Chamilo
E-Learning & Collaboration Software

Opening Pandora's Box

Chamilo 2.0 Repositories

Opening Pandora’s Box

Chamilo 2.0 Repositories

By Hans De Bisschop

Credits

Author

Hans De Bisschop (hans.de.bisschop@ehb.be)

Developers

- Hans De Bisschop (hans.de.bisschop@ehb.be)
- Magali Gillard (magali.gillard@ehb.be)
- Nicolas Rod (nicolas.rod@unige.ch)
- Jens Vanderheyden (jevdheyd@vub.ac.be)

Special thanks

- Patrick Roth (patrick.roth@unige.ch)
- Sven Vanpoucke (sven.vanpoucke@hogent.be)
- Eduard Vossen (eduard.vossen@ehb.be)

... and the entire Chamilo community

We keep moving forward, opening new doors, and doing new things, because we're curious and curiosity keeps leading us down new paths.

- Walt Disney (1901-1966)

Contents

Introduction

re•pos•i•tor•y
[ri-poz-i-tawr-ee, -tohr-ee]
–noun, plural -tor•ies.

1. a receptacle or place where things are deposited, stored, or offered for sale: a repository for discarded clothing.
2. an abundant source or supply; storehouse: a repository of information.

So there you have it: a place where information is deposited. Repositories are ancient, there's nothing new about them. Mankind has been gathering and storing knowledge ever since they realized they can do things which no other critter on this planet can do (for now). Some of these repositories have taken on almost legendary proportions like e.g. the library of Alexandria, that famous repository of knowledge of the ancient world. For centuries though, such repositories were only accessible for a select group of people: scholars, nobility, clergy, etc. After all knowledge is power and quite a few people didn't exactly feel like sharing that power.

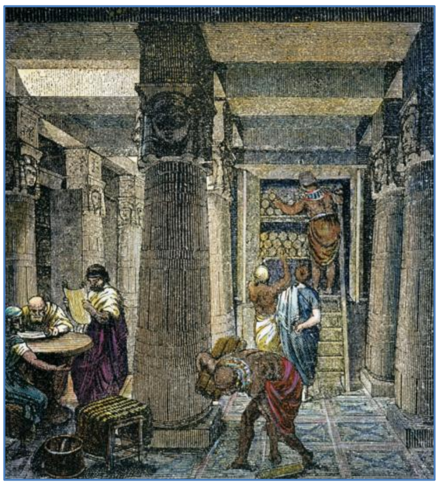


Fig. 1 – The ancient library at Alexandria

But let's flash-forward in time just a little bit, to the 20th century to be somewhat more exact. The "microcomputer" was born, which opened up a multitude of options concerning the sharing of knowledge. But unless you went around exchanging a diverse selection of ever more exotic media to get knowledge from one computer to another, sharing knowledge was still cumbersome. That turned out to be a minor obstacle though. We can't share information between computers? Well, then let's hook them up in one big global network. Anno 2010 we call this the "internet".

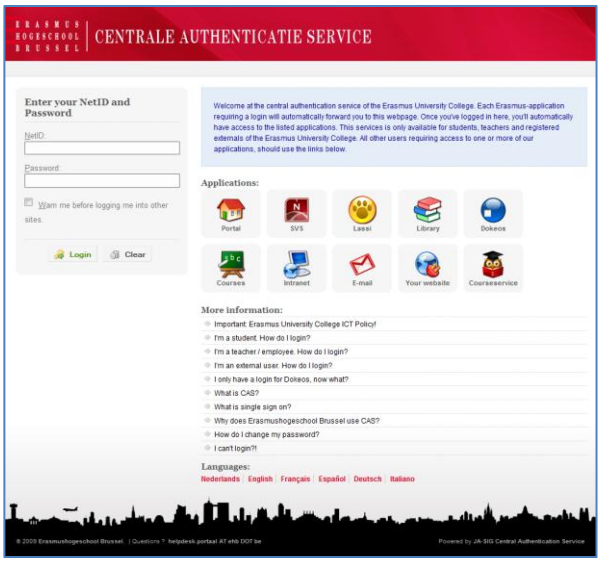
Coming up with a global (no pun intended) definition of the internet is difficult to say the least. For the sake of simplicity and to keep things confined to the context of this article its best regarded as an almost infinite repository of knowledge. There are the obvious repositories like Wikipedia[1] and YouTube, but also things like the BBC News[2] website, a local broadcaster's television guide, etc.

There's thousands of ways we can use these repositories to share information, but because of that global information highway at least we can do it with relative ease. At the same time we risk flooding people with enormous amounts of information which might be irrelevant in their specific case. So we built repositories on top of repositories to facilitate our quest for that particular piece of knowledge. Google[3] is just one the many examples of super-repositories out there. It "knows" a lot of things

[1] http://www.wikipedia.org/, The Free Encyclopedia
[2] http://www.bbc.co.uk/news/
[3] http://www.google.com/

about an even bigger amount of websites, thus making it easier for us to find that which we are looking for. At least in theory.

Resistance is futile: Do it yourself of integration?


ERASMUS HOGESCHOOL BRUSSEL | CENTRALE AUTHENTICATIE SERVICE

Enter your NetID and Password

NetID:

Password:

☐ Warn me before logging me into other sites.

Login | Clear

Welcome at the central authentication service of the Erasmus University College. Each Erasmus-application requiring a login will automatically forward you to this webpage. Once you've logged in here, you'll automatically have access to the listed applications. This services is only available for students, teachers and registered externals of the Erasmus University College. All other users requiring access to one or more of our applications, should use the links below.

Applications:

Portal | SVS | Lassi | Library | Dokeos | Courses | Intranet | E-mail | Your website | Courseservice

More information:

- Important: Erasmus University College ICT Policy!
- I'm a student. How do I login?
- I'm a teacher / employee. How do I login?
- I'm an external user. How do I login?
- I only have a login for Dokeos, now what?
- What is CAS?
- What is single sign on?
- Why does Erasmushogeschool Brussel use CAS?
- How do I change my password?
- I can't login?!

Languages:

Nederlands | English | Français | Español | Deutsch | Italiano

© 2009 Erasmushogeschool Brussel | Questions ? helpdesk.portaal AT ehb DOT be

Powered by JA-SIG Central Authentication Service

Fig. 2 – The login screen of Jasig's Central Authentication Service implementation at Erasmus University College

The proverbial project X is very interesting, because it contains features Y and Z. What to do if you would like to offer that same functionality on your own platform? Do you implement that functionality yourself from scratch or do you build a bridge between the 2 projects allowing them to communicate flawlessly? Depending on the projects and the context, the answer will be different. I can remember very "interesting" theoretical discussions about the issue and at the time quite a few people thought option 1 was an absolute must from a pedagogical point of view. What did end users want? Somehow they managed to "forget" to ask. The result? After a few years of development you have a home built tool which no one wants to use because it's already outdated, doesn't boast as many features as their more popular counterparts and simply isn't what people are already used too. At that point it became very tempting to say "I told you so, I said that was going to happen, but you wouldn't listen". Needless to say I just kept silent and enjoyed the moment.

Does that mean we shouldn't do things ourselves at all? Absolutely not. In specific cases what's out there simply might not cover your needs on any level. When that happens you can seriously start to think about starting up a project yourself. Although even at that time, you're probably not all that fond of having to reinvent the wheel all by yourself. But that's why we invented that wonderful thing called communication and again the internet greatly facilitates things. In no time at all you can find likeminded people and institutions from all over the world that may be facing the same problems. More often than not these kinds of situations lead to some very interesting projects being created which cover an enormous amount of requirements formulated by a multitude of interested parties.

A very nice example of such a project is Jasig's Central Authentication Service[4].

From Wikipedia:
CAS is a single sign-on protocol for the web. Its purpose is to permit a user to access multiple applications while providing their credentials (such as userid and password) only once. It also allows web applications to authenticate users without gaining access to a user's security credentials, such as a password

[4] http://www.jasig.org/cas

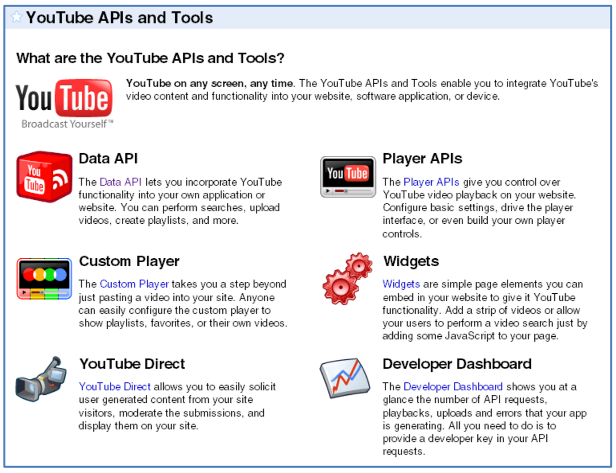
Years of collaboration have resulted in a completely open protocol which is now being used by countless organizations and millions of users worldwide. On top of that clients have been written for almost all popular programming languages and authentication modules exist for a lot of applications to allow them to use CAS for authentication. (Chamilo 2.0 being one of them)

But what if you need more than that which CAS currently offers? No problem. By carefully choosing the frameworks upon which the software is built and because the developers opted for a very dynamic, flexible and open architecture every developer worth mentioning should be able to adapt the system for his or her use case without changing as much as one single line of the original source code. Isn't that great?

Sadly enough this isn't always an option. Not every piece of software out there is as open as we would like it to be … and I'm not only talking about actual closed source software. There's plenty of open source software out there which might be nice on its own but utterly useless if you have the unfortunate assignment to integrate it with your learning content management system (LCMS). In a way the issue is not at all related to the ongoing war between open and closed source software providers. It's a matter of making data accessible in a generalized way for both end users and developers alike.

To API or not to API? It's not that difficult a question

Access to data is essential when you want to link two systems and that's when application programming interfaces (API) and/or web services become vital. They allow us to access and use a particular piece of functionality without actually having to know anything, or very little, about the software's inner workings.



YouTube APIs and Tools

What are the YouTube APIs and Tools?

YouTube on any screen, any time. The YouTube APIs and Tools enable you to integrate YouTube's video content and functionality into your website, software application, or device.

Data API
The Data API lets you incorporate YouTube functionality into your own application or website. You can perform searches, upload videos, create playlists, and more.

Player APIs
The Player APIs give you control over YouTube video playback on your website. Configure basic settings, drive the player interface, or even build your own player controls.

Custom Player
The Custom Player takes you a step beyond just pasting a video into your site. Anyone can easily configure the custom player to show playlists, favorites, or their own videos.

Widgets
Widgets are simple page elements you can embed in your website to give it YouTube functionality. Add a strip of videos or allow your users to perform a video search just by adding some JavaScript to your page.

YouTube Direct
YouTube Direct allows you to easily solicit user generated content from your site visitors, moderate the submissions, and display them on your site.

Developer Dashboard
The Developer Dashboard shows you at a glance the number of API requests, playbacks, uploads and errors that your app is generating. All you need to do is to provide a developer key in your API requests.

Fig. 3 – Google provides an extremely extensive API for most of the services it provides, as well as several client libraries.

Do I really care how YouTube handles the conversion of the video I am uploading? I probably don't, I just want it to work and if at all possible I want to upload a video directly from my LCMS onto YouTube and use it in the body text of an announcement. Unless YouTube provided a fully documented API this would have been hard at best, impossible at worst.

The API and/or web services allow us to use a substantial set of features offered by the software from within the relative comfort of our own platform. On top of that it allows you to maintain the same visual style throughout the entire content creation process, saves the user the trouble of having to re-authenticate x-number of times or go through a ridiculous amount of steps to add his content inside your platform.

Let's have a look at what the user would have to do to add some video to an announcement.

The usual way

1. Log in on YouTube
2. Upload a video, adding some metadata in the process
3. Open the video page and copy the url or code snippet
4. Log in to Chamilo
5. Access your course
6. Create an announcement, adding some text
7. Select the html editor's video plug-in[5] and paste the URL or add the snippet to the source
8. Save the announcement

Now, wouldn't it be easier if we could just…

1. Log in to Chamilo
2. Access a course
3. Create an announcement, adding some text
4. Directly select the video file to add using the html editor's video plug-in
5. Save the announcement

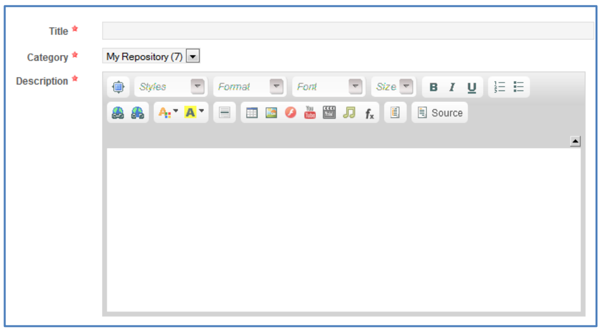


Fig. 4 – Chamilo 2.0's HTML Editor includes several content plug-ins for multimedia

No need to switch sites, no need to constantly re-authenticate and no need to copy all kinds of html mark-up and paste it who knows where in your document. Why should you bother an end user with actual HTML code, that's just so 1980s isn't it?

More and more software projects as well as free and commercial hosted services are providing or publishing API's to allow exactly that. Instead of previously accessing the data by means of several SQL commandos, imitated requests and posts and perhaps some filtering of the resulting html messages, you simply call one particular method and pass on a few parameters. Again: isn't that great?

Particularly popular nowadays are so called RESTful web services and/or APIs and its being used by quite a few of the repositories we've linked to Chamilo 2.0.

From Wikipedia:
REST-style architectures consist of clients and servers. Clients initiate requests to servers; servers process requests and return appropriate responses. Requests and responses are built around the transfer of "representations" of "resources". A resource can be essentially any coherent and meaningful concept that may be addressed. A representation of a resource is typically a document that captures the current or intended state of a resource.

At any particular time, a client can either be in transition between application states or "at rest". A client in a rest state is able to interact with its user, but creates no load and consumes no per-client storage on the set of servers or on the network.

[5] Chamilo 2.0 uses CKEditor 3.x for HTML editing, http://ckeditor.com/

From Wikipedia (continued):
The client begins sending requests when it is ready to make the transition to a new state. While one or more requests are outstanding, the client is considered to be in transition. The representation of each application state contains links that may be used next time the client chooses to initiate a new state transition.

REST was initially described in the context of HTTP, but is not limited to that protocol. RESTful architectures can be based on other Application Layer protocols if they already provide a rich and uniform vocabulary for applications based on the transfer of meaningful representational state. RESTful applications maximize the use of the pre-existing, well-defined interface and other built-in capabilities provided by the chosen network protocol, and minimize the addition of new application-specific features on top of it.

A RESTful web service (also called a RESTful web API) is a simple web service implemented using HTTP and the principles of REST. It is a collection of resources, with three defined aspects:

- the base URI for the web service, such as http://example.com/resources/
- the MIME type of the data supported by the web service. This is often JSON, XML or YAML but can be any other valid MIME type.
- the set of operations supported by the web service using HTTP methods (e.g., POST, GET, PUT or DELETE).

At any rate interacting with these RESTful APIs or web services is proving to extremely easy when compared to the alternatives. Quite a few of the currently available RESTful APIs are based on the atom publishing protocol, which is in itself a protocol adhering to the REST architecture.

The openness created by these kinds of interfaces can’t be underestimated. But we have to be honest; it’s not entirely without risks either.

Something stirs in the east … or the risks of going external

Linking or integrating an external repository to your own is not entirely without risks. The link will only work as long as the interface we use to communicate with the service remains unchanged. Since the only software that doesn’t change is discontinued software, this is a pretty serious risk. But once again that’s where openness becomes important. If changes to APIs are implemented gradually, well documented and remain backwards compatible for an extended period of time, you shouldn’t be facing any real issues.

The minor changes you’ll have to implement will still outweigh the cost of reinventing the wheel by far. The impact of changes also depends on the nature of the product you’re using. Free or commercial hosted services could be considered the biggest risk of all. You don’t control when they get updated and can only hope that its developers have struck a decent balance between openness and any kind of commercial interests that might be involved. So just make sure everyone understands the risks involved completely when and if you should choose to heavily rely upon such a service.

Whenever there's a hosted service for those of us without big budgets, there's always some kind of project around which will more than likely do pretty much the same thing (and maybe even more), but will allow you to set up your own servers hosting one or more instances of the repository software. This doesn't eliminate the risk of updates, but in a lot of cases it does eliminate the risk of updates which weren't communicated or which end up breaking things. At the very least you should have the time to verify whether everything will keep on working, make changes if necessary and then update.

At any rate choosing any kind of solution or project for a specific use case should be well researched. Whether you'll be developing your own application, using an existing one out of the box or modifying it for your own needs, make sure there are no insurmountable problems before you get started. If you don't you're taking an extremely big leap of faith, without a parachute.

On the origin of species or how Chamilo 2.0 was born

Every once in a while you have to take a leap of faith though and that's exactly what we did with Chamilo 2.0. Most LMS' nowadays still focus on the teacher and the course, with the student being little more than someone who's watching at the sidelines. Content is mostly provided for them and can't be altered or improved upon. So how can you learn anything if you're not an actual part of the learning process. That's like spending the entire soccer match on the bench only to get complaints why you didn't score a goal afterwards.

So, out with the course based concept and let's make a system that focuses on the user. "The user" could be a teacher, a student, a member of staff, etc. at any rate such labels are little more than a specific function that person has on a specific location at a specific point in time. It should by no means be set in stone and unchangeable for all eternity. Considering that a user's function may change, wouldn't it be more logical then if he owns all his content, which is collected in a central content repository? That way he could share it with other users and they actually have the ability to collaborate on something, outside the context of a course!

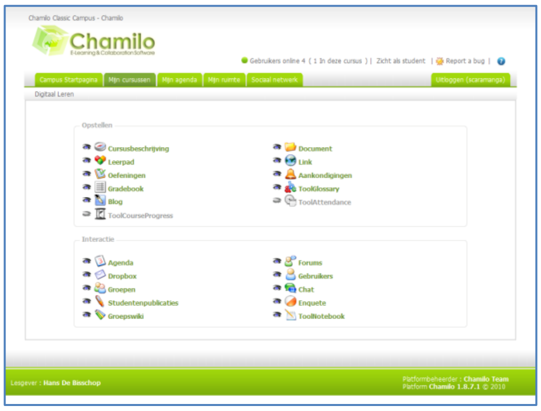


Fig. 5 - A typical Chamilo 1.8 course homepage

In some cases the entire concept of a "course" might even be outdated or irrelevant. Learning is so much more than just following courses. Portfolios, wikis, video conferencing, etc. can all be an integral part of the learning process. But wait a minute? That portfolio might actually want to display some content which I previously created for a course, isn't that going to be a problem? Not at all, considering that we've got an actual repository and are no longer storing everything inside the context of a course.

Allowing different usages of the same content also implies having different contexts to use them in. The software we now know as Chamilo 1.8 is just one of many applications being developed for Chamilo 2.0. As such the new platform has become somewhat of a framework allowing developers to write their own applications without having to worry about several key ingredients like user and group management, roles and rights, installations, package management, etc. Several interfaces exist to access the functionality of these so called core applications, making it very easy for a developer to allow the end user to use his repository content in the application.

These objects are no longer copied from one location to another (which would be a nightmare as far as version management is concerned) but published. Publications, in essence, are little more than links to the actual object in a repository, placed in a certain context.

In essence Chamilo had just evolved from a strict learning management system to a (learning) content management system, allowing it to be used for basically anything you may want to use it.

The sky's the limit, unless you have a space shuttle

Anything? Surely you must be joking? No sir, I kid you not. This, for example, is actually a website using Chamilo 2.0 as its back-end.

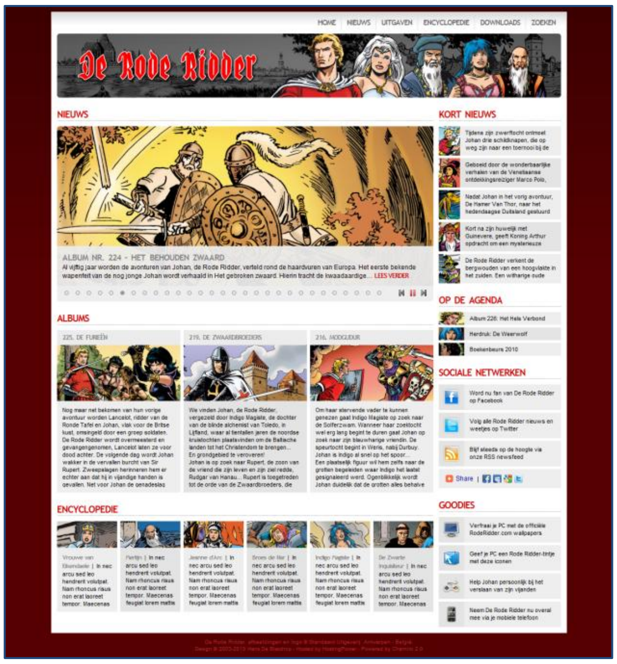


Fig. 6 – http://www.roderidder.com, information website for a Belgian comic book series

This just illustrates that virtually anything is possible once you have a working repository as the basis for your platform. It also offers possibilities for smaller organizations or schools who can't afford to host a separate website and an LCMS at the same time.

Even so, it would be somewhat arrogant to think that Chamilo 2.0 is the software equivalent of the Holy Grail. (The lost ark, maybe, but not the Holy Grail) Some functionality is just so specific and so complex that you really need a dedicated solution for it. Sometimes that might be special software, but in a lot of cases it also involves a somewhat more specialized hardware setup. Streaming media is a perfect example of a dedicated setup, simply because of the server load caused by the conversion of video, ranging from Full HD to resolutions suitable for smartphones.

But it doesn't take a rocket scientist to figure out that people tend to "want" these kinds of bells and whistles and at the same time they want them to be simple to use. If anything streaming video is not simple or easy … the software that handles it even less so. I would challenge you to explain to the average teacher that he has to upload the movie file to a server via FTP, then access a special administration page, get bombarded with tons of options in what seems like Chinese to them and who knows what else.

Even if they manage all that (0,05% of all users?) they still have to actually use it within the Chamilo context. Oh my.

Eh...what's up, doc?

It was time to think about a solution. How could we integrate all these kinds of external content? How could we do it in such a way that adding new external repositories to the list was as easy as possible? As a matter of fact: what exactly will we allow users to do with these repositories and their content? It was time for a requirements list:

- Browse the content of the repository based on the user's rights
- Show previews of specific repository content
- Directly add content to the repository
- Import or link objects from the repository to the user's own repository in Chamilo 2.0
- Edit an object's metadata (including but not limited to title, description, etc.)
- Export local objects to one or more repositories
- Manage the synchronization status of local and external objects
- Uniform layout across all implementations to keep things simple for the end-user
- Extendible framework allowing a functional implementation with as little code as possible
- Provide abstractions for repositories implementing a certain kind of standard or content type (e.g. MediaMosa, Opencast Matterhorn and YouTube all handle streaming media)
- Allow multiple instances of a repository type if and when supported (e.g. there's only one YouTube, but there's who knows how many Fedora Commons repositories out there)

While trying to determine basic functionality for a pseudo-application that should manage these repositories, you inevitably end up compiling a wish-list of repositories you'd like to see integrated at one point.

Images & Photos				
23	flickr	photobucket	Picasa	WIKIMEDIA COMMONS
Audio & Video				
MM	matterhorn	YouTube	vimeo	SOUNDCLOUD
General repositories				
	box	Chamilo E-Learning & Collaboration Software	Alfresco	Fedora Commons™
Protocols				
CMIS, WebDAV, JSR 170/283, …				

It doesn't hurt to be ambitious, right? At any rate it's just the tip of the iceberg. Supporting one or more generic content exchange protocols / standards should facilitate integration even more. To facilitate accessing specific repositories via RESTful web services it would be very practical to have some readily available libraries which developers can use to communicate with the server of their choice.

Good news everyone: it's alive … twice!

The first steps towards integration of external repositories in Chamilo 2.0 were taken by the development team of the University of Geneva. Through the official network provider of all Swiss educational institutions, SWITCH, they have access to a content repository called SWITCHcollection[6].

From Switch:
SWITCHcollection is a national library of reusable learning objects like courses, modules, images, video clips and text documents contributed by swiss universities

Today's Situation with e-Learning Content

- digital learning content production is increasing
- content is distributed and stored in institutional, local or private LMS or CMS
- re-using digital content for education can be tricky due to copyright issues

Goals and Motivation of SWITCHcollection

- Re-use e-learning content
- Enhance intra- and inter-instituional collaboration
- Make teaching activities visible to peers and to the public
- Attract students
- Long-term archiving and distribution of content with stable URL (content lifecyle)
- Make teaching activities citable and referenceable
- Single point of entry for all kinds of e-learning contents - independently from distribution platform or learning management system

Key Requirements

- Very easy to use: easy to contribute and reuse contents
- Metadata model: simple for common usage but extensible for special applications
- Tight integration with existing learning management systems
- Customizable user interfaces
- Federation of national repositories with a single search service
- Authors control access rights, content license and usage policies

SWITCHcollection is based on the Fedora Commons digital asset management project and as such provided the perfect incentive to implement a first external repository in Chamilo 2.0. The 2.0 development team got several presentations of the extension and was impressed to say the least. The extension already implemented quite a few of the features that would later be formulated as basic requirements for any and all external repository connections. A small step for one organization, but a pretty big one for Chamilo 2.0.

But just like the moon landings, other forces were at work in the wonderful world of Chamilo and apart ideas of additional repository implementations were shelved for the time being. One spring day in 2010 however the need arose for integration of streaming media servers. In the previous months

[6] https://collection.switch.ch/

some efforts had already been made to integrate YouTube somewhat more intuitively, but now we had to go further. We needed some kind of framework to allow integration of several streaming media services.

The framework would have to support both YouTube (a hosted service with a public API[7]) and MediaMosa (an open source solution being researched by the Vrije Universiteit Brussel). Due to the continued efforts of an intern over at Erasmus University College and the team at Vrije Universiteit Brussel, we managed to produce a basic, working framework to support the majority of streaming video solutions. For reasons unknown the Fedora Commons extension was completely overlooked and potentially overlapping features were not merged yet at this point.

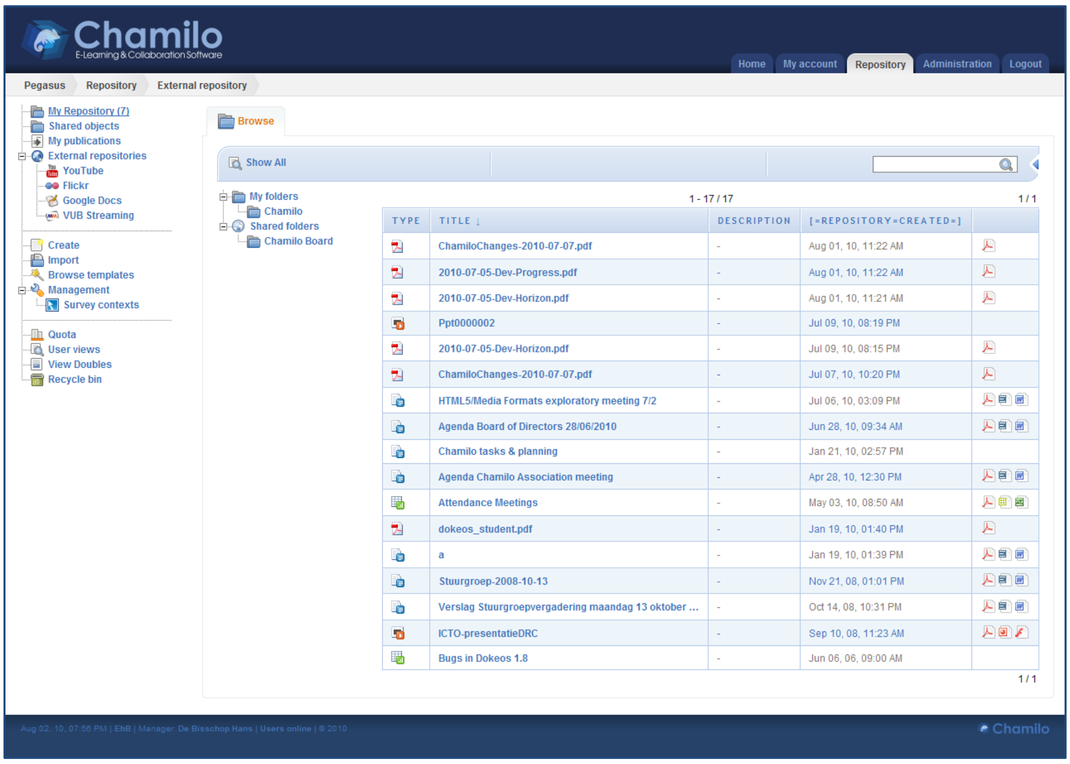
We soon connected the dots though and compared both mini-frameworks functionality-wise. The streaming framework lacked synchronization information and multiple instances of a specific repository type and the Fedora Commons extension lacked generalized components. In theory merging them should produce a pretty neat solution. Once more: isn't that great?

Elementary, my dear Watson

Before we have a look at the technical part of the solution, let's do a visual walkthrough of the Chamilo External Repository Manager and its functionality.

First of all there are 2 ways to access the manager, both fulfilling a specific task.

Inline External Repository Manager



1 - 17 / 17 1 / 1

TYPE	TITLE ↓	DESCRIPTION	[=REPOSITORY=CREATED=]	
	ChamiloChanges-2010-07-07.pdf	-	Aug 01, 10, 11:22 AM	
	2010-07-05-Dev-Progress.pdf	-	Aug 01, 10, 11:22 AM	
	2010-07-05-Dev-Horizon.pdf	-	Aug 01, 10, 11:21 AM	
	Ppt0000002	-	Jul 09, 10, 08:19 PM	
	2010-07-05-Dev-Horizon.pdf	-	Jul 09, 10, 08:15 PM	
	ChamiloChanges-2010-07-07.pdf	-	Jul 07, 10, 10:20 PM	
	HTML5/Media Formats exploratory meeting 7/2	-	Jul 06, 10, 03:09 PM	
	Agenda Board of Directors 28/06/2010	-	Jun 28, 10, 09:34 AM	
	Chamilo tasks & planning	-	Jan 21, 10, 02:57 PM	
	Agenda Chamilo Association meeting	-	Apr 28, 10, 12:30 PM	
	Attendance Meetings	-	May 03, 10, 08:50 AM	
	dokeos_student.pdf	-	Jan 19, 10, 01:40 PM	
	a	-	Jan 19, 10, 01:39 PM	
	Stuurgroep-2008-10-13	-	Nov 21, 08, 01:01 PM	
	Verslag Stuurgroepvergadering maandag 13 oktober ...	-	Oct 14, 08, 10:31 PM	
	ICTO-presentatieDRC	-	Sep 10, 08, 11:23 AM	
	Bugs in Dokeos 1.8	-	Jun 06, 06, 09:00 AM	

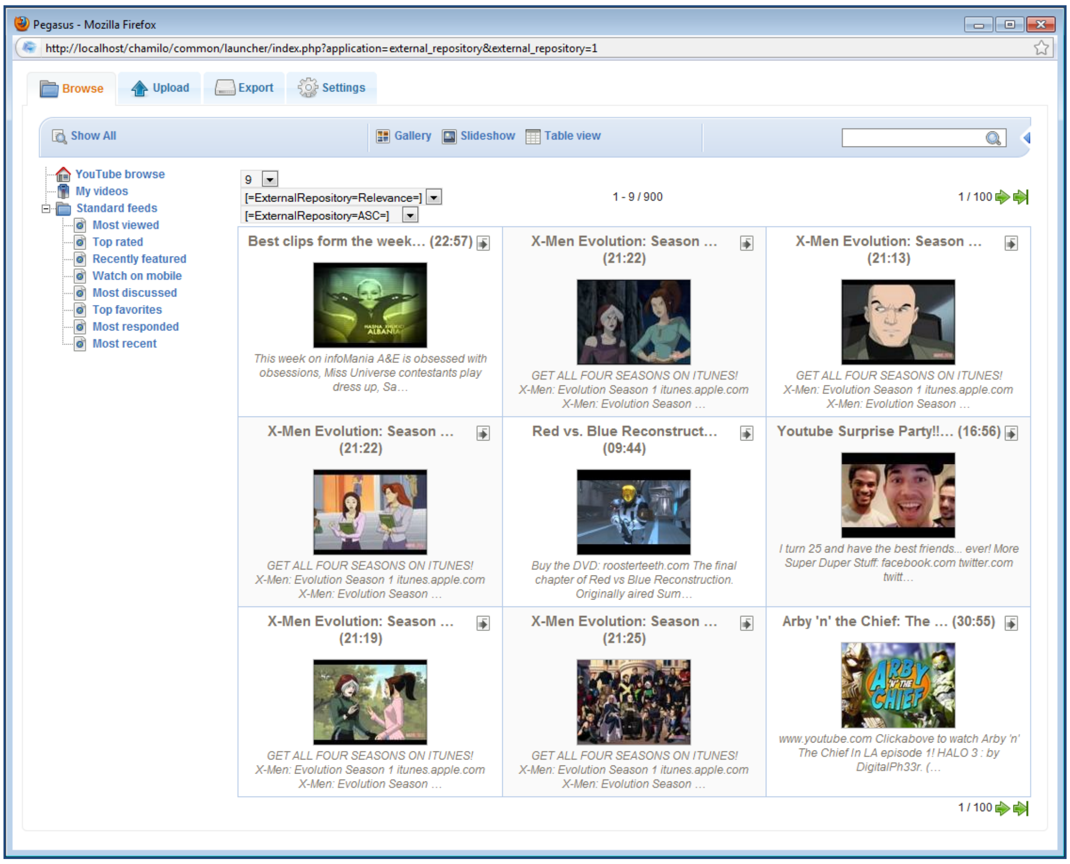
1 / 1

Aug 02, 10, 07:56 PM | EhB | Manager: De Bisschop Hans | Users online | © 2010

7 http://code.google.com/apis/youtube/overview.html

The inline modus is particularly useful when you want to use the external repository manager directly within the context of your application, which in this case is the user's personal repository.

Standalone External Repository Manager



Best clips form the week... (22:57)
This week on infoMania A&E is obsessed with obsessions, Miss Universe contestants play dress up, Sa...

X-Men Evolution: Season ... (21:22)
GET ALL FOUR SEASONS ON ITUNES! X-Men: Evolution Season 1 itunes.apple.com X-Men: Evolution Season ...

X-Men Evolution: Season ... (21:13)
GET ALL FOUR SEASONS ON ITUNES! X-Men: Evolution Season 1 itunes.apple.com X-Men: Evolution Season ...

X-Men Evolution: Season ... (21:22)
GET ALL FOUR SEASONS ON ITUNES! X-Men: Evolution Season 1 itunes.apple.com X-Men: Evolution Season ...

Red vs. Blue Reconstruct... (09:44)
Buy the DVD: roosterteeth.com The final chapter of Red vs Blue Reconstruction. Originally aired Sum...

Youtube Surprise Party!!... (16:56)
I turn 25 and have the best friends... ever! More Super Duper Stuff: facebook.com twitter.com twitt...

X-Men Evolution: Season ... (21:19)
GET ALL FOUR SEASONS ON ITUNES! X-Men: Evolution Season 1 itunes.apple.com X-Men: Evolution Season ...

X-Men Evolution: Season ... (21:25)
GET ALL FOUR SEASONS ON ITUNES! X-Men: Evolution Season 1 itunes.apple.com X-Men: Evolution Season ...

Arby 'n' the Chief: The ... (30:55)
www.youtube.com Clickabove to watch Arby 'n' The Chief In LA episode 1! HALO 3 : by DigitalPh33r. (...

In some cases it might be impractical or impossible for the user to navigate away from his or her current page to select or view an object in an external repository and that's when the standalone modus comes in handy. When running in standalone mode, the external repository manager will open in a popup or new window and won't interfere with the page you were on. Once you've selected the object you want to use the window will be closed and your selection will be processed according to the needs and requirements of the external object and the application you're using.

External Repository Manager Components

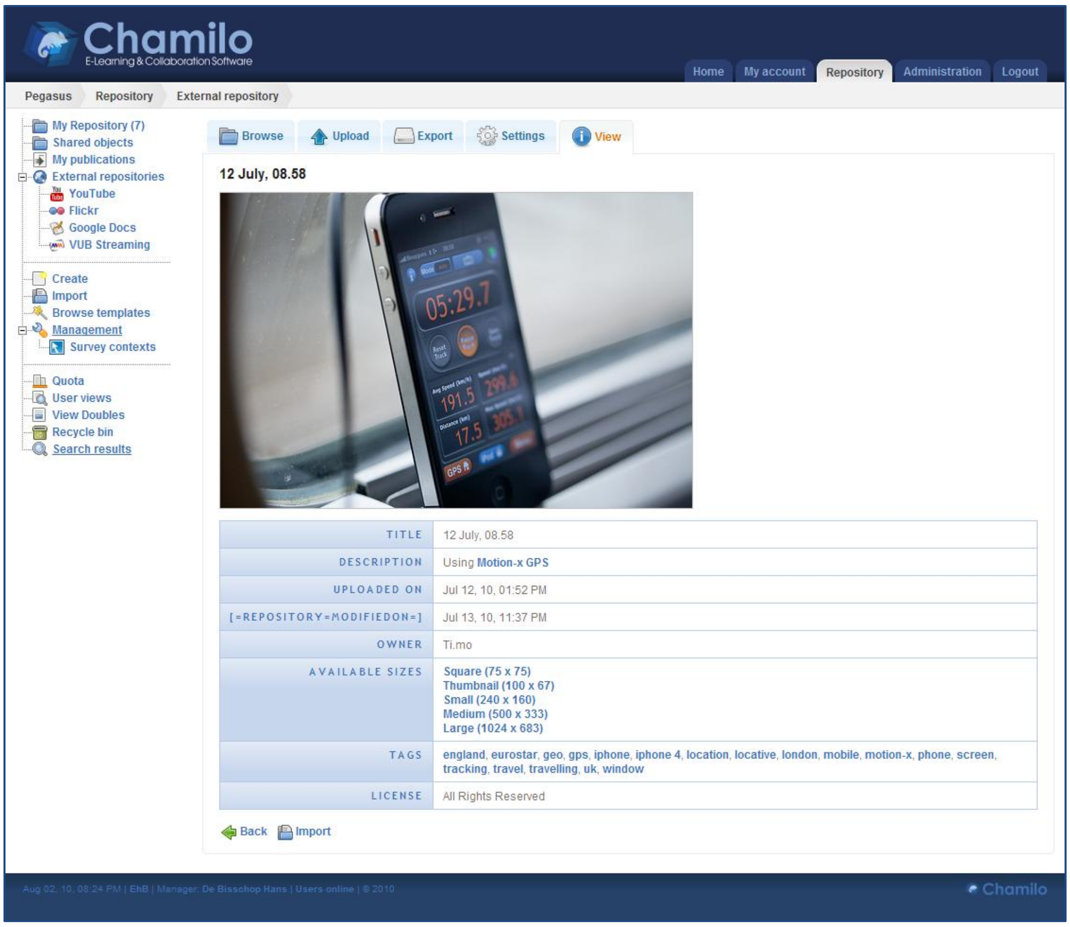
Several generic components have been implemented making it very easy for developers to quickly add functionality which end users might consider "basic".

Browser

As the backbone of the entire manager it provides an overview of all the objects available in the external repository. Results can be filtered based on search queries or predefined categories. Depending on the implementation several views could be available for the browser: gallery, slideshow and table. Examples of both can be seen in the previous sections on the inline and standalone variants of the manager.

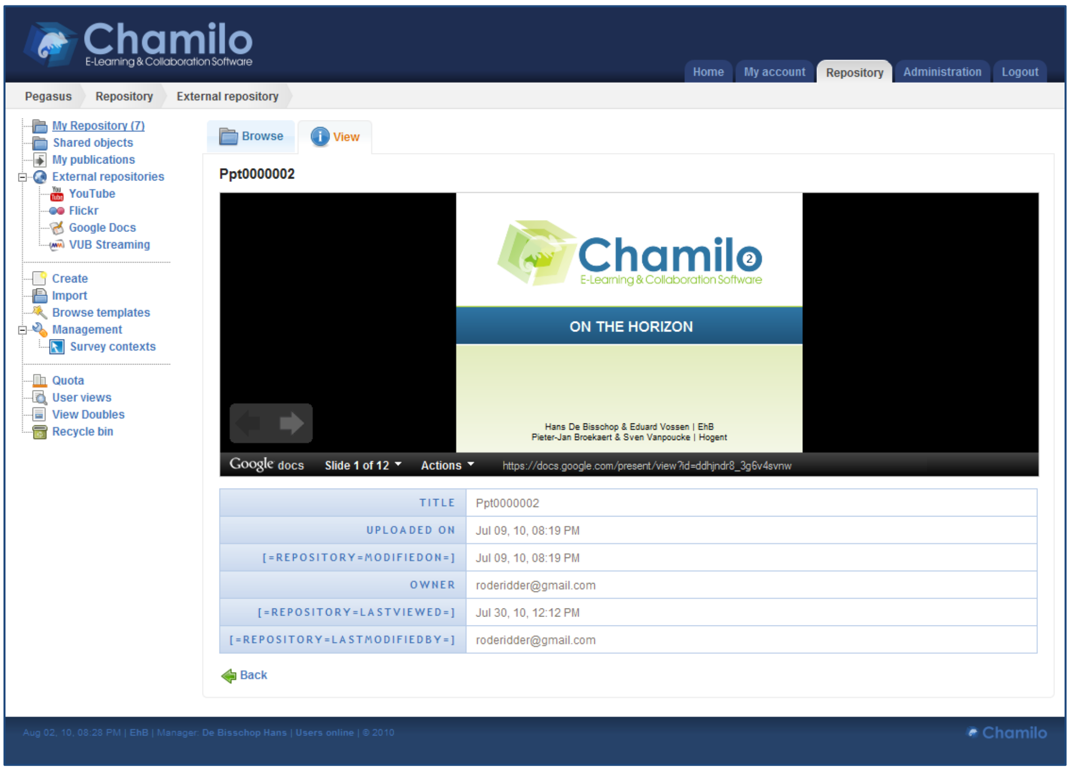
Viewer

The external repository object's viewer allows you to get a more detailed view of the object you selected. This could be an enhanced preview (e.g. embedded player for streaming media), additional properties, specific actions, etc. Apart from some basic features, it all pretty much depends on how far the developer of the implementation was willing to go.



12 July, 08.58

TITLE	12 July, 08.58
DESCRIPTION	Using Motion-x GPS
UPLOADED ON	Jul 12, 10, 01:52 PM
[=REPOSITORY=MODIFIEDON=]	Jul 13, 10, 11:37 PM
OWNER	Ti.mo
AVAILABLE SIZES	Square (75 x 75) Thumbnail (100 x 67) Small (240 x 160) Medium (500 x 333) Large (1024 x 683)
TAGS	england, eurostar, geo, gps, iphone, iphone 4, location, locative, london, mobile, motion-x, phone, screen, tracking, travel, travelling, uk, window
LICENSE	All Rights Reserved

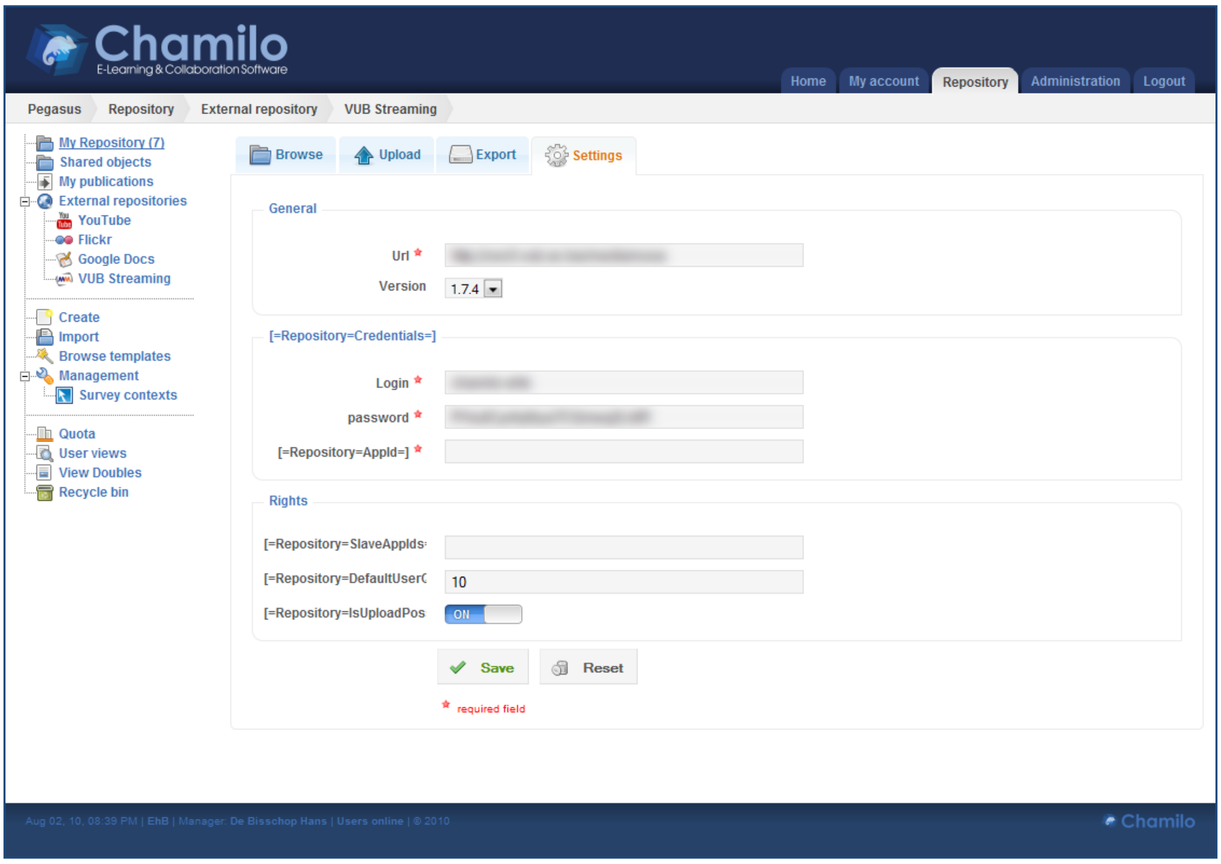


Ppt0000002

TITLE	Ppt0000002
UPLOADED ON	Jul 09, 10, 08:19 PM
[=REPOSITORY=MODIFIEDON=]	Jul 09, 10, 08:19 PM
OWNER	roderidder@gmail.com
[=REPOSITORY=LASTVIEWED=]	Jul 30, 10, 12:12 PM
[=REPOSITORY=LASTMODIFIEDBY=]	roderidder@gmail.com

Settings

Most external repositories will require some kind of configuration, ranging from authentication credentials, over user quota to a developer access key. A generic form (with dynamic properties) was implemented to make the management of these settings as easy as possible.



General

Url *	[redacted]
Version	1.7.4

[=Repository=Credentials=]

Login *	[redacted]
password *	[redacted]
[=Repository=AppId=] *	

Rights

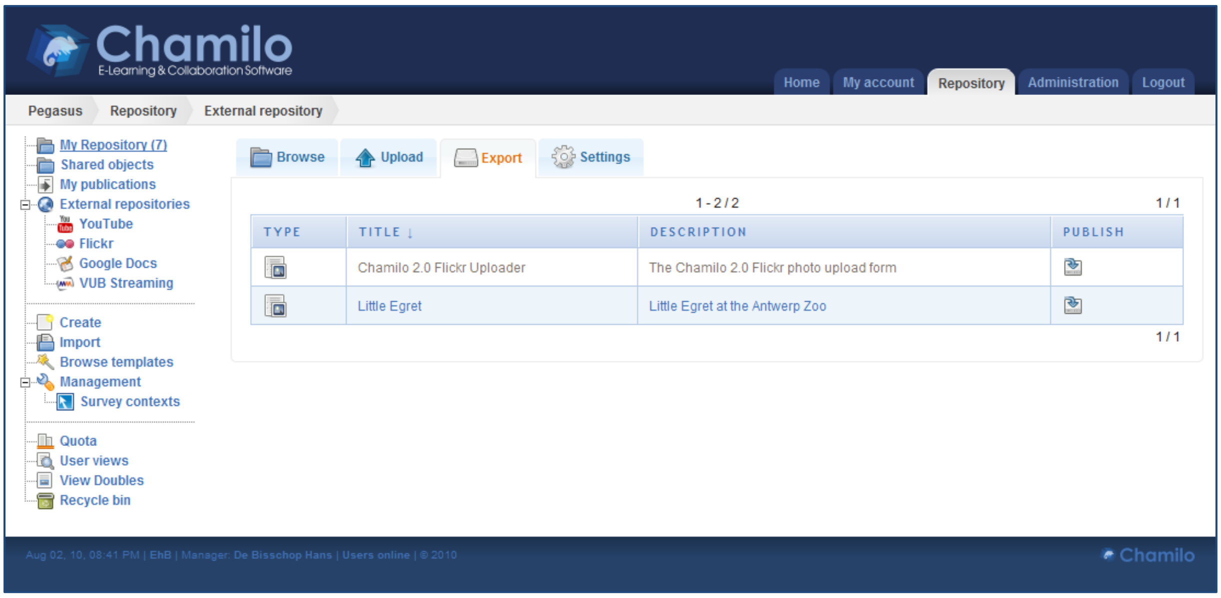
[=Repository=SlaveAppIds	
[=Repository=DefaultUserC	10
[=Repository=IsUploadPos	ON

* required field

Aug 02, 10, 08:39 PM | EhB | Manager: De Bisschop Hans | Users online | © 2010

Export

Depending on the repository and your rights, you may be able to export objects from your local repository to the external repository. Just select one of your objects to export them.



1 - 2 / 2 1 / 1

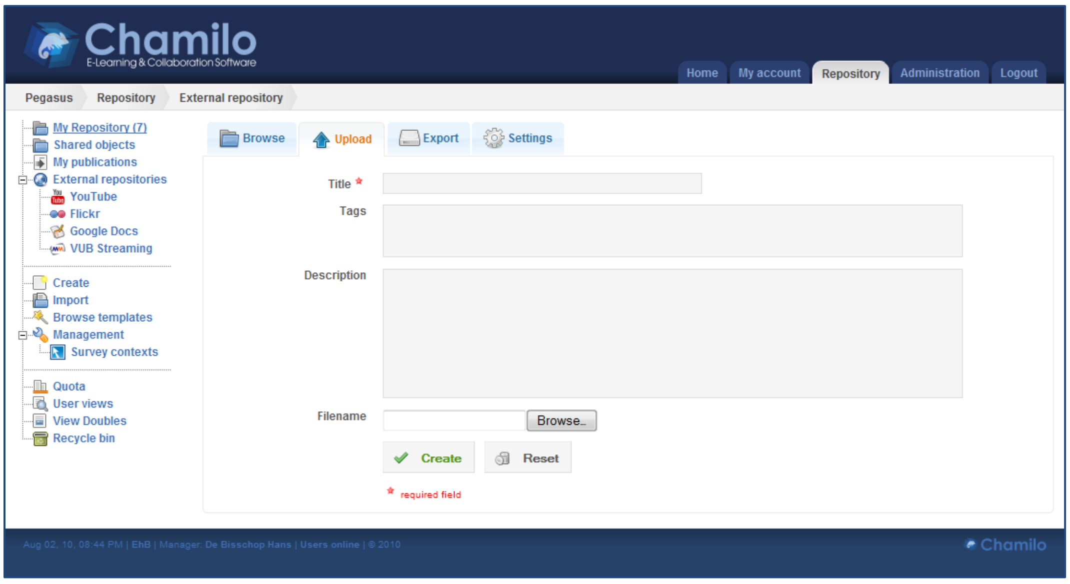
TYPE	TITLE ↓	DESCRIPTION	PUBLISH
	Chamilo 2.0 Flickr Uploader	The Chamilo 2.0 Flickr photo upload form	
	Little Egret	Little Egret at the Antwerp Zoo	

1 / 1

Aug 02, 10, 08:41 PM | EhB | Manager: De Bisschop Hans | Users online | © 2010

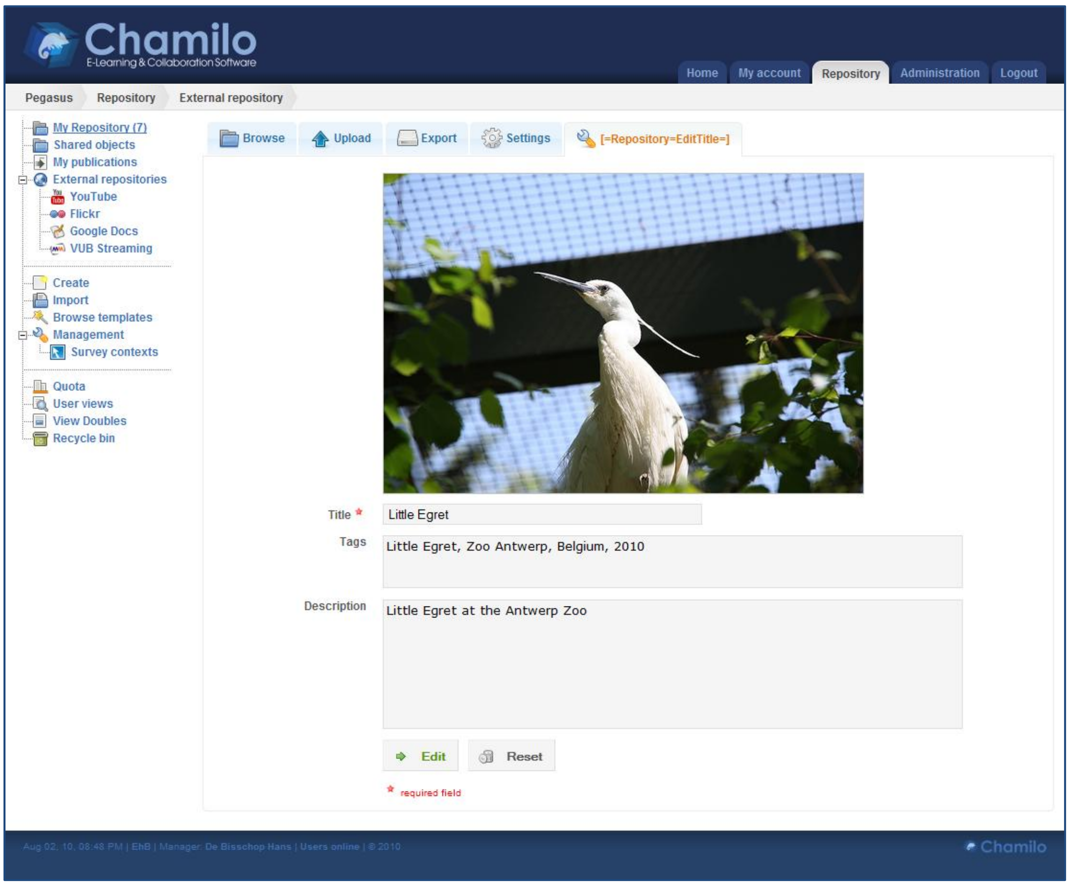
Upload

In some cases you might want to upload content directly to the external repository. Depending on the implementation and the possibilities offered by the API, an upload component may be available.



Editor

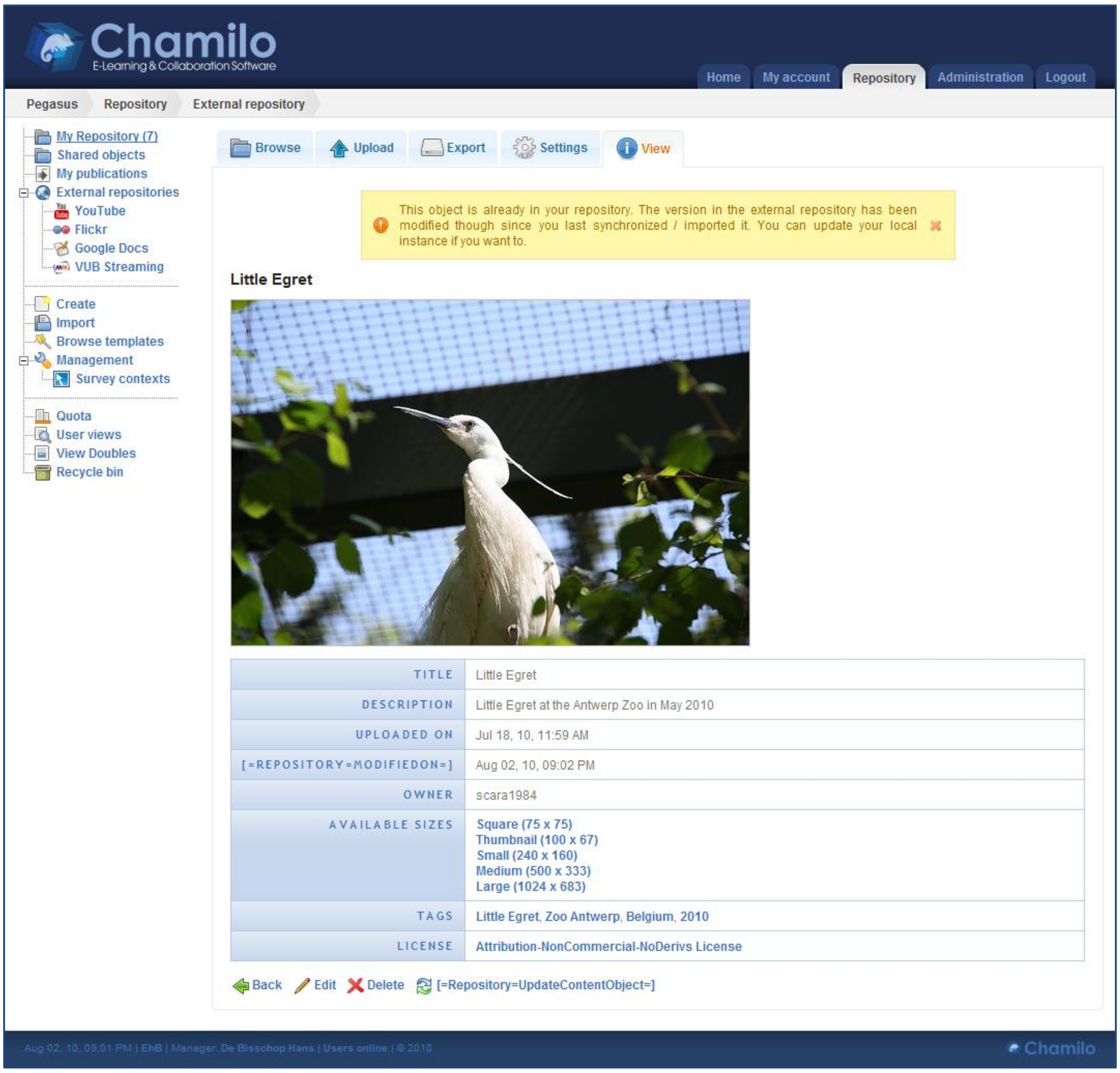
Properties of the external object might be editable; if and when this is the case the editor will be available and will allow you to manipulate one or more of the objects basic and additional properties.



Title *: Little Egret
Tags: Little Egret, Zoo Antwerp, Belgium, 2010
Description: Little Egret at the Antwerp Zoo

Synchronization

As soon as you import / link an external object with the local repository the two objects can mutually check whether a newer version is available remotely or whether your local version supersedes the remote version. Depending on which of the options is your case you will be able to synchronize the changes.



This object is already in your repository. The version in the external repository has been modified though since you last synchronized / imported it. You can update your local instance if you want to.

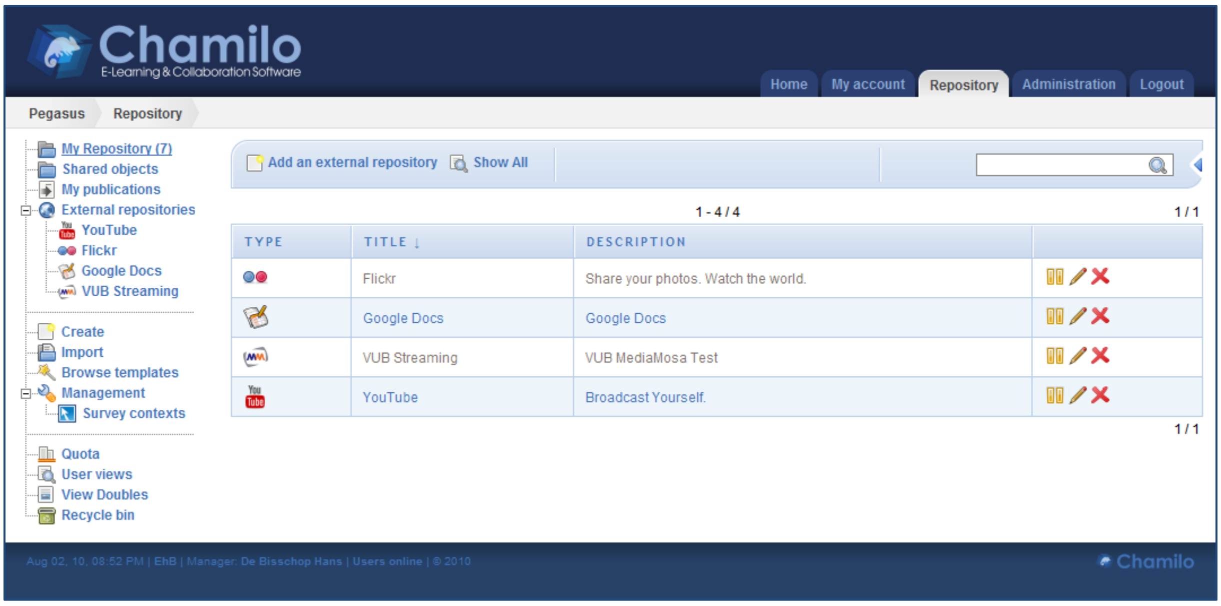
Little Egret

TITLE	Little Egret
DESCRIPTION	Little Egret at the Antwerp Zoo in May 2010
UPLOADED ON	Jul 18, 10, 11:59 AM
[=REPOSITORY=MODIFIEDON=]	Aug 02, 10, 09:02 PM
OWNER	scara1984
AVAILABLE SIZES	Square (75 x 75) Thumbnail (100 x 67) Small (240 x 160) Medium (500 x 333) Large (1024 x 683)
TAGS	Little Egret, Zoo Antwerp, Belgium, 2010
LICENSE	Attribution-NonCommercial-NoDerivs License

Aug 02, 10, 09:01 PM | EhB | Manager: De Bisschop Hans | Users online | © 2010

External Repository Instance Management

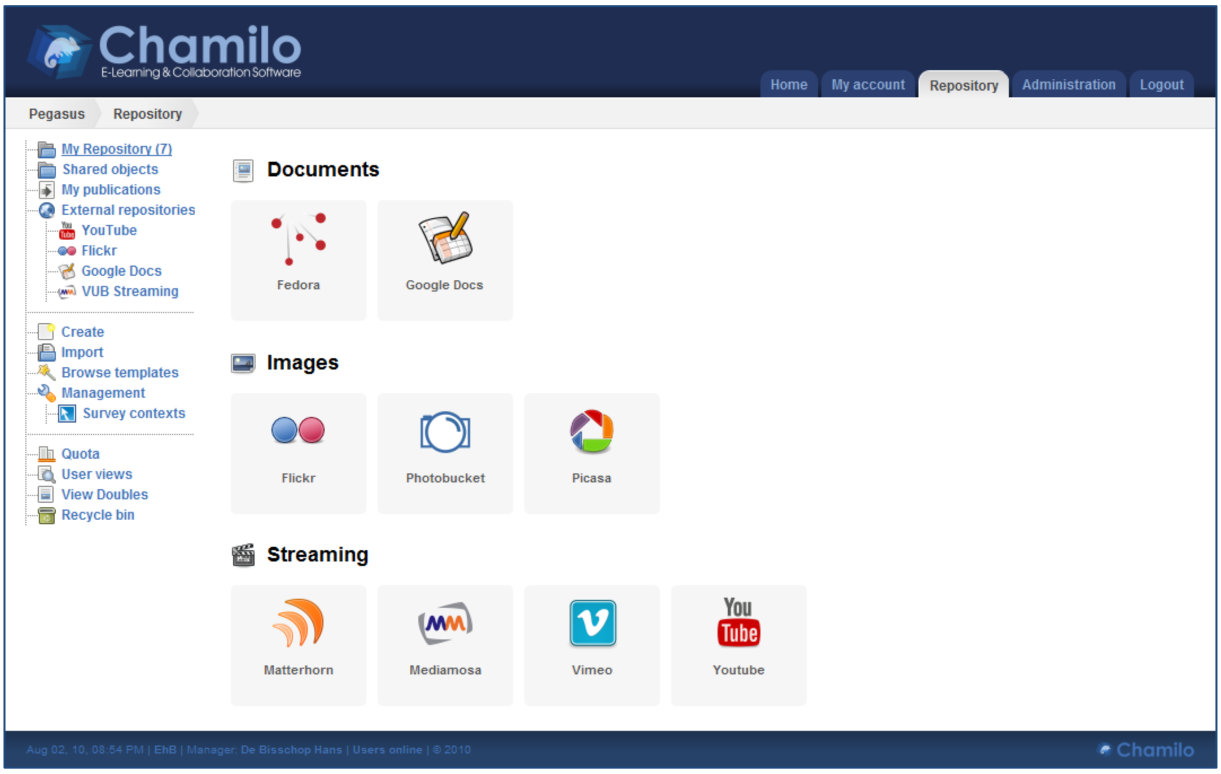
As mentioned before some repositories can have more than one instance active at any one time. To be able to manage these instances and to activate or deactivate existing instances, a management module was added to the repository for administrators. It allows you to add, edit and configure instances whenever necessary.



TYPE	TITLE ↓	DESCRIPTION	
	Flickr	Share your photos. Watch the world.	
	Google Docs	Google Docs	
	VUB Streaming	VUB MediaMosa Test	
	YouTube	Broadcast Yourself.	

Adding an external repository instance

To facilitate the selection of a repository type, the different types were divided into categories based on their primary goal.



That's all folks!

This concludes the quick visual overview of most of the generic visual components of the external repository manager. But these components are just the tip of the iceberg. They probably cover 90% of what most end-users would like to use, but, and there always is one, some functionality simply can't be generalized, even though it might be essential in the context of the repository.

Developers should not experience serious problems extending the framework to implement these additional components. So if you are a developer: read on and discover what makes this framework tick.

The blue pill or the red pill

Enough already with the easy stuff, let's move on to the more technical part. The intention of this article is not to explain in full detail how the framework works, but to make sure you have a basic understanding of its structure and what you need to implement to create a working external repository manager implementation yourself.

Storage units

Settings and properties of individual repository instances as well as synchronization data of external repository objects is stored persistently, using the platform's data managers. In most cases this will be a DBMS, so that's the case which will be reviewed here.

Field	Type	Extra
id	int(10)	The numeric identifier of the instance
title	char(50)	The title
description	text	The description
type	char(50)	The type of the instance, e.g. youtube, mediamosa, etc.
enabled	tinyint(3)	Whether or not the instance is active
created	int(10)	The creation date as a unix timestamp
modified	int(10)	The last modification date as a unix timestamp

Fig. 7 – Storage unit structure for external_repository

Field	Type	Extra
id	int(10)	The numeric identifier of the setting
external_repository_id	int(10)	The numeric identifier of the repository instance
variable	varchar(255)	The name of the setting
value	text	The setting's value
user_setting	tinyint(3)	Whether ot not the setting is user-specific

Fig. 8 – Storage unit structure for external_repository_setting

Field	Type	Extra
id	int(10)	The numeric identifier of the user_setting
user_id	int(10)	The numeric identifier of the user
setting_id	int(10)	The numeric identifier of the setting
value	text	The user setting's value

Fig. 9 – Storage unit structure for external_repository_user_setting

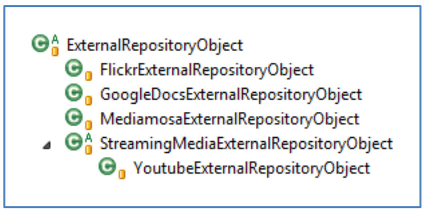
Field	Type	Extra
id	int(10)	The numeric identifier of the synchronization information object
content_object_id	int(10)	The numeric identifier of the content object

content_object_timestamp	int(10)	The unix timestamp of the content object upon creation
external_repository_id	int(10)	The numeric identifier of the external repository
external_repository_object_id	varchar(255)	The numeric identifier of the external content object
external_repository_object_timestamp	int(10)	The unix timestamp of the external content object upon creation
created	int(10)	The unix timestamp of the date the synchronization was initially done
modified	int(10)	The unix timestamp of the most recent synchronization

Fig. 10 – Storage unit structure for external_repository_sync

You might have come to expect that there would also be a table for the actual external repository objects. But in doing so we would at the same time be creating a ridiculously huge table and we would be defeating the entire point of linking to the external repository in the first place.

External objects



- ExternalRepositoryObject
  - FlickrExternalRepositoryObject
  - GoogleDocsExternalRepositoryObject
  - MediamosaExternalRepositoryObject
  - StreamingMediaExternalRepositoryObject
    - YoutubeExternalRepositoryObject

Fig. 11 – Type hierarchy of the external repository objects

As mentioned before, external objects solely exist in memory. They are simply a means to map a variety of properties on an external system to some generic format Chamilo 2.0 can understand and handle or display in its various components.

Most methods will also spawn fatal errors if the object being passed on to them is not an instance of an external repository object or one of its extensions.

The format of these external repository objects can vary from repository to repository. To keep things manageable the general External Repository Object has a number of basic properties, which every external repository should be able to provide one way or another.

Property	Type	Description
Id	String	The identifier of the object in the external repository
External Repository Id	Integer	The numeric identifier of the external repository instance
Title	String	The title of the object
Description	String	A description, if available
Owner	String	The identifier of the original owner
Created	Integer	A unix timestamp representing the date the object was created in the external repository
Modified	Integer	A unix timestamp representing the date the object was last modified. Can be identical to Created if modifying is not supported
Type	String	The type of the object in the context of the repository. E.g. for Flickr this could be jpg, png, bmp, etc.
Rights	Array	An array containing a few basic rights used by the external repository manager to determine what a user can or can't do

Fig. 12 – Default properties of an external repository object

Depending on the implementation additional properties might be available. A few examples:

Property	Type	Description
Category	String	The category of the video on YouTube
Tags	Array	An array of tags assigned to the video by the uploader

Fig. 13 – Additional properties for a YouTube object

Property	Type	Description
Urls	Array	The URL's for the different available sizes of the photo
License	Array	The license which applies to the photo
Tags	Array	Tags as defined by the uploader

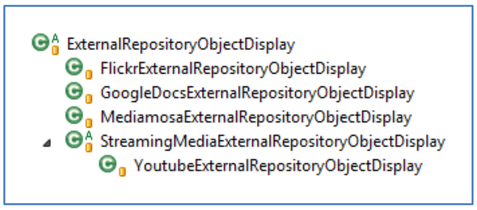
Fig. 14 – Additional properties for a Flickr object

Property	Type	Description
Viewed	Integer	The unix timestamp of the date the document was last viewed
Content	Integer	A link to the document's content
Modifier Id	String	The identifier of the user which last modified the document

Fig. 15 – Additional properties for a Google Docs object

External Repository Object Display

Considering that an external repository object is not all that different from a regular content object as far as the end-user is concerned, it might not come as a surprise that it also has a dedicated display class.



```
ExternalRepositoryObjectDisplay
    FlickrExternalRepositoryObjectDisplay
    GoogleDocsExternalRepositoryObjectDisplay
    MediamosaExternalRepositoryObjectDisplay
    StreamingMediaExternalRepositoryObjectDisplay
        YoutubeExternalRepositoryObjectDisplay
```

Fig. 16 – Type hierarchy of the external repository object display

The display class provides an HTML representation of the actual object. Per default it will (try to) show a title, a preview and a list of properties. Needless to say that this can be changed or expanded upon whenever the need arises. We'll look at a specific example later on, but basically you could have a functional dummy `MyExternalRepositoryObjectDisplay` which extends the default `ExternalRepositoryObjectDisplay` class.

External Repository Manager

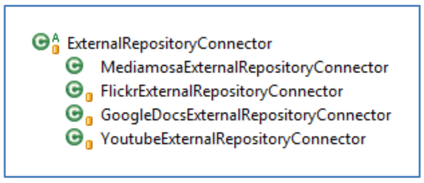
At the root of the entire framework we find the `ExternalRepositoryManager`, which all specific implementations will extend. This superclass contains basic functionality which:

- handles the displaying of headers and footers
- redirects the requests of the manager components to a specific implementation
- defines a number of default actions, available for external repository objects
- defines the views which will be available in the browser (table, gallery, slideshow)
- provides a factory to create instances of specific implementations

External Repository Connector

Here comes the tricky one. As you may have guessed from the title this is the class which will actually allow us to connect to the external repository and retrieve information from it or manipulate it. Why is this tricky? For the very simple reason that there's hardly any two systems out there which are the same. They may use the same communication protocol, they may even use the same architecture ...

but there will almost always be very specific requirements for specific connectors. That's also why there is no default implementation of a connector.



```
ExternalRepositoryConnector
    MediamosaExternalRepositoryConnector
    FlickrExternalRepositoryConnector
    GoogleDocsExternalRepositoryConnector
    YoutubeExternalRepositoryConnector
```

Fig. 17 – Type hierarchy of the external repository connector

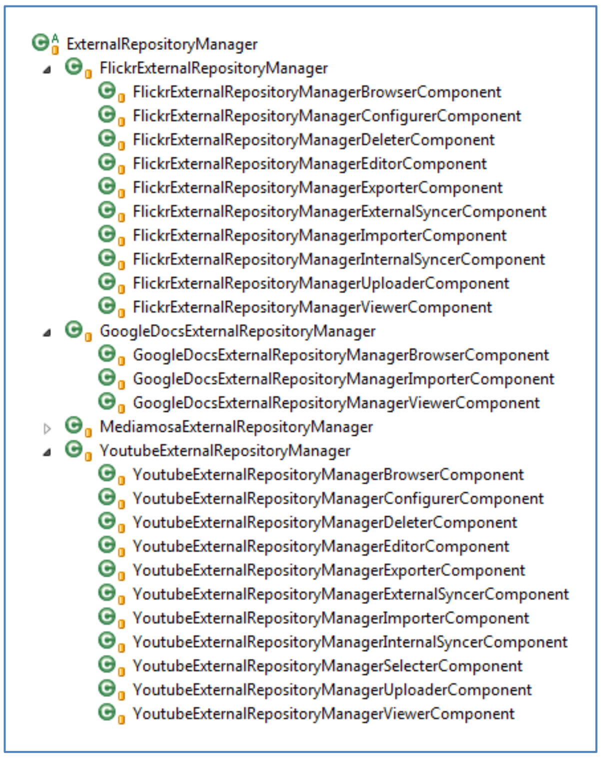
We do however have an external repository connector interface which all connectors should implement. This makes it a lot easier for the developer to know which methods he should implement if he wants to enable all basic functionality for his or her specific external repository. Apart from these few methods, the layout of these classes could be very different indeed, depending on what they're actually connecting with.

Some libraries will be provided though to simplify connecting to certain repositories by means of an open standard and/or well defined architecture or protocol. The library for calling RESTful web services is one such example.

In a lot of cases though, you'll simply have to provide an interface from Chamilo 2.0 to a client library which is readily available for usage in PHP projects. E.g. Zend Gdata for the Google APIs and phpFlickr for Flickr.

External Repository Manager Components

When implementing the architecture for the external repository manager, a few difficult choices had to be made concerning flexibility, extensibility and ease-of-development. To keep things really easy for the developers we could have provided a default set of components which could not be extended or modified in an easy way. It would basically reduce the development of new implementations to writing a connector as mentioned before.



```
ExternalRepositoryManager
  ◢ FlickrExternalRepositoryManager
        FlickrExternalRepositoryManagerBrowserComponent
        FlickrExternalRepositoryManagerConfigurerComponent
        FlickrExternalRepositoryManagerDeleterComponent
        FlickrExternalRepositoryManagerEditorComponent
        FlickrExternalRepositoryManagerExporterComponent
        FlickrExternalRepositoryManagerExternalSyncerComponent
        FlickrExternalRepositoryManagerImporterComponent
        FlickrExternalRepositoryManagerInternalSyncerComponent
        FlickrExternalRepositoryManagerUploaderComponent
        FlickrExternalRepositoryManagerViewerComponent
  ◢ GoogleDocsExternalRepositoryManager
        GoogleDocsExternalRepositoryManagerBrowserComponent
        GoogleDocsExternalRepositoryManagerImporterComponent
        GoogleDocsExternalRepositoryManagerViewerComponent
  ▷ MediamosaExternalRepositoryManager
  ◢ YoutubeExternalRepositoryManager
        YoutubeExternalRepositoryManagerBrowserComponent
        YoutubeExternalRepositoryManagerConfigurerComponent
        YoutubeExternalRepositoryManagerDeleterComponent
        YoutubeExternalRepositoryManagerEditorComponent
        YoutubeExternalRepositoryManagerExporterComponent
        YoutubeExternalRepositoryManagerExternalSyncerComponent
        YoutubeExternalRepositoryManagerImporterComponent
        YoutubeExternalRepositoryManagerInternalSyncerComponent
        YoutubeExternalRepositoryManagerSelecterComponent
        YoutubeExternalRepositoryManagerUploaderComponent
        YoutubeExternalRepositoryManagerViewerComponent
```

Fig. 18 – Type hierarchy of the external repository manager and a few basic implementations

But that means we would not be able to fully utilize a repository's additional properties or quite a bit of the functionality of them may offer on top of what we consider the basic set of tools. Apart from that, it's just not the Chamilo 2.0-way of doing things. If a few semi-dummy classes allow the platform to have a lot more potential than it would be rather silly to not allow for it.

So we set out to identify the basic set of components an external repository might offer:

- browsing objects
- configuring the connection / instance
- deleting an object
- uploading new content
- synchronization (bi-directional)
- importing content into Chamilo
- viewing an object

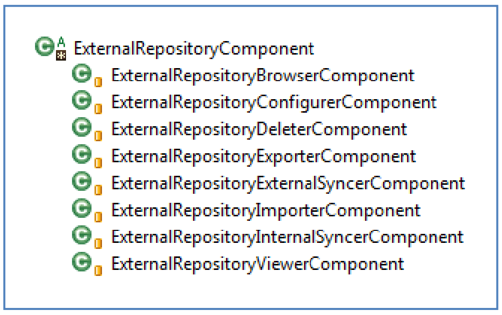

ExternalRepositoryComponent
ExternalRepositoryBrowserComponent
ExternalRepositoryConfigurerComponent
ExternalRepositoryDeleterComponent
ExternalRepositoryExporterComponent
ExternalRepositoryExternalSyncerComponent
ExternalRepositoryImporterComponent
ExternalRepositoryInternalSyncerComponent
ExternalRepositoryViewerComponent

Fig. 19 – Type hierarchy of the generic external repository manager components

Each of these components depends on the external repository manager, its repository-specific implementation and a correctly implemented external repository connector to make it work. E.g. if you want to use the general viewer, you will have to implement the `retrieve_external_repository_object` method in your external repository connector. If you don't you'll have to write your own viewing component. However, if you do adding a viewer-component will be nothing more than a few lines of code.

The main advantage is obviously that we are still able to implement components which are specific for a particular repository, but at the same time we're also able to use some general, fully functional, components as well. Isn't that great?

Let the games begin: The Flickr external repository manager

Enough theory, let's have a look at a practical example which is already available in the codebase[8]. Before we can actually start to develop the manager, there are a few things which we should check out.

The Flickr API

In the case of Flickr a pretty extensive API[9] is readily available and very well documented. You may want to have a look at it, before reading on. The API supports REST, XML-RPC and SOAP for requests and can return responses as REST, XML-RPC, SOAP, JSON or serialized PHP data. Flickr's popularity is clearly demonstrated by the fact that client libraries exist for 14 different programming languages, including PHP.

Remembering that we don't want to reinvent the wheel, we had a look at the 3 available client libraries for PHP. Of the 3, phpFlickr[10] proved to be the most flexible and actively maintained one. So no real need to worry about the actual API, just a matter of having a look at the PHP library. Even so it's still a good idea to at the very least have a basic understanding of the actual API. It will only make it easier for you to find out which methods to call, what parameters to pass and what Flickr expects them to be value-wise.

Settings

Most external repositories will require you to provide some kind of authentication, for the application as well as the user. If you want to know more about this authentication process, just read the relevant pages[11] on the Flickr website. It's definitely worth a read as the principle will be more or less the same for quite a few external services. To allow your Flickr manager to work, you'll need to store a developer key and a secret somewhere. Both strings are unique for your application, which is

[8] http://code.google.com/p/chamilo/source/browse/?repo=chamilo
[9] http://www.flickr.com/services/api/
[10] http://phpflickr.com/
[11] http://www.flickr.com/services/api/misc.userauth.html

a Chamilo installation in our case. On top of that, if a user successfully completes the initial authentication, you'll have to store the obtained token somewhere.

These three keys are perfect examples of settings for an external repository manager and will be treated as such. How do we handle settings on the rest of the platform? By defining an XML file which is parsed upon installation / registration of the application and which will be used to render the settings form. That works rather well, so why would we do things differently here?

This results in the following settings file for our Flick manager:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<application name="flickr">
   <settings>
      <category name="credentials">
         <setting name="key" field="text" default="">
            <validations>
               <validation rule="required" message="ThisFieldIsRequired" />
            </validations>
         </setting>
         <setting name="secret" field="text" default="">
            <validations>
               <validation rule="required" message="ThisFieldIsRequired" />
            </validations>
         </setting>
         <setting name="session_token" field="text" default="" user_setting="1"/>
      </category>
   </settings>
</application>
```

Keep in mind that, as is the case with application settings, you'll also need to define a settings connector for settings that have a dynamic list of options.

Properties

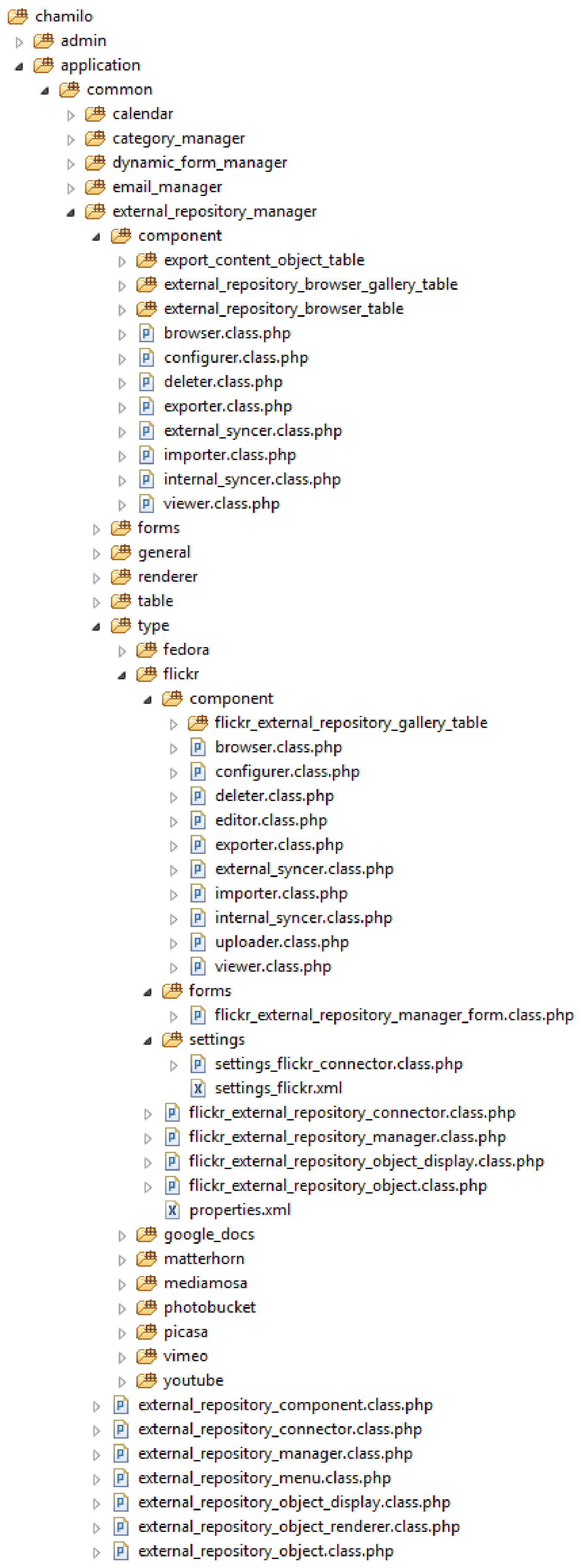
First settings and now properties ... what the hell? The properties file is also XML-based and right now stores 2 settings or properties which are related to the management of repository instances.

- What type of external repository is it? (Documents, streaming media, images, etc.)
- Does the repository allow multiple instances? (e.g. YouTube vs. Opencast Matterhorn)

In XML format this looks like this:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<repository name="mediamosa">
      <properties>
            <property name="section" value="streaming" />
            <property name="multiple" value="1" />
      </properties>
</repository>
```

The multiple-property can set to 0 or dropped altogether if the repository does not allow multiple instances.



```
chamilo
  admin
  application
    common
      calendar
      category_manager
      dynamic_form_manager
      email_manager
      external_repository_manager
        component
          export_content_object_table
          external_repository_browser_gallery_table
          external_repository_browser_table
          browser.class.php
          configurer.class.php
          deleter.class.php
          exporter.class.php
          external_syncer.class.php
          importer.class.php
          internal_syncer.class.php
          viewer.class.php
        forms
        general
        renderer
        table
        type
          fedora
          flickr
            component
              flickr_external_repository_gallery_table
              browser.class.php
              configurer.class.php
              deleter.class.php
              editor.class.php
              exporter.class.php
              external_syncer.class.php
              importer.class.php
              internal_syncer.class.php
              uploader.class.php
              viewer.class.php
            forms
              flickr_external_repository_manager_form.class.php
            settings
              settings_flickr_connector.class.php
              settings_flickr.xml
            flickr_external_repository_connector.class.php
            flickr_external_repository_manager.class.php
            flickr_external_repository_object_display.class.php
            flickr_external_repository_object.class.php
            properties.xml
          google_docs
          matterhorn
          mediamosa
          photobucket
          picasa
          vimeo
          youtube
        external_repository_component.class.php
        external_repository_connector.class.php
        external_repository_manager.class.php
        external_repository_menu.class.php
        external_repository_object_display.class.php
        external_repository_object_renderer.class.php
        external_repository_object.class.php
```

General overview

As a reference for the general structure of the external repository manager we've included a quick overview of most of the relevant files to the left.

This includes, but is not limited to generic components, implementations and more in particular the Flickr implementation, the external repository manager superclasses, and the location of all these files within the general Chamilo folder structure.

The only relevant files which have not been included in this overview are those related to management of repository instances, since that functionality does not have to be extended or modified to implement new external repositories.

The data class and additional properties

As listed in Fig. 14 we decided to support three additional properties for the Flickr repository objects and implemented them as such.

```php
<?php
require_once dirname(__FILE__) . '/../../external_repository_object.class.php';

class FlickrExternalRepositoryObject extends ExternalRepositoryObject
{
    const OBJECT_TYPE = 'flickr';

    const PROPERTY_URLS = 'urls';
    const PROPERTY_LICENSE = 'license';
    const PROPERTY_TAGS = 'tags';

    const SIZE_SQUARE = 'square';
    const SIZE_THUMBNAIL = 'thumbnail';
    const SIZE_SMALL = 'small';
    const SIZE_MEDIUM = 'medium';
    const SIZE_LARGE = 'large';
    const SIZE_ORIGINAL = 'original';

    static function get_default_property_names()
    {
        return parent :: get_default_property_names(array(self :: PROPERTY_URLS,
            self :: PROPERTY_LICENSE, self :: PROPERTY_TAGS));
    }

    static function get_object_type()
    {
        return self :: OBJECT_TYPE;
    }
}
?>
```

The make sure the object knows about these properties, we need to extend the method `get_default_property_names()` by calling that method in the parent and passing on an array of additional properties. Additional constants in the class include `OBJECT_TYPE` which represents the `repository_type` (as is used, among others, to display relevant icons) and several sizes. These sizes are related to the urls-property, an array which contains links to all available sizes of a particular photo or image. Since these sizes are predefined and used as such by the Flickr API, we're adding a few constants to help us out.

... and of course the class contains getters and setters for the properties, as well as some helper-methods to make the data easier to handle. E.g. retrieve the exact dimensions of a particular named size of the photo.

Going visual

Remember that general external repository object display class? Its purpose was to make it very easy to display previews and additional properties of the external repository object. If your additional properties were well considered you should have little or no problems in getting a preview for your object. The helper methods mentioned in the previous paragraph, partially own their existence to the fact they were needed for the display logic, but not necessarily limited or exclusive for said display.

```php
<?php
require_once dirname(__FILE__) .
              '/../../external_repository_object_display.class.php';

class FlickrExternalRepositoryObjectDisplay extends ExternalRepositoryObjectDisplay
{

    function get_display_properties()
    {
        $object = $this->get_object();

        $properties = parent :: get_display_properties();
        $properties[Translation :: get('AvailableSizes')]
          = $object->get_available_sizes_string();
        $properties[Translation :: get('Tags')] = $object->get_tags_string();
        $properties[Translation :: get('License')] = $object->get_license_string();

        return $properties;
    }

    function get_preview($is_thumbnail = false)
    {
        $object = $this->get_object();
        $size = ($is_thumbnail ? FlickrExternalRepositoryObject :: SIZE_SQUARE :
          FlickrExternalRepositoryObject :: SIZE_MEDIUM);
        $class = ($is_thumbnail ? 'thumbnail' : 'with_border');

        $html = array();
        $html[] = '<img class="' . $class . '" src="' .
          $object->get_url($size) . '" />';
        return implode("\n", $html);
    }
}
?>
```

The method `get_display_properties()` gets extended just like `get_default_property_names()` in the data class. Get preview, which returns a “no preview”-widget per default, gets overwritten and returns an image tag which uses one of the URI’s as returned by the Flickr API. Nothing too difficult up until now, right?

Extending the external repository manager

To get a working extension we need to implement a Flickr external repository manager which implements a few mandatory methods. They have been defined as abstract by the external repository manager superclass and have to be implemented by all non-abstract children.

Method	Returns	Description
validate_settings	Boolean	Verifies the settings and redirects the administrator to the settings form if something is wrong. Displays a message for all other users.
run	Void	Starts a component based on the query parameters. Identical to all other application structures in Chamilo 2.0.
get_external_repository_object_viewing_url	String	The generic browser needs a URL to link to the details page of each object. Parameters depend on the implementation, so we need to implement this locally.
get_menu_items	Array	Menu-items in the default array format used throughout the system. In the case of Flickr an example of a menu item is “My Photos”.

get_content_object_type_conditions	Condition	External repository objects aren't necessarily mapped to an exclusive local object. E.g. Flickr photos become documents. The other way round is not always true. A text document is also a Chamilo document, but definitely not suitable for Flickr. This is why we need additional conditions to filter the export list.

Fig. 20 – Basic methods required to make your external repository manager work

Additionally some methods can be overwritten to enable or disable default functionality.

Method	Returns	Description
get_external_repository_actions	Array	What can we do with a particular object? A mono-dimensional array containing all relevant ACTION-constants
get_available_renderers	Array	By default only the table view will be enabled for the browser component. Overwrite this method to enable more or other views. In this specific case we also wanted to enable the gallery and slideshow views.

Fig. 21 – Additional methods which can be overwritten

A code snippet to illustrate the aforementioned methods in this particular context. Please check the actual codebase for a working example as this snippet leaves out some methods which are outside the scope of this document or which are dependencies inherited by the external repository manager from its own parent(s).

```php
<?php
require_once dirname(__FILE__) . '/flickr_external_repository_connector.class.php';

class FlickrExternalRepositoryManager extends ExternalRepositoryManager
{
    function validate_settings()
    {
        $key = ExternalRepositorySetting :: get('key');
        $secret = ExternalRepositorySetting :: get('secret');

        if (! $key || ! $secret)
        {
            return false;
        }
        return true;
    }

    function get_external_repository_object_viewing_url(ExternalRepositoryObject
                                                         $object)
    {
        $parameters = array();
        $parameters[self :: PARAM_EXTERNAL_REPOSITORY_MANAGER_ACTION]
                    = self :: ACTION_VIEW_EXTERNAL_REPOSITORY;
        $parameters[self :: PARAM_EXTERNAL_REPOSITORY_ID]
                    = $object->get_id();

        return $this->get_url($parameters);
    }

    function get_menu_items()
    {
        $menu_items = array();
        $my_photos = array();
        $my_photos['title'] = Translation :: get('MyPhotos');
        $my_photos['url'] = $this->get_url(array(self :: PARAM_FEED_TYPE => self ::
                                FEED_TYPE_MY_PHOTOS), array(ActionBarSearchForm ::
                                PARAM_SIMPLE_SEARCH_QUERY));
        $my_photos['class'] = 'user';
        $menu_items[] = $my_photos;

        return $menu_items;
    }

    function get_external_repository_actions()
    {
        $actions = array(self :: ACTION_BROWSE_EXTERNAL_REPOSITORY, self ::
                    ACTION_UPLOAD_EXTERNAL_REPOSITORY, self ::
                    ACTION_EXPORT_EXTERNAL_REPOSITORY);

        $is_platform = $this->get_user()->is_platform_admin() &&
                        (count(ExternalRepositorySetting :: get_all()) > 0);

        if ($is_platform)
        {
            $actions[] = self :: ACTION_CONFIGURE_EXTERNAL_REPOSITORY;
        }

        return $actions;
    }

    function run()
    {
        $parent = $this->get_parameter(ExternalRepositoryManager ::
                    PARAM_EXTERNAL_REPOSITORY_MANAGER_ACTION);

        switch ($parent)
        {
            case ExternalRepositoryManager :: ACTION_BROWSE_EXTERNAL_REPOSITORY :
                $component = $this->create_component('Browser', $this);
                break;
        }

        $component->run();
    }

    function get_available_renderers()
    {
        return array(ExternalRepositoryObjectRenderer :: TYPE_GALLERY,
                     ExternalRepositoryObjectRenderer :: TYPE_SLIDESHOW,
                     ExternalRepositoryObjectRenderer :: TYPE_TABLE);
    }

    function get_content_object_type_conditions()
    {
        $image_types = Document :: get_image_types();
        $image_conditions = array();
        foreach ($image_types as $image_type)
        {
            $image_conditions[] = new PatternMatchCondition(Document ::
                                  PROPERTY_FILENAME, '*.' . $image_type, Document
                                  :: get_type_name());
        }

        return new OrCondition($image_conditions);
    }
}
?>
```

As you may have noticed the method `validate_settings` makes a static get-call to the external repository setting class. A similar static function exists for the external repository user setting class. Their goal is to facilitate the retrieval of both instance and instance user settings. Both take a number of parameters of which only the parameter variable is required. `ExternalRepositorySetting :: get` accepts one additional parameter: `external_repository_id`, which would be the numerical identifier of the repository instance we're retrieving a setting for. If we don't pass this parameter the current instance, as defined by the query parameters, is used.

The first optional parameter for `ExternalRepositoryUserSetting :: get` is identical to the one described for the `ExternalRepositorySetting`, being the `external_repository_id`. Considering that it is a user setting though, the second optional parameter is a `user_id`, a numerical identifier of a Chamilo 2.0 user. If not passed on the system will use the `user_id` of the user currently logged in.

Adding a few components

Remember how we added some generic components? Well, now that you've defined your external object, its display class and the actual manage extension, it's time to add an actual functional component and the browser would be the first and most obvious choice.

Since the Flickr browser is not that special in terms of exceptions to the standard browser we defined, implementing is easy to say the least. Just have a look:

```php
<?php
class FlickrExternalRepositoryManagerBrowserComponent extends
       FlickrExternalRepositoryManager
{

    function run()
    {
        $browser = ExternalRepositoryComponent ::
                     factory(ExternalRepositoryComponent :: BROWSER_COMPONENT,
                     $this);
        $browser->run();
    }
}
?>
```

Time for our catchphrase: isn't that great? Quite a few of the other generic components are equally easy to implement. The default viewer and configuration component are even exactly as easy. There's only one small catch concerning gallery tables. Given the visual nature of said tables it's almost impossible to completely generalize them.

So if and when you decide you need a gallery view for your browser, don't forget to implement a gallery table extension for it. It's most important component is by far the cell renderer, which actually makes sure the correct information concerning each object is displayed in each cell.

A similar abstraction might be added at a later date for the general tables and the current structure for these browser tables is by no means set in stone. It will no doubt evolve as soon as more external repositories are integrated and other use-cases surface.

```php
<?php
require_once dirname(__FILE__) .
    '/../../../../table/default_external_repository_gallery_object_table_cell_render
    er.class.php';

class FlickrExternalRepositoryGalleryTableCellRenderer extends
        DefaultExternalRepositoryGalleryObjectTableCellRenderer
{
    private $browser;

    function FlickrExternalRepositoryGalleryTableCellRenderer($browser)
    {
        parent :: __construct();
        $this->browser = $browser;
    }

    function get_cell_content(ExternalRepositoryObject $object)
    {
        $html = array();
        $display = ExternalRepositoryObjectDisplay :: factory($object);
        $html[] = '<h4>' . Utilities :: truncate_string($object->get_title(), 25) .
                  '</h4>';
        $html[] = '<a href="' . $this->browser-
                  >get_external_repository_object_viewing_url($object) . '">' .
                  $display->get_preview(true) . '</a>';

        if ($object->get_description())
        {
            $html[] = '<br/>';
            $html[] = '<i>' . Utilities :: truncate_string($object-
                      >get_description(), 100) . '</i>';
            $html[] = '<br/>';
        }

        return implode("\n", $html);
    }

    function get_modification_links($object)
    {
        $toolbar = new Toolbar(Toolbar :: TYPE_VERTICAL);
        $toolbar->add_items($this->browser-
                >get_external_repository_object_actions($object));
        return $toolbar->as_html();
    }
}
?>
```

But obviously not everything can be reduced to an extension which is this simple; some components do require additional development to make them work 100%. Let's have a look at a few examples.

Deleter component

For the most part, deleting is handled automatically, but what to do once we've deleted the object? We'll want to redirect the user somewhere. Depending on the external repository and the context (e.g. selected category) we may want to do something completely different. That's why the actual redirect is placed in the delete component of the Flickr extension and not in the general component. The same principle is applied to the exporter component.

```php
<?php
class FlickrExternalRepositoryManagerDeleterComponent extends
       FlickrExternalRepositoryManager
{

    function run()
    {
        $deleter = ExternalRepositoryComponent ::
          factory(ExternalRepositoryComponent :: DELETER_COMPONENT, $this);
        $deleter->run();
    }

    function delete_external_repository_object($id)
    {
        $success = parent :: delete_external_repository_object($id);
        if ($success)
        {
            $parameters = $this->get_parameters();
            $parameters[ExternalRepositoryManager ::
               PARAM_EXTERNAL_REPOSITORY_MANAGER_ACTION] =
               ExternalRepositoryManager ::
               ACTION_BROWSE_EXTERNAL_REPOSITORY;
            $this->redirect(Translation :: get('DeleteSuccesfull'), false,
               $parameters);
        }
        else
        {
            $parameters = $this->get_parameters();
            $parameters[ExternalRepositoryManager ::
               PARAM_EXTERNAL_REPOSITORY_MANAGER_ACTION] =
               ExternalRepositoryManager ::
               ACTION_VIEW_EXTERNAL_REPOSITORY;
            $parameters[ExternalRepositoryManager ::
               PARAM_EXTERNAL_REPOSITORY_ID] = $id;
            $this->redirect(Translation :: get('DeleteFailed'), true,
               $parameters);
        }
    }
}
?>
```

Importer component

Importing is a default action, but actually importing an object can hardly be generalized. We can retrieve the object we're trying to import in our local repository but that's about it. Everything else, from the actual creation of an object in our own repository to the redirect is handled by the repository-specific implementation. The same principle applies to the synchronization components as well.

```php
<?php
class FlickrExternalRepositoryManagerImporterComponent extends
       FlickrExternalRepositoryManager
{

    function run()
    {
        $importer = ExternalRepositoryComponent ::
           factory(ExternalRepositoryComponent :: IMPORTER_COMPONENT, $this);
        $importer->run();
    }

    function import_external_repository_object($external_object)
```

```
        {
            if ($external_object->is_importable())
            {
                $image = ContentObject :: factory(Document :: get_type_name());
                $image->set_title($external_object->get_title());

                if (PlatformSetting :: get('description_required', 'repository') &&
                     StringUtilities :: is_null_or_empty($external_object-
                     >get_description()))
                {
                    $image->set_description('-');
                }
                else
                {
                    $image->set_description($external_object->get_description());
                }

                $image->set_owner_id($this->get_user_id());
                $image->set_filename($external_object->get_id() . '.jpg');

                $sizes = $external_object->get_available_sizes();
                $image->set_in_memory_file(file_get_contents($external_object-
                      >get_url(array_pop($sizes))));

                if ($image->create())
                {
                    ExternalRepositorySync :: quicksave($image, $external_object,
                                                $this->get_external_repository()-
                                                >get_id());

                    $parameters = $this->get_parameters();
                    $parameters[Application :: PARAM_ACTION] = RepositoryManager ::
                      ACTION_BROWSE_CONTENT_OBJECTS;
                    $this->redirect(Translation :: get('ImportSuccesfull'), false,
                      $parameters, array(ExternalRepositoryManager ::
                      PARAM_EXTERNAL_REPOSITORY, ExternalRepositoryManager ::
                      PARAM_EXTERNAL_REPOSITORY_MANAGER_ACTION));
                }
                else
                {
                    $parameters = $this->get_parameters();
                    $parameters[ExternalRepositoryManager ::
                      PARAM_EXTERNAL_REPOSITORY_MANAGER_ACTION] =
                      ExternalRepositoryManager :: ACTION_VIEW_EXTERNAL_REPOSITORY;
                    $parameters[ExternalRepositoryManager ::
                      PARAM_EXTERNAL_REPOSITORY_ID] = $external_object->get_id();
                    $this->redirect(Translation :: get('ImportFailed'), true,
                      $parameters);
                }
            }
            else
            {
                $parameters = $this->get_parameters();
                $parameters[ExternalRepositoryManager ::
                  PARAM_EXTERNAL_REPOSITORY_MANAGER_ACTION] = ExternalRepositoryManager
                  :: ACTION_VIEW_EXTERNAL_REPOSITORY;
                $parameters[ExternalRepositoryManager :: PARAM_EXTERNAL_REPOSITORY_ID]
                  = $external_object->get_id();
                $this->redirect(null, false, $parameters);
            }

        }
    }
?>
```

Editor component

... and then there are the annoying components. Things like an editor for instance, which could definitely be considered default functionality. Sadly enough, depending on the repository, the properties which can be manipulated can be very different and might require a completely different kind of editing form.

That's why this kind of component, which also includes the creation / uploader components, will always have to be written from scratch. That being said it's not illegal to let yourself be inspired by what's already there for Flickr, YouTube, MediaMosa and Google Docs. Just make sure you check the external repository's possibilities concerning uploading and editing data remotely. Most will definitely allow it, but the way it works might be very different.

Connecting the dots

Finished, right? Wrong. We have components and data structures, but do we already have our actual data? Whoops. Time to implement an actual external repository connector. As mentioned before there's no general solution for external repository connectors, just some guidelines which were derived from the way the generic components work.

Method	Returns	Description
count_external_repository_objects	Integer	Essential for pagination of tables. Returns the total number of objects that match the current search conditions
delete_external_repository_object	Mixed	Delete the given object from the external repository
export_external_repository_object	Mixed	Export the given local object to the external repository, especially takes care of the part of the export that needs to be handled by the API
retrieve_external_repository_object	ExternalRepositoryObject	Retrieve one particular external repository object based on it's ID
retrieve_external_repository_objects	ArrayResultSet	Retrieve a complete set of external objects as an ArrayResultSet of ExternalRepositoryObjects, matching the conditions and limited by an amount and offset.
translate_search_query	Mixed	Function to translate the search query (if any) to something the external repository can understand

Fig. 22 – Required methods for every external repository connector

Depending on the external repository you're trying to connect, additional methods are more than likely going to be necessary once you start adding non-default functionality. What follows are some extracts from the Flickr connector.

```php
<?php
require_once Path :: get_plugin_path() . 'phpflickr-3.0/phpFlickr.php';
require_once dirname(__FILE__) . '/flickr_external_repository_object.class.php';

class FlickrExternalRepositoryConnector extends ExternalRepositoryConnector
{
    /**
     * @param ExternalRepository $external_repository_instance
     */
    function FlickrExternalRepositoryConnector($external_repository_instance)
    {
        parent :: __construct($external_repository_instance);
        $this->key = ExternalRepositorySetting :: get('key', $this-
                    >get_external_repository_instance_id());
        $this->secret = ExternalRepositorySetting :: get('secret', $this-
                        >get_external_repository_instance_id());
        $this->flickr = new phpFlickr($this->key, $this->secret);

        $session_token = ExternalRepositoryUserSetting :: get('session_token',
                         $this->get_external_repository_instance_id());

        if (! $session_token)
        {
            $frob = Request :: get('frob');

            if (! $frob)
            {
                $this->flickr->auth("delete", Redirect :: current_url());
            }
            else
            {
                $token = $this->flickr->auth_getToken($frob);
                if ($token['token'])
                {
                    $setting = RepositoryDataManager :: get_instance()-
                               >retrieve_external_repository_setting_from_variable_
                               name('session_token', $this-
                               >get_external_repository_instance_id());
                    $user_setting = new ExternalRepositoryUserSetting();
                    $user_setting->set_setting_id($setting->get_id());
                    $user_setting->set_user_id(Session :: get_user_id());
                    $user_setting->set_value($token['token']);
                    $user_setting->create();
                }
            }
        }
        else
        {
            $this->flickr->setToken($session_token);
        }
    }

    /**
     * @param mixed $condition
     * @return int
     */
    function count_external_repository_objects($condition)
    {
        $photos = $this->retrieve_photos($condition, $order_property, 1, 1);
        return $photos['total'];
    }

    function retrieve_external_repository_object($id)
    {
        $licenses = $this->retrieve_licenses();
        $photo = $this->flickr->photos_getInfo($id);

        $object = new FlickrExternalRepositoryObject();
        $object->set_external_repository_id($this-
                 >get_external_repository_instance_id());
        $object->set_id($photo['id']);
        $object->set_title($photo['title']);
        $object->set_description($photo['description']);
        $object->set_created($photo['dateuploaded']);
        $object->set_modified($photo['dates']['lastupdate']);
        $object->set_owner_id($photo['owner']['username']);

        $tags = array();
        foreach ($photo['tags']['tag'] as $tag)
        {
            $tags[] = array('display' => $tag['raw'], 'text' => $tag['_content']);
        }
        $object->set_tags($tags);

        $photo_sizes = $this->flickr->photos_getSizes($photo['id']);
        $photo_urls = array();

        foreach ($photo_sizes as $photo_size)
        {
            $key = strtolower($photo_size['label']);
            $photo_urls[$key] = $photo_size;
        }

        $object->set_urls($photo_urls);
        $object->set_license($licenses[$photo['license']]);
        $object->set_type('flickr');

        return $object;
    }

    /**
     * @param ContentObject $content_object
     * @return mixed
     */
    function export_external_repository_object($content_object)
    {
        return $this->flickr->sync_upload($content_object->get_full_path(),
                 $content_object->get_title(), $content_object->get_description());
    }

    /**
     * @param string $id
     * @return mixed
     */
    function delete_external_repository_object($id)
    {
        return $this->flickr->photos_delete($id);
    }
}
?>
```

It's worth noting that the actual connection with Flickr, via phpFlickr, is already established in the constructor of the connector. It also contains the logic needed to make sure the application and/or user can be authenticated.

To infinity and beyond

At the moment this is being written, implementations have already been realized for Flickr, YouTube, Google Docs and MediaMosa, which already covers a big collection of content and content types. The Fedora connector still has to be converted to the new framework and support for Opencast Matterhorn will be added in the very near future.

As this particular functionality evolves and more and more connectors are being added we hope this may prove to be one of the big assets of Chamilo 2.0. So if we allow users to collaborate within the confines of our platform, surely we want them to collaborate beyond those borders as well? Implementing a framework for external repositories is a definite first step in that direction. And to answer that question I've asked throughout this document. Yes, we really think that's kind of great.

Get involved

Do you like the possibilities that external repositories create for Chamilo 2.0? Would you like to get involved and implement an extension of your own? Are you on the development team of a repository project and are you interested in cooperating?

Don't hesitate to contact us via e-mail or let us know about your project via the community website located at http://www.chamilo.org. We're looking forward to hearing from you.

Index