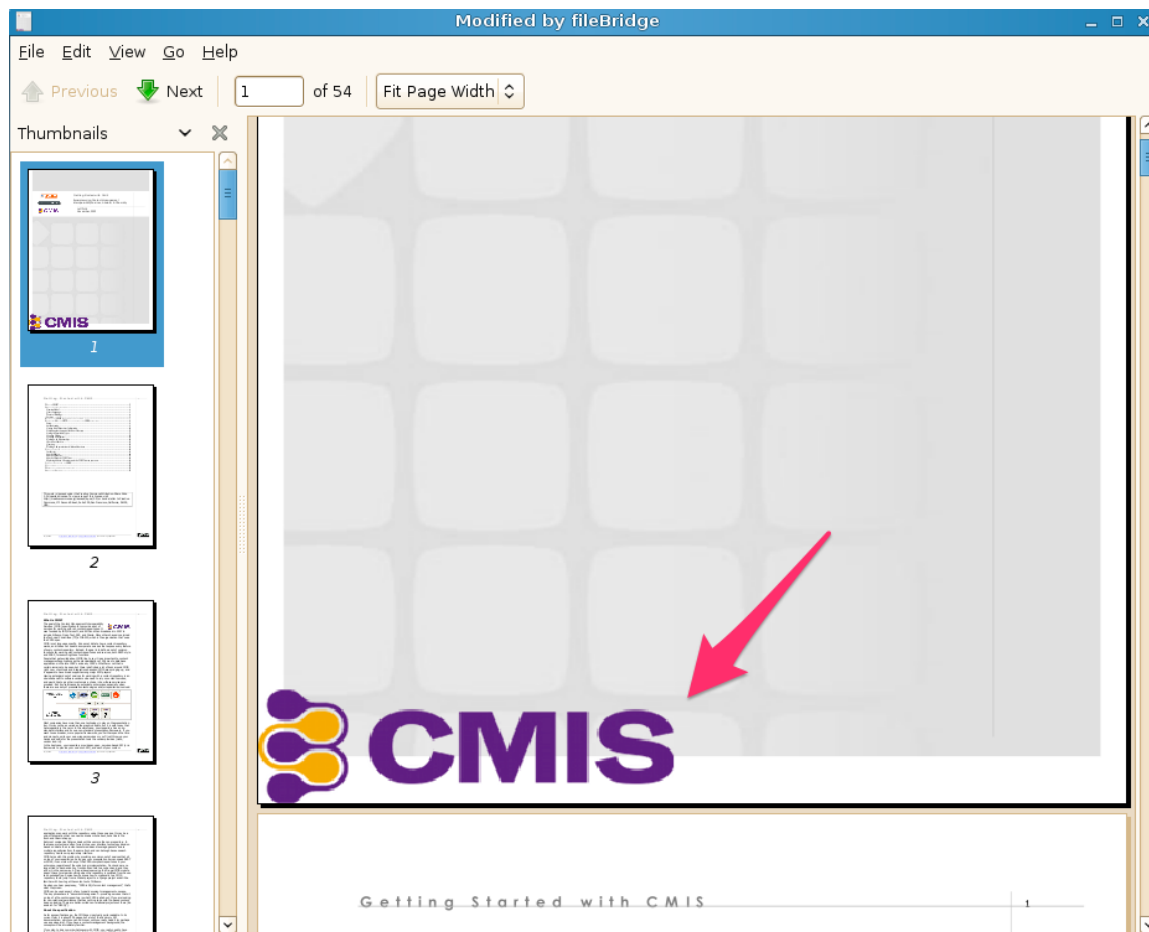
The arrow shows the CMIS graphic that has been added to the first page in the lower left. Also note the header of the application showing the new PDF's title "Modified by fileBridge".

Now check the original PDF by loading it directly from your filesystem to verify only your CMIS copy was modified.

Additional exercise ideas (advanced / optional)

Advanced (Level I):
Add custom document metadata support to your FileBridge server. The metadata can be stored in the local file system with a small .xml or .json file named something like originalfilename.ext.metadataFileBridge. Remember you have to handle the requests to your type collection as well if you are adding any properties that are not already part of CMIS.

Advanced (level II):
Add mutable document metadata to your FileBridge server. That is allow CMIS 1.1 clients to define new subclasses of files to support along with their own custom metadata. The resulting metadata would be stored in the same hidden shadow file as we used in the last example.

Manually deploying plugins (optional / intermediate)

In a real world scenario you would not be building your OpenCMIS server with the plugins already compiled in. Instead you would be taking an off the shelf working CMIS server and modifying its WAR file to include your additional logic. So let's do that now. If you have a real CMIS server that supports the server plugins (like the FileNet CMIS 1.1 Tech preview) then you can use that. If you don't have one handy you can use FileBridge and walk through these steps manually.

First build the `FileBridge` project from the command line to get a new copy of the WAR file from the `target` directory. Then take a copy of the `CmisCustomLoggingServiceWrapper` and create a brand new project. Rename the new logging wrapper to something different. The only other thing you will need is the `pom.xml` from our sample project and that one java wrapper file. Once you have it compiling, change the output of the logger so that you will be able to identify it uniquely from the other output. Then add the jar file for your new logging wrapper to the FileBridge WAR file's `lib` directory.

See the 'Deploying the Plugin' section for additional details on this. Next edit the `repository.properties` file to include your new wrapper then deploy the new WAR to a test Tomcat container. Spin it up and see if you missed anything. If it does not work you should have enough information at this point to debug and find out why.

Conclusion of Part 2

In part two you were introduced to the concept of CMIS runtime server plugins including their design. You have seen how to modify the FileBridge server from Part 1 to support these plugins. You have built your own simple custom plugin that will output logging information for all `folder.getChildren()` calls that arrive at your server. Finally, you created a plugin that watermarks all retrieved PDF documents requested by a flagged user.

We hope this document has been helpful for you in your understanding of building CMIS servers.   You are now armed with enough information to go out and customize CMIS servers that you have purchased and tailor them specifically for your needs. If you have questions about this material or suggestions for a future version please contact the authors directly.

Resources

OASIS CMIS 1.1 Specification :
http://docs.oasis-open.org/cmis/CMIS/v1.1/os/CMIS-v1.1-os.html

OASIS CMIS 1.1 Specification (PDF version) :
http://docs.oasis-open.org/cmis/CMIS/v1.1/CMIS-v1.1.pdf

CMIS Cheat Sheet : http://cmis.alfresco.com/cmis-cheatsheet.pdf

CMIS and Apache Chemistry in Action (Manning) : http://www.manning.com/mueller/

OpenCMIS 0.12.0 JavaDoc : http://chemistry.apache.org/java/0.12.0/maven/apidocs/

Apache Maven : http://maven.apache.org/

Eclipse : http://eclipse.org/

Github mirror of OpenCMIS : https://github.com/apache/chemistry-opencmis

SLF4J : http://www.slf4j.org

Apache PDFBox : https://pdfbox.apache.org/

Apache Chemistry : http://chemistry.apache.org/