# MyLibrary in Croatia: A Workshop

MyLibrary is a digital library toolbox -- a set of object-oriented Perl modules designed to do I/O against a data model rooted in the Dublin Core meta-data elements and supplemented with a faceted classification system. These modules, in combination with other pieces of software, enable librarians and developers to create digital library collections and services.

The goals of this workshop are to describe the functionality of MyLibrary, demonstrate a number of ways it can be used, and make participants more aware additional venues for creating and maintaining digital libraries. At the end of the workshop participants will be able to: describe what MyLibrary can and can not do, design a faceted classification system, understand how to use the MyLibrary API to create digital library collections and implement digital library services, outline a process of harvesting OAI content into a MyLibrary instance, as well as outline methods to syndicate MyLibrary content.

*Eric Lease Morgan*
University Libraries of Notre Dame

Feburary 22, 2006

# Workshop outline

A. MyLibrary is a toolbox, not an application
B. Digital libraries are a set of collections combined with services
C. A workflow for implementing a digital library
    1. Answer questions regarding information architecture: users, context, and content
    2. Allocate resources: time, money, hardware & software and people
    3. Implement
    4. Conduct usability studies
    5. Evaluate
    6. Repeat
D. Facets & terms: Designing a controlled vocabulary
E. Doing data-entry
    1. Sets of books
    2. Sets of journals
    3. Sets of Internet resources
F. Making content searchable
    1. Index the content with swish-e
    2. Make the index accessble via SRU
    3. Create a simple SRU client
G. Creating user interfaces
    1. Browsable interfaces
    2. Searchable interfaces
    3. Personalized/customized interfaces
H. Automate data-entry
    1. Import sets of MARC records
    2. Import a set of images from an OAI repository
    3. Import a set of journal titles from an OAI repository
I. Syndicating MyLibrary content
    1. Write a pathfinder
    2. Create a search bookmarklette
    3. Make an OAI data repository
    4. Create RSS feeds
J. Discuss the state of librarianship

# MyLibrary implemenetation and maintenance

This essay outlines one way to implement and maintain a MyLibrary instance. The following process is suggested:

1. Assemble a team of people to do the work.
2. Give the team the necessary resources to accomplish the job.
3. Answer questions regarding information architecture.
4. Install and configure MyLibrary.
5. Fill MyLibrary with content -- do input.
6. Create interfaces to provide access to the content -- facilitate output.
7. Do usability testing against the interfaces.
8. Maintain the content.
9. Evaluate and go to Step #1.

Each of these steps is described in more detail in the following sections.

## Assemble a team of people to do the work

You will need to assemble a team of people to do the work, unless of course Leonardo Di Vinci works in your library. Few people posses all of the necessary skills. At the very least your team will probably consist of a:

- systems administrator
- Perl programmer
- graphic designer
- subject specialist

The systems administrator is responsible for maintaining your computer's hardware, software, and networking infrastructure. They need to be knowledgeable about operating systems, filesystems, users/groups, and Internet connections. They are the people who install and configure things like Apache, Perl, and MySQL. Some places have computer centers who routinely do these sorts of activities. Working with the programmer, the systems administrator will install the MyLibrary Perl modules. Once MyLibrary is installed the systems administrator will be primarily responsible for making sure the computer is running smoothly. Make sure they back up your data on a regular basis.

The Perl programmer is responsible for creating functional interfaces to the underlying MyLibrary database. Some of these interfaces are computer-to-computer interfaces such as the importing of MARC records from a catalog or the exporting of Real Simple Syndication (RSS) feeds. Other interfaces will have human components, and in such cases the programmer will need to work closely with the graphic designer. The programmer is not expected to create everything from scratch since the MyLibrary distribution comes with a number of sample interfaces. You might want to simply use one of these interfaces instead of creating your own. It is essential for the programmer to be familiar with object-oriented programming techniques and common gateway interface (CGI) scripting.

The graphic designer is responsible for making sure your human-to-computer

interfaces are usable and aesthetically pleasing. (Usability is different from functionality.) They need to have an in-depth knowledge of HTML, XML, cascading stylesheets, and the principles of user-centered design. Ideally the user interfaces written by the Perl programmer will output rudimentary HTML with plenty of HTML class and id attributes to be used as hooks for the cascading stylesheets. Through the stylesheets the graphic designer should be able to modify the look & feel of the interface. This is called separating presentation from content. The graphic designer should also be an advocate for usability testing, and they should have a thick skin enabling them to take criticism well.

Finally, the team will require someone who is knowledgeable about content, a subject specialist. This person will bring to the team the principles collection development, cataloging & classification, as well as reference services -- all of the traditional activities of librarianship. This person will be the primary driver behind the process of answering questions regarding information architecture, outlined below. Once the questions are answered, the subject specialist will be responsible for putting the answers into practice through data-entry. The subject specialist will need to articulate sets of facets and terms, select information resources, and enter everything into the system accordingly. The subject specialist should also be keenly aware of user-centered design principles because the nature of librarianship has dramatically changed with the advent of the Internet. Expectations regarding the access and use of information now are quite different from the expectations of ten years ago.

None of the people and skills outlined above are more important than the other. Each are equally necessary for a successful implementation. At the same time you might consider supplementing your team with people with more specialized skills such as:

- relational database design and implementation
- indexing techniques
- advanced XML applications and XSLT programming
- conducting surveys and doing statistical analysis
- facilitating focus group interviews and usability studies
- creating and maintaining controlled vocabularies
- doing large volumes of data-entry and maintenance

Each of the activities and skill sets are described in greater detail throughout the book. You are encouraged to consult those chapters for more detail.

## Give the team the necessary resources to accomplish the job

Computer hardware/software and time are the necessary resources for the team to complete their implementation.

The hardware/software requirements for implementing MyLibrary are minimal. Really. About any Intel-based computer with at least 512 MB of RAM and 2 GB of disk space will do just fine in terms of hardware. The more RAM the better. Now-a-days it is uncommon to have a computer with less then 20 GB of disk

space. If you were to purchase a new computer to host just MyLibrary, then $2,000 will buy you a great piece of hardware.

MyLibrary is essentially an open source, LAMP (Linux, Apache, MySQL, Perl) system. Therefore, if you or your institution already have a Linux computer up and running, then it will probably work quite well. MyLibrary is designed to do input/output against a relational database. The MyLibrary installation process is designed for use the relational database program called MySQL. Since MyLibrary is a set of Perl modules, you will need Perl installed on your computer. Any version of Perl 5.0 or later will work. We use version 5.8.5. The MyLibrary modules require a number of other Perl modules. The easiest way to install these modules is through the use of CPAN. Invariably you will want to serve your MyLibrary content over the Web through an HTTP server. We use Apache. Any HTTP server will do as long as you are able to run CGI scripts from within its filesystem. Your systems administrator and Perl programmer are expected to understand these details.

In short, if you wanted to start from scratch you could probably use one of the desktop hand-me-downs lying around the office. Install on it Linux, Apache, MySQL, Perl, and you are ready to go. If you already have a computer in place, and it already has Apache, MySQL, and Perl installed, then that computer will work just fine too.

Time is by far the more expensive resource necessary to fully implement MyLibrary. Time will need to be allocated in a number of ways. First of all, time will need to be spent allowing the team members to actually become a team. Many people think this process is too "touchy-feely". On the other hand, the sooner the team establishes norms of behavior, decides how to build consensus, and learns how to work with each other the more quickly your implementation will come to fruition. This is especially true if the team members do not regularly work together. Allow the members to go on a field-trip or two as well as one more more retreats. Feeding them helps too.

Second, time will need to be spent answering the questions of information architecture. On the surface this too appears to be a lot of "navel gazing" but time spent addressing these issues will uncover hidden assumptions, help you set priorities, outline the problems MyLibrary will be expected to address, and build relationships with your patrons. While this work does not produce a whole lot of tangible results, the result forms the entire foundation of your implementation.

Finally, time will need to be spent doing the work normally associated with the implementation of computer technology. Setting up hardware and software. Writing and/or configuring computer programs. Customizing interfaces to meet your specific needs. Filling the system with data. Maintaining the data. Evaluating success. Repeating the entire process. Here again, remember that any computer implementation consists of 20% computer work and 80% people work.

Using the interfaces supplied with the MyLibrary distribution, a competent Perl computer programmer should be able to

install, configure, and make accessible a couple dozen Internet resources through a searchable/browsable interface in about two days. Such an implementation circumvents the prospects for a robust hardware/software infrastructure, an integrated user interface with the balance of your site, let alone the principles of information architecture.

If you wanted MyLibrary to be the primary driver of your library's website, then the entire implementation process might take as long as a year. The time you spend will not necessarily be computer-related but related to the why's and wherefore's of the system as well as ongoing maintenance.

## Answer questions regarding information architecture

The first thing for the implementation team to do is to answer questions regarding information architecture. There are essentially three questions to be addressed:

1. Who is your audience and what are their needs/desires?
2. What is the purpose of your implementation and how does it fit within the context of your institution?
3. What type of content will your implementation contain, and how will it be conceptually organized?

These questions were elaborated upon in a previous chapter.

As answers regarding information architecture are articulated, write them down and share them with the stakeholders throughout your institution -- both inside and outside the library.

Answering these questions is a never-ending process. Regularly revisit the answers regarding your information architecture.

## Install, configure, and fill MyLibrary

Technically speaking, MyLibrary is a set of object-oriented Perl modules providing the means for doing input and output against a specifically shaped relational database. Therefore you will need a computer with Perl installed with hooks to a relational database. MyLibrary is presently configured to use MySQL as the database, but without too much tweaking it should be able to do input/output against other relational databases such as Oracle or Postgres. Similarly, MyLibrary was developed on top of a Unix operating system, but people have installed it on the Windows platform.

Assuming you already have MySQL installed, below is an outline of the necessary steps used to install the Perl modules. Much of this process is done for you by running the perl Makefile.PL command from within an uncompressed MyLibrary distribution:

1. Create a MyLibrary database using the sample-data.sql or STRUCTURE-ONLY.sql files found in the distribution's db directory.
2. Create and configure a MySQL user with permissions to read and write to the newly created database.
3. Edit MyLibrary's Config.pm module to record the network location of the database as well as the username/password of the authorized user.

4. Install the Perl modules.

Once this is done you should be able to write CGI or command-line driven scripts allowing you to do various types of input and output against the database. Many sample scripts are located in the distribution's bin, cgi-bin, and cgi-admin directories.

Almost invariably you will want to use a Web-based interface to do at least some of your data-entry. The cgi-admin directory contains a family of CGI scripts allowing you to do this. Like all the other scripts in the distribution, the scripts are only samples. Save them in a directory on your Web server where CGI script execution is permitted and begin data-entry. To do so try this:

1. Articulate a set of facet and terms used to provide the conceptual organization of your content. This is described in more detail elsewhere in this document.
2. Use the administrative interface to enter the facets and terms.
3. Optionally, add descriptions of one or more librarians and be sure to associate them with one or more facet/term combinations.
4. Create at least one location type. Information resources take many forms as do their location types. For right now, create a location type called something like "Internet resource". These location types will take the form of URL's.
5. Finally, use the administrative interface to add to your collection. At the very least you will want to give each resource a title, a description, and a location (URL). You will also need to associate each resource with at least one facet/term combination.

Since MyLibrary is really a set of Perl modules and not an application, data-entry can be done from the command-line as well as in batch mode. For example, here at Notre Dame we regularly dump sets of MARC records (supplemented with facet/term combinations) from our catalog, convert these files into RDF/XML files, and import them into our MyLibrary database. Alternatively, since the fields in the underlying MyLibrary database are a superset of the basic Dublin Core elements, it is possible to harvest content from OAI-PMH repositories and cache it to MyLibrary. This provides another way to fill a MyLibrary instance.

## Create interfaces, and do usability testing

The MyLibrary distribution includes a few sample interfaces to your MyLibrary implementation. These interfaces will not, nor are they expected to, satisfy the needs of every institution. Instead, they are examples of how the underlying system can be exploited to meet you and your patron's needs. Programmers are expected to read the Perl API, examine the code from the sample applications, work with the balance of the MyLibrary team, and write programs fitting your particular needs.

Usability testing is a highly structured communications process. It is not science. The word "test" is a misnomer. A better word might be "study". Other sections of this book described usability studies in much greater detail.

Finally, and just as importantly, make sincere efforts to practice user-centered design when creating your interfaces and doing your usability testing. The Internet has significantly changed user's expectations regarding the access and use of information. The older roles of libraries learned in library schools are increasingly outmoded. Us librarians need to rethink much of what it means to be a librarian in an era of globally networked information. Put less emphasis on personal experience and antidotal evidence. Instead, use focus groups, surveys, log file analysis, and usability "studies" to form the basis of your decision-making.

## Maintain content

Once you have a production implementation of MyLibrary in place the largest ongoing activity will be maintaining the content. How you do this depends a great deal on the types of content in your implementation, where it originates, and where it is used.

For example, if your content primarily comes from your catalog and gets imported into MyLibrary via sets of MARC records, then maintaining your content will be a matter of maintaining your catalog. You already have processes in place for this type of work.

If your content comes from OAI data repositories, then maintence will most likely take the form of regularly run programs against those repositories.

More than likely, your content will be a mixture of things from your catalog and sets of Internet resources usually not deemed worthy of putting in your catalog. (For example, items you do not own nor have licensed.) In these cases you will probably use a combination of automated and manual data-entry methods. The records in your system that were entered manually will need to be regularly examined. Do the links still work? Are they still relevant according to your overall information architecture? Do they still fit within your collection development policy? If not, then you will need to update or weed them from your collection.

The organization of MyLibrary content is postulated on sets of locally-designed facets and terms -- a controlled vocabulary. By definition, a controlled vocabulary is a form of human language. Language is ambiguous and ever-changing. It will be necessary to monitor your facets and terms updataing them as time goes on. Do you need to create new subject facets? Have new audiences become a part of your community and will it then be necessary to create an audience facet? Do you now have access to new types of information like sounds or data sets? If so, then you may need to update your facets and terms. Does your hosting institution (college, university, company, or municipality) host a portal? Do you want to advertize not only your information resources but also your services in the portal? If so, then you may need to go beyond the traditional facets such as subjects, formats, and research tools, and enhance the them with things like help and bibliographic instruction.

Processes for maintaining your content will differ greatly from library to library. Consider reallocating existing personnel for the task. In principle the maintenance process is similar to the maintenance process of other content in your library.

The difference is only the environment in which it takes place.

## Evaluate and repeat

Library work is never done. Students come and students go. Younger people get older and require/desire different aspects of library service. Collections are rarely complete. Technology is constantly changing, and these changes modify user expectations. Priorities are modified over time. Budgets fluxuate.

For all these reasons it is a bad idea to think of your MyLibrary implementation as a static thing. It will need constant monitoring. Is it getting used in the manner you expected? It it meeting expressed user needs and desires? Does it cost more than the percieved benefits? On a regular basis you will want to ask yourself these sorts of questions, and depending on the answers you will want to return to Step #1.

# First Principles of Information Architecture: “On your Mark. Get set. Go!” not “Fire, and then Aim.”

At its core, information architecture is about users, context, and content. By answering questions regarding these issues your MyLibrary implementation will not only be functional. It will be understandable to your intended audience, serve a meaningful purpose, and contain relevant content. Information architecture is the result of a planning process. It is about "On your mark. Get set. Go!" not "Fire, and then aim." This essay elaborates on these ideas and outlines some of things you need to think about as you begin to implement any information system, not just MyLibrary.

## A definition

Information architecture is often illustrated using a Venn diagram depicting three interlocking circles representing users, context, and content. Users are the intended audience of an information system, context is the reason the systems exists, and content is the data/information the system has to communicate. For good information architecture to take place, a concrete understanding of an information system's audience, purpose, and data/information is necessary. This is like the architecture of buildings, where an understanding of who is going to live there, what the building is for, and what it will contain must be outlined before construction can begin.

At the risk of pushing the metaphor too far, the result of information architecture is a "blueprint illustrating the framework" which you will fill with content, organize with controlled vocabularies, hang site-wide navigation, and make browsable as well as searchable. If you do this with an eye to satisfying the expressed needs and desires of your users as well as your hosting institution, you will end up building something usable (not just functional), and they will come.

## Users

The first step in designing your information architecture is answering questions regarding users. You need to define the primary audience of your information system, build relationships with them, and learn what they need and desire.

Defining your information system's primary audience is easier than you may think. In a private university like Notre Dame, the primary audience includes the University's students, faculty, and staff. The needs of these people take precedence over the needs of the general public, alumni, or scholars from other institutions. There are limited resources (time and money) allotted to the implementation of your information system, and it is not possible to be all things to all people. Consequently, you need to prioritize and decide to whom, primarily, you are going to cater your service. At a public university, the audience may be broader, including the general public, especially the public of the immediate area or region. In a public library, the primary audience may be area residents. In special and school libraries, the answers to these questions will seem almost obvious.

After defining who your audience is, you need to establish inter-personal relationships with them. No, you don't have to become their best buddy, but you do need to build rapport to learn their expressed needs and desires. You need to learn and, more importantly, understand the challenges and difficulties they are having when it comes to

doing their work. I'm sure you can create a long list of their challenges and difficulties, but since you are not them you can not prioritize which of the challenges and difficulties are the ones they need addressed. By building relationships with your primary audience you will learn these priorities and be able to focus your resources on making them easier to accomplish.

There are many ways to build relationships and learn of your audience's priorities. Surveys are the first thing that come to mind. They are relatively inexpensive. They can touch large numbers of people, and they are good for answering "what" types of questions. "What is your age?" "What do you like and what do you dislike about our present information system?" "If you could change one thing, what would it be?" The answers to survey questions often need to be short and succinct; few people are going to give you a lot of detail while answering survey questions. The results of surveys usually manifest themselves numerically and then get converted into graphs. Along the lines of surveys are log file analysis. By looking at the statistics captured by your staff as well as your present information systems, you will get an idea of what your audience uses. People will often say one thing and act differently. Log files help put this behavior into perspective.

The other side of surveys are focus group interviews, structured communication sessions used to learn about your audience's feelings. When compared to surveys, focus group interviews require a greater degree of interpersonal skills on the part of a facilitator. They touch fewer people than surveys and therefore are often times seen as more expensive. On the other hand, focus group interviews answer questions surveys don't answer, specifically "why" questions. "Why do you like this service as opposed to another?" "Why do you think it is important to for us to implement such and such feature?" "Why do you spend your time working in this particular manner?" Just like surveys, the focus group interview process ranges from the simplistic to the complex. It can be as simple as a one-on-one chat over coffee, or it can be as elaborate as a meeting of six to twelve homogeneous people who answer questions in a moderated setting by a professional facilitator.

In conclusion, in order to learn about your audience's needs and desires, consider issuing one or more surveys first and following up with sets of focus group interviews second. This process will enable you to validate the survey's conclusions and learn why people answered the survey the way they did.

## Context

The next step in articulating an information architecture is to answer questions regarding context. What is the purpose of your information system, how does it fit within the totality of your institution's products and services, and what sorts of resources (time and money) are allotted to the system's development and maintenance?
Your information system will not exist in a vacuum. It will be a reflection of its hosting institution, and in order for the information system to reflect well you will need to know the goals and priorities of your institution. For example, you need to know the purpose of the hosting institution. What problems is it trying to solve? How can your information system be expected to contribute to the solutions? Look to your institution's mission statement for answers. Here at Notre Dame the library's role is to help the students, faculty and staff of the University community do their learning, teaching, and scholarship. The role of our website (and our MyLibrary implementation) mirrors the purpose of the University Libraries: to help facilitate learning and teaching, to assist in scholarship, to supplement access to collections and service, and to facilitate communication.

The context of your information system will also be tempered by the amount of resources allotted to its development and maintenance. These resources take the form of time, money, hardware, software, people, and expertise. The implementation and ongoing maintenance of your information system will require a diverse set of skills. None of which are necessarily more important than the other. The people with the necessary skills include subject experts, leaders of people, graphic designers, people who can mark up texts, knowledge workers who can organize content, usability experts, marketers, programmers, and systems administrators. The amount of time and energy these sorts of people can bring to the implementation of your information system is directly proportional to what your information system will enable people to do and do well. When the Web began a little more than fifteen years ago people's expectations were low, but with the growing size and diversity of the Web, people's expectations have matured, and consequently so does the need to allocate more resources to your implementation.

Defining the purpose of your information system and articulating what resources will be spent on its development is the second step in the creation of your information architecture. Do not set your goals too high lest you set yourself up for failure. Determine the importance of your information system compared to the other products and services you offer, and allocate your resources accordingly.

## Content

The third step in the creation of your information architecture is defining what content it will contain. This is akin to articulating a collection development policy.

Not even Google provides access to the totality of the world's content, and there is no reason to expect you to fill this role. Instead, focus on the answers regarding users and content to define the scope of your content. Ask yourself, "What are the strengths of my institution?" "To what degree does my collection need to be comprehensive, authoritative, up-to-date, written in a particular language, presented in an aesthetically pleasing manner, etc.?" In other words, create a list of guidelines that your information resources need to embody in order to be a part of your collection. Just because a particular information resource is about a particular subject does not necessarily mean it is a good candidate.

When the University Libraries of Notre Dame re-created its website using MyLibrary, we decided the content would not be very much different from the content of traditional, physical libraries. It contains tools to access bibliographic information, access to digital library services and collections, instructions for pedagogy, and last but not least, access to people who can help with all these processes -- librarians. The website is not designed to be a comprehensive list of resources. Instead, it is designed to highlight the most significant resources and provide starting points for learning and research. The content of the website is very much like the content of traditional library pathfinders.

## Summary

Information architecture is about answering difficult questions regarding users, context, and content. It is not possible to be everything to everybody, therefore you need to define who your primary audience is. Users. Your information system is a part of a larger institution, therefore it behooves you to make sure the system fits into the institution's goals and objectives. Context. The world of information is too large for any system to embody, and therefore you need to limit the scope of your collection. Content. Once you answer the questions regarding users, context, and content, write down the

answers. Use them as guidelines for a specific period of time (at least one year), and then regularly revisit the guidelines. On your mark. Get set. Go. Not, fire and then aim.

# MyLibrary API Tutorial

This MyLibrary Applications Programmer Interface (API) tutorial gives the reader an overview of how to use the MyLibrary modules. It is only an introduction. The reader is expected to understand the principles of basic object-oriented Perl programming.

By the end of the tutorial the reader should be able to: create sets of facets, create sets of terms, create sets of librarians, create sets of location types, create sets of resources, classify librarians and resources with terms, work with sets of resources assoicated with particular sets of terms, output the resources' titles, descriptions and locations, create a freetext index of MyLibrary content, harvest OAI repositories and cache the content in a MyLibrary database.

## Initialization

To include MyLibrary into your scripts you "use" it:

```
# include the whole of MyLibrary
use MyLibrary::Core;
```

This will enable all the necessary modules. You can use selected modules if you so desire. This will save you a bit of RAM and compile time, but not a whole lot. For example:

```
# include just selected modules
use MyLibrary::Facet;
use MyLibrary::Term;
```

Make your life easy. Just include the whole of MyLibrary. See MyLibrary::Core's pod for more information.

## Configuration

Each installation of the MyLibrary modules is configured, by default, to work against at least one MyLibrary instance. This instance was created during the make process. When you include MyLibrary::Core, the default instance will be read from and written to.

If you want to read and write to a different instance of MyLibrary, then you will need to use the MyLibrary::Config methods to specify the database options for that instance.

## Facets

One of the first things you will want to do with any MyLibrary instance is create a set of facets.

Facets are a set of broad classification headings. Most instances of MyLibrary will contain some sort of Subjects facet to denote the "aboutness" of items. Other possible facets include Formats or Audiences. Formats could denote the physical manifestations of information resources. Audiences might denote who are the intended users of information resources.

Here are a number of ways to create and manipulate facet objects:

```
# create a facet object
$facet = MyLibrary::Facet->new;

# set the facet's name and note
$facet->facet_name('Subjects');
$facet->facet_note('The "aboutness" of items');

# save the facet to the database
$code = $facet->commit;
if ($code ne 1) { die 'commit failed' }

# get the facet's id; think "database key"
$id = $facet->facet_id;

# get the facet's name and note
$name = $facet->facet_name;
$note = $facet->facet_note;

# get a specific facet based on an id
$facet = MyLibrary::Facet->new(id => 1);
```

Given the methods outlined above, you could use the following code to create, save, retrieve, and then display a facet:

```
# configure
$name = 'Formats';
$note = 'The physical manifestation of resources';

# create, save, retrieve, and display
$facet = MyLibrary::Facet->new;
$facet->facet_name($name);
$facet->facet_note($note);

# save
$facet->commit;
$id = $facet->facet_id;

# retrieve
$facet = MyLibrary::Facet->new(id => $id);

# display
print '  ID: ' . $facet->facet_id   . "\n";
print 'Name: ' . $facet->facet_name . "\n";
print 'Note: ' . $facet->facet_note . "\n";
```

You will often want to get a list of all the facets in your system in order to facilitate browsable interfaces to your collection of resources. The get_facets method is used for this purpose. Since get_facets returns an array of objects, you can now loop through the array and process each item. This is how you might display them:

```
# create a list of all the facets in the system
@facets = MyLibrary::Facet->get_facets;

print "ID\tName\t(Note)\n";
foreach $f (@facets) {

  print $f->facet_id   . "\t" .
        $f->facet_name . "\t(" .
        $f->facet_note . ")\n"

}
```

Read the scripts named manage-facets.pl and subroutines.pl to see an example of how to manage sets of facets from a terminal-based interface. For more information read the pod of MyLibrary::Facets.

## Terms

Terms are a set of narrower classification headings, and each term is associated with one and only one facet -- its parent. Terms are expected to be the controlled vocabulary of your MyLibrary implementation, and consequently they are expected to be assigned to one or more information resources. Terms might include Astronomy, Music, or Mathematics, and these terms may have a parent facet named Subjects. Other terms might be Book, Journal, or Image, and these terms might be associated with a facet called Formats. Still other examples include Catalog, Dictionary, or Encyclopedia, and could associated with a facet named Research Tools.

The methods of MyLibrary term objects are very similar to the methods of facet objects:

```
# create a term object
$term = MyLibrary::Term->new;

# set the term's name and note
$term->term_name('Dictionary');
$term->term_note('A list of word definitions');

# create a facet named Research Tools
$facet = MyLibrary::Facet->new;
$facet->facet_name('Research Tools');
$facet->facet_note('Traditional library objects like dictionaries.');
$facet->commit;

# associate (join) this term to that facet
$term->facet_id($facet->facet_id);

# save the term to the database
$code = $term->commit;
```

```
if ($code ne 1) { die 'commit failed' }

# get the term's id; think "database key"
$id = $term->term_id;

# get the term's name and note
$name = $term->term_name;
$note = $term->term_note;

# get a specific term based on its id
$term = MyLibrary::Term->new(id => 1);
```

Given the methods outlined above, you could use the following code to create, save, retrieve, and then display relevant term data:

```
# configure
$term_name  = 'Sophomores';
$term_note  = 'Students in the second year of college';
$facet_name = 'Audiences';
$facet_note = 'People who use your services';

# create a term
$term = MyLibrary::Term->new;
$term->term_name($term_name);
$term->term_note($term_note);

# create a facet
$facet = MyLibrary::Facet->new;
$facet->facet_name($facet_name);
$facet->facet_note($facet_note);
$facet->commit;

# join and save
$term->facet_id($facet->facet_id);
$term->commit;

# get and display
$id = $term->term_id;
$term = MyLibrary::Term->new(id => $id);
$facet = MyLibrary::Facet->new(id => $term->facet_id);
print '    ID: ' . $term->term_id     . "\n";
print '  Name: ' . $term->term_name   . "\n";
print '  Note: ' . $term->term_note   . "\n";
print 'Parent: ' . $facet->facet_name . "\n";
```

Like the facets, you will often want to get a list of all the terms in your system in order to facilitate some sort of browse function. The get_terms method is used for this purpose:

```
# get all the terms
@terms = MyLibrary::Term->get_terms;
foreach $term (@terms) { print 'Term: ' . $term->term_name . "\n" }
```

Creating a list of sorted terms involves creating a list of term ids and calling the sort method denoting the sorting field, usually name:

```
# get all terms
@terms = MyLibrary::Term->get_terms;

# print
foreach $t (@terms) { print 'Term: ' . $t->term_name . "\n" }

# create a list of term ids
foreach $t (@terms) { push @term_ids, $t->term_id }

# get a sorted list of term id
@terms = MyLibrary::Term->sort(term_ids => [@term_ids],
                               type => 'name');

# print, again
foreach $t (@terms) {

  $term = MyLibrary::Term->new(id => $t);
  print 'Term: ' . $term->term_name . "\n";

}
```

After terms have been assigned to MyLibrary resource objects a number of other useful term methods present themselves, but they are outlined in a later section named "Terms and resources revisited".

Read the scripts named manage-terms.pl and subroutines.pl to see how you can manage sets of terms from a terminal-based interface. For more detail read the MyLibrary::Term pod.

## Librarians

Question: What do libraries have that Yahoo and Google don't have? Answer: Librarians -- people who are willing and able to address the information needs of others. That is why librarian objects are a part of MyLibrary.

Think of librarian objects as information resources with the characteristics of people: name, address, telephone number, and URL of home page. In libraries librarians usually have subject specialties, and that is why it is possible to "catalog" librarians with facet/term combinations.

The setting and getting of MyLibrary librarian objects works like this:

```
# create a librarian
$librarian = MyLibrary::Librarian->new;

# give the librarian characteristics
$librarian->name('Fred Kilgour');
$librarian->email('kilgour@oclc.org');
$librarian->telephone('1 (800) 555-1212');
$librarian->url('http://oclc.org/~kilgour/');

# create an astronomy term as a child of the subjecs facet
```

```
  $term = MyLibrary::Term->new;
  $term->term_name('Astronomy');
  $term->term_note('Studying the stars');
  $facet = MyLibrary::Facet->new(name => 'Subjects');
  $term->facet_id($facet->facet_id);
  $term->commit;

  # associate (join) the librarian with astronomy
  $librarian->term_ids(new => [$term->term_id]);

  # save the librarian
  $librarian->commit;

  # get the librarian
  $id = $librarian->id;
  $librarian = MyLibrary::Librarian->new(id => $id);

  # display basic information about the librarian
  print '       ID: ', $librarian->id, "\n";
  print '     Name: ', $librarian->name, "\n";
  print '    Email: ', $librarian->email, "\n";
  print 'Telephone: ', $librarian->telephone, "\n";
  print 'Home page: ', $librarian->url, "\n";

  # display each of their associated subject areas
  @term_ids = $librarian->term_ids;
  foreach $id (@term_ids) {

    $term = MyLibrary::Term->new(id => $id);
    print '    Term: ', $term->term_name, "\n";

  }
```

Just like everything else, you might want to pull all of the librarians out of the system. The class method get_librarians is used for this purpose. It returns an array of librarian objects:

```
  # get all librarians
  @librarians = MyLibrary::Librarian->get_librarians;

  # print each librarian's name and email address
  foreach $l (@librarians) { print $l->name . ' <' . $l->email . ">\n"
}
```

Question: Who are you going to call? Answer: The Librarian. By creating a set of facet/term combinations and associating them with information resources you can effectively group like things together. By associating the same facet/term combinations to librarians you can begin to make connections between information resources and librarians. Thus, when displaying lists of information resources, consider adding the associated librarian's name and contact information to the list.

For more detail regarding librarian objects read the MyLibrary::Librarian pod.

## Location types

The world of information resources is made up of many different types. For example there are books, journals, and websites. To complicate matters, things like the same books or journals can be manifested in physical or digital form. Heck, the book or journal could even exist in a number of physical forms such as a codex or microfiche or maybe even a film. Because of these things a single information resource may have many different locations and each of these locations may be of different types: call numbers, URL's, buildings, etc. Because all information resources have some sort of location will need to create at least one location type in your MyLibrary implementation.

Location types are just labels for different types of locations. For example, almost all MyLibrary implementations will have a location type such as Internet Resource, or URL. If the information resources in your MyLibrary implementation includes books -- physical, every-day books -- then another location type for your implementation might be Call Numbers. Suppose you have an electronic journal and one URL associated with the journal is a pointer to the content and another URL points to a help file. In this case you might want to have an additional location type such as Help Location.

Here are an example of how you might implement a simple Internet resource location type:

```
# create a location type
$location_type = MyLibrary::Resource::Location::Type->new;

# give it characteristics
$location_type->name('URL');
$location_type->description('A type of Internet resource');

# save it and get its id
$location_type->commit;
$id = $location_type->location_type_id;

# get a location by an id and display its data
$location_type = MyLibrary::Resource::Location::Type->new(id => $id);
print '   ID: ' . $location_type->location_type_id . "\n";
print ' Type: ' . $location_type->name . "\n";
print ' Note: ' . $location_type->description . "\n";
```

Like most of the other modules, MyLibrary::Resource::Location::Type provide a class method for getting everything. In this case it is all_types, and it returns an array of location type ids:

```
# get all location types
@location_types = MyLibrary::Resource::Location::Type->all_types;

# display them
foreach $l (@location_types) {

  $location = MyLibrary::Resource::Location::Type->new(id => $l);
  print 'Type: ' . $location->name . "\n";
```

```
}
```

You can also create a location type object by calling it by name, but the name must exist in the underlying database. To do this you supply the name parameter to the new method:

```
# get the location type object named URL
$location_type=MyLibrary::Resource::Location::Type>new(name=>'URL');

# print
print '  Location type ID: ' . $location_type->location_type_id."\n";
print 'Location type name: ' . $location_type->name          ."\n";
print 'Location type note: ' . $location_type->description   ."\n";
```

Because information resources are manifested in many ways, and since each of these ways are usually associated with different types of "addresses" (such as URLs or call numbes) MyLibrary provides as means of creating and listing such types.

See the terminal-based program called location-types.pl as well as the pod for MyLibrary::Resource::Location::Type for more detail.

## Resources

Now the fun really begins.

With the exception of the librarians, all of the previous sections essentially described how to create sets of controlled vocabularies. Facets. Terms. Location types. You are now ready to create lists of information resources, describe them, classify them, and save them to the underlying database. Once you have built your collection you are expected to write reports against it implementing various services such as: browse, search, What's New?, Find More Like This One, most popular, most useful, export subsets to a file, send subsets as email, create RSS feeds, etc. In today's world of changing user expectations it is not only about collections. It is more about the effective combination of collections and services.

MyLibrary resource objects include methods for setting and getting the objects' characteristics, and these characteristics are a superset of the basic fifteen Dublin Core elements. There is an implicit one-to-one relationship between the basic Dublin Core element names and many of the MyLibrary resource object methods/objects, listed below:

1. contributor - MyLibrary::Resource->contributor
2. coverage - MyLibrary::Resource->coverage
3. creator - MyLibrary::Resource->creator
4. date - MyLibrary::Resource->date
5. description - MyLibrary::Resource->note
6. format - MyLibrary::Resource->format
7. identifier - MyLibrary::Resource::Location

8. language - MyLibrary::Resource->language
9. publisher - MyLibrary::Resource->publisher
10. relation - MyLibrary::Resource->relation
11. rights - MyLibrary::Resource->rights
12. source - MyLibrary::Resource->source
13. subject - MyLibrary::Resource->subject
14. title - MyLibrary::Resource->name
15. type - MyLibrary::Resource->type

This mapping makes it relatively easy to store Dublin Core-based descriptions of information resources into a MyLibrary implementation. The items described in OAI-PMH data repositories come immediately to mind.

As a simple example of setting and getting values of MyLibrary resource objects, let's set and get a link to a fictional electronic version of The Adventures of Huckleberry Finn:

```
# create a resource object
$resource = MyLibrary::Resource->new;

# describe it
$resource->creator('Mark Twain');
$resource->format('ebook');
$resource->language('en');
$resource->name('The Adventures of Huckleberry Finn');
$resource->note('This is a coming of age story.');
$resource->subject('young adult reading');
$resource->type('text/html');

# give it a URL
$location_type = MyLibrary::Resource::Location::Type->new
  (name =>'URL');
$resource->add_location(location => 'http://library.org/finn.html',
  location_type => $location_type->location_type_id);

# save it
$resource->commit;

# get it
$id = $resource->id;
$resource = MyLibrary::Resource->new(id => $id);

# output the data
print '      Author: ' . $resource->creator     . "\n";
print '      Format: ' . $resource->format      . "\n";
print '    Language: ' . $resource->language    . "\n";
print '       Title: ' . $resource->name        . "\n";
print ' Description: ' . $resource->note        . "\n";
print '     Subject: ' . $resource->subject     . "\n";
print '   MIME type: ' . $resource->type        . "\n";

# get the url; assume there is only one
@locations = $resource->resource_locations;
print '         URL: ' . $locations[0]->location . "\n";
```

With the exception of the location attributes, this should be pretty straight-forward. (Remember, information resources can have more than one location and more than one location type. This is why setting and getting the location of resource objects is not as simple as the other attributes.)

While the procedure outlined above is functional, it is not necessarily complete. It does not take advantage of your facet/term combinations. Let's assume you have a facet called Subjects. Let's also assume you have the terms American Literature and Young Adult Reading assigned to the Subjects facet. Given this you can use the related_terms method to classify a resource with these terms. Very, very important! To get the terms back out you again use the related_terms method. It returns an array of term ids (keys):

```
# get the facet id for subjects
$facet = MyLibrary::Facet->new(name => 'Subjects');
$facet_id = $facet->facet_id;

# create the subject term amerian literature
$term = MyLibrary::Term->new;
$term->term_name('American Literature');
$term->term_note('Writings of the New World');
$term->facet_id($facet_id);
$term->commit;
$american_literature = $term->term_id;

# create the subject term young adult reading
$term = MyLibrary::Term->new;
$term->term_name('Young Adult Reading');
$term->term_note('Literature for the middle schoolers');
$term->facet_id($facet_id);
$term->commit;
$young_reading = $term->term_id;

# get huck finn and assume there is only one matching record
@resources = MyLibrary::Resource->new(
  name => 'The Adventures of Huckleberry Finn');
$resource = $resources[0];
$resource->related_terms
  (new => [$american_literature, $young_reading]);
$resource->commit;

# output the data
print '      Author: ' . $resource->creator     . "\n";
print '      Format: ' . $resource->format      . "\n";
print '    Language: ' . $resource->language    . "\n";
print '       Title: ' . $resource->name        . "\n";
print ' Description: ' . $resource->note        . "\n";
print '     Subject: ' . $resource->subject     . "\n";
print '   MIME type: ' . $resource->type        . "\n";

# get the url; assume there is only one
@locations = $resource->resource_locations;
print '         URL: ' . $locations[0]->location . "\n";

# get the related terms
@related_terms = $resource->related_terms;
```

```
foreach $rt (@related_terms) {

  # print the term name
  $term = MyLibrary::Term->new(id => $rt);
  print '       Term : ' . $term->term_name . "\n";

}
```

Read manage-resources.pl, subroutines.pl to learn how to implement these ideas in a terminal-based environment. See the pod of MyLibrary::Resource for more detail because there are many more methods to be found there.

A Zen Master once said, "Collections without services are useless, and services without collections are empty." Use MyLibrary to create a collection of information resources, and then use MyLibrary to provide services against the collection. Both are necessary in order to meet the expectations of today's users of libraries.

## Terms and resources revisited

Once you have created sets of MyLibrary resources and associated them with sets of MyLibrary terms you can exploit a couple more term methods to query your MyLibrary database.

You can use the MyLibrary::Term class method called related_resources to create a list of resources associated with a term. For example, suppose you have a term named Astronomy, then you could use the following code to list the names and descriptions of all those resources:

```
# require the necessary modules
use MyLibrary::Term;
use MyLibrary::Resource;

# get the id for the astronomy term, assume there is only one
@terms = MyLibrary::Term->get_terms
  (field => 'name', value => 'Astronomy');
$term = MyLibrary::Term->new(id => @terms[0]->term_id);
$astronomy = $term->term_id;

# create astronomy resources, #1 of 3
$resource = MyLibrary::Resource->new;
$resource->name('Stars amoung us');
$resource->note('No, not movie stars');
$resource->related_terms(new => [$astronomy]);
$resource->commit;

# resource #2 of 3
$resource = MyLibrary::Resource->new;
$resource->name('Guiding lights');
$resource->note('Soap operas and beyond');
$resource->related_terms(new => [$astronomy]);
$resource->commit;

# resource #3 of 3
```

```
$resource = MyLibrary::Resource->new;
$resource->name('My Guide the the Galaxy');
$resource->note('As if the Hitchhikers was not good enough');
$resource->related_terms(new => [$astronomy]);
$resource->commit;

# get all astronomy resources through the term
$term = MyLibrary::Term->new(id => $astronomy);
@resource_ids = $term->related_resources;

# display information about the resources
foreach $id (@resource_ids) {

    $resource = MyLibrary::Resource->new(id => $id);
    print ' Name: ' . $resource->name . "\n";
    print ' Note: ' . $resource->note . "\n\n";

}
```

The suggested_resources method allows you to set and get lists of resource ids determined to be particularly useful. Think recommendations. For example, suppose there is a resource called Most Cool Astronomy Site. Suppose also it is determined that this particular site lives up to its name and when displaying lists of astronomy resources you would like to highlight this one in particular. To do this you would first use the suggested_resources method set this value:

```
# get the id for astronomy; assume there is only one
@terms = MyLibrary::Term->get_terms
  (field => 'name', value => 'Astronomy');
$term = MyLibrary::Term->new(id => @terms[0]->term_id);
$astronomy = $term->term_id;

# create an astronomy resource
$resource = MyLibrary::Resource->new;
$resource->name('Most Cool Astronomy Site');
$resource->related_terms(new => [$astronomy]);

# save and get its id (key)
$resource->commit;
$id = $resource->id;

# get the astronomy term
$term = MyLibrary::Term->new(id => $astronomy);

# denote our resource as a suggested item for astronomy, and save
$term->suggested_resources(new => [$id]);
$term->commit;
```

You can then list all the resources associated with a term and then specify which ones are recommended:

```
# get the id for astronomy; assume there is only one term
@terms = MyLibrary::Term->get_terms
 (field => 'name', value => 'Astronomy');
$term = MyLibrary::Term->new(id => @terms[0]->term_id);
```

```
# get all astronomy resource ids and suggestion ids
@resources   = $term->related_resources(sort => 'name');
@suggestions = $term->suggested_resources;

# process each resource
foreach $r (@resources) {

    # get the resource and print its name
    $resource = MyLibrary::Resource->new(id => $r);
    print ' Name: ' . $resource->name;

    # loop through each suggestion
    foreach $s (@suggestions) {

      # compare suggestion and resource ids
      if ($s == $r) {

        # specify this as suggested resource
        print ' (suggested)';
        last;

      }

    }

  print "\n";

}
```

You will often want work with entire sets or subsets of resources from your MyLibrary implementation, and the get_resources method is used for this purpose. Once you get the set of resources you are expected to loop through them and extract the ones you really need. Here is a simple way to get all the resources as objects and print their names:

```
# get all resources and display
@resources = MyLibrary::Resource->get_resources;
foreach $resource (@resources) { print $resource->name . "\n" }
```

You can do the same thing, but return a sorted list

```
# get a sorted list, by name, of resources
@resources = MyLibrary::Resource->get_resources(sort => 'name');
foreach $resource (@resources) { print $resource->name . "\n" }
```

Besides the basic Dublin Core elements, MyLibrary allows you to assign additional attributes to resources. The first of note is foreign key through the fkey method. This is intended to store things like OCLC numbers, catalog record numbers, or OAI identifiers in MyLibrary resource objects. By combining the fkey values with things like URL it is often possible to link to back to some other list of information resources. You set and get fkey values just like most of the other attributes:

```
# create a resource
$resource = MyLibrary::Resource->new;
```

```
# set the name and fkey value
$resource->name('Tom Sawyer Rides Again');
$resource->fkey('123457');
$resource->commit;

# print it
print ' Foreign key: ' . $resource->fkey . "\n";
```

The lcd ("lowest common denominator") method is intended to denote information resources that are useful to anybody, not restricted to any MyLibrary term. For example, most librarians will believe the catalog is a tool useful for everybody for every discipline. A general encyclopedia and dictionary are other examples. Denote a resource as a "lowest common denominator" resource like this:

```
# create and denote a resource as an "lcd" resource
$resource = MyLibrary::Resource->new;
$resource->name('Library catalog');
$resource->lcd(1);
$resource->commit;
```

To get a list of all the resource objects denoted as lowest common denominator resources, use the class method lcd_resources:

```
# get all "lcd" resources and display
@lcd_resources = MyLibrary::Resource->lcd_resources;
print "These resources are useful to everyone:\n";
foreach $r (@lcd_resources) { print "\t" . $r->name . "\n" }
```

Through the qsearch_prefix, qsearch_suffix, and qsearch_redirct methods you are able to reverse engineer many Internet search engines. Take a simple Google search for the word cat, http://www.google.com/search?q=cat. This URL can be divided into at least three parts: 1) the root (http://www.google.com/), 2) a prefix (search?q=), 3) the query itself (cat), and 4) an optional suffix (null in this example).

You might use this code to create a resource for Google and add a search prefix to it:

```
# create a resource describing Google
$resource = MyLibrary::Resource->new;
$resource->name('Google');
$resource->note('A very popular Internet index');

# get the location type of URL
$location_type = MyLibrary::Resource::Location::Type->new
  (name => 'URL');
$type_id = $location_type->location_type_id;

# give the resource a URL
$resource->add_location(location => 'http://www.google.com/',
                        location_type => $type_id);

# add a quick search prefix and save
$resource->qsearch_prefix('search?q=');
```

```
  $resource->commit;
  $id = $resource->id;

  # begin echoing results
  $resource = MyLibrary::Resource->new(id => $id);
  print '  Title: ' . $resource->name . "\n";
  print '   Note: ' . $resource->note . "\n";

  # get the location; assume there is only one
  @locations = $resource->resource_locations;

  # echo more results
  print '    URL: ' . $locations[0]->location . "\n";
  print ' Prefix: ' . $resource->qsearch_prefix . "\n";
```

Now suppose you have some sort of HTML form that accepts text input. Using the qsearch_redirect method the input can be transformed into a URL to search the resource. Something like this:

```
  # get a query
  $query = 'foobar';

  # get the Google resource; assume there is only one
  @resources = MyLibrary::Resource->new(name => 'Google');
  $resource  = $resources[0];

  # get the location; assume there is only one
  @locations = $resource->resource_locations;
  $root_url  = $locations[0]->location;

  # build a URL to search
  $url = $resource->qsearch_redirect(resource_id => $resource->id,
                                     qsearch_arg => $query,
                                     primary_location => $root_url);

  # display in an HTML snippet
  print "<a href='$url'>Click here</a> to search Google for
'$query'.\n";
```

Take this technique a step further. Suppose your MyLibrary implementation contains records from your library catalog. Suppose each MyLibrary resource record includes an fkey value pointing to the full record in the catalog. Suppose that you also have a MyLibrary record describing your catalog, and that record is complete with a qsearch_prefix and optional qsearch_suffix values. Using the qsearch_redirect method you could display brief records on a Web page and link back to the full record in your library catalog by using the fkey value as the qsearch_arg attribute.

## Implementing search

This section outlines a method for making the content of your MyLibrary implementation searchable.

MyLibrary is essentially a relational database application. As such, searching the database requires queries be converted into SQL. By definition these SQL queries must specify what fields to search. Unfortunately people expect to perform freetext queries against sets of content, not necessarily fielded searches. Moreover, people increasingly expect relevancy ranked output as well as output sorted by this, that, and the other thing. For these reasons you are encouraged to use an intermediary indexing application to implement searchability instead of querying the database directly.

Making your MyLibrary content searchable through an intermediate indexer uses this process:

1. Write a report against MyLibrary.
2. Feed the report to the indexer.
3. Index the report.
4. Provide a Perl interface to search the index.
5. Search results are integers -- MyLibrary database keys.
6. Use the keys to get data from MyLibrary.
7. Reformat the data for display and return it to the user.

Swish-e is a good indexer. Simple, fast, and it comes with a Perl API. This will be the indexer in this example, but something like Plucene would work just as well.

The first step is to write a report against the MyLibrary database. Swish-e expects its input to look much like HTML. The following code outputs a very simple report containing only the titles and notes of every resource in a MyLibrary instance. The report is in a form swish-e expects:

```
# require
use MyLibrary::Resource;

# first, get all of the resource ids
@resource_ids = MyLibrary::Resource->get_ids;

# process each id
foreach $resource_id (@resource_ids) {

     # get a resource
     $resource = MyLibrary::Resource->new(id => $resource_id);

     # get its id, title, and note
     $id    = $resource->id;
     $title = $resource->name;
     $note  = $resource->note;

     # build the report
     $output = '';
     $output .= '<html>';
     $output .= '<head>';
     $output .= "<meta name='title' content='$title' />";
     $output .= "<meta name='note'  content='$note' />";
     $output .= '</head>';
```

```
        $output .= '<body>';
        $output .= "<h1>$title</h1>";
        $output .= "<p>$note</p>";
        $output .= '</body>';
        $output .= '</html>';

        # output a swish-e header
        print "Path-Name: $id\n";
        print "Content-length: " . scalar(length $output) . "\n";
        print "Document-Type: HTML*\n";
        print "\n";

        # output the report
        print "$output";

}
```

The next step is to build a configuration file telling swish-e what to look for in the report. In our case the configuration needs to know about the title and note, and the configuration could be this simple:

```
# define fields to index and make searchable
MetaNames title note
PropertyNames title note

# define the location and name of the resulting index
IndexFile ./mylibrary.idx
```

Finally, you need to run your report generator and pipe the output to swish-e specifying the configuration file. You can do this from the command line. Assuming your report generator is called mylibrary2swish.pl and your swish-e configuration file is called mylibrary2swish.cfg, then the command might look like this:

```
./mylibrary2swish.pl | swish-e -c ./mylibrary2swish.cfg \
-S prog -i stdin
```

The result should be a two files, mylibrary.idx and mylibrary.idx.prop. These files are your swish-e index and you should be able to search them from the command line using the swish-e binary. Remember, you only included name and note in your output so only those fields will be searched. Also, queries will only return ids, not words.

The next step is to provide the ability to search the index. As queries are accepted the swish-e Perl API is used to search the index. Queries return MyLibrary keys, and these keys are used to look up the values of resources:

```
# require the necessary modules
use MyLibrary::Resource;
use SWISH::API;

# define the location of your index
$INDEX = './mylibrary.idx';

# get the input
```

```
print "Enter a query. ";
chop (my $query = <STDIN>);

# search the index
$swish   = SWISH::API->new($INDEX);
$results = $swish->Query($query);
$hits    = $results->Hits;

# branch according to the number of hits
if ($hits) {

      print "Your search ($query) returned $hits hit(s).\n\n";
      $counter = 0;

      # process each result
      while ($result = $results->NextResult) {

            # get the id (key)
            $id = $result->Property("swishdocpath");

            # get the resource, title, and note
            $resource = MyLibrary::Resource->new(id => $id);
            $title    = $resource->name;
            $note     = $resource->note;

            # increment the counter
            $counter++;

            # print the result
            print "$counter. $title - $note\n\n";

      }

}

else {

      print "No hits. Sorry.\n";

}
```

This section has outlined the most basic of search interfaces. Your reports sent to swish-e will want to be much more verbose.

For more information see index-resources.pl, index-resources.cfg, resources2swish.pl, and search.pl that came with the distribution. These files implement a more full-featured, terminal-based program for search.

## MyLibrary and OAI

Because the MyLibrary database so closely resembles the basic Dublin Core elements, and because OAI requires data repositories to support Dublin Core, it is almost trivial to harvest the content of OAI repositories and cache it to a MyLibrary database.

The following script does just that, and in a nutshell here is how it works:

1. Define the repository to harvest.
2. Create a facet called Formats, if it doesn't exist.
3. Create a term called Images, if it doesn't exist.
4. Create a location type called URL, if it doesn't exist.
5. Harvest all the records from the repository.
6. Loop through each harvested record.
7. Create a MyLibrary resource object.
8. Fill the resource with attributes.
9. Save the resource.
10. Go to Step #6 'till done.

```
# include the necessary modules
use MyLibrary::Core;
use Net::OAI::Harvester;

# define the repository
$repository = 'http://infomotions.com/gallery/oai/index.pl';

# check for a facet called Formats
$facet = MyLibrary::Facet->new;
if (! MyLibrary::Facet->get_facets
  (value => 'Formats', field => 'name')) {

      # create it
      $facet->facet_name('Formats');
      $facet->facet_note
        ('Types of physical items embodying information.');
      $facet->commit;
      print "\nThe facet Formats was created.\n";

}

else {

      # already exists
      $facet = MyLibrary::Facet->new(name => 'Formats');
      print "\nThe facet Formats already exists.\n";

}

# remember this facet id
$facetID = $facet->facet_id;

# check for a term named Images
$term = MyLibrary::Term->new;
if (! MyLibrary::Term->get_terms
  (value => 'Images', field => 'name')) {

      # create it
      $term->term_name('Images');
      $term->term_note('Photographs or paintings.');
      $term->facet_id($facetID);
      $term->commit;
```

```
      print "The term Images was created.\n";

}

else {

      # it already exists
      $term = MyLibrary::Term->new(name => 'Images');
      print "The term Images already exists.\n";

}

# remember this term id
$imageTermID = $term->term_id;

# check for a location type called URL
$location_type = MyLibrary::Resource::Location::Type->new;
if (! MyLibrary::Resource::Location::Type->new(name => 'URL')) {

      # create it
      $location_type->name('URL');
      $location_type->description('A type of Internet resource.');
      $location_type->commit;
      print "The location type URL was created.\n";
}

else {

      # it already exists
      $location_type = MyLibrary::Resource::Location::Type->new
        (name => 'URL');
      print "The location type URL already exists.\n";

}

# remember the location type id
$location_type_id = $location_type->location_type_id;

# create a harvester and get the data
$harvester = Net::OAI::Harvester->new('baseURL' => $repository);
$records = $harvester->listAllRecords('metadataPrefix' => 'oai_dc');

# process each record
while ($record = $records->next) {

      $FKey         = $record->header->identifier;
      $metadata     = $record->metadata;
      $name         = $metadata->title;
      @description  = $metadata->description;
      $description  = join (' ', @description);
      $location     = $metadata->identifier;
      print "$name...";

      # check to see if it already exits
      if (! MyLibrary::Resource->new(fkey => $FKey)) {

            # create it
            $resource = MyLibrary::Resource->new;
```

```
            $resource->name($name);
            $resource->note($description);
            $resource->fkey($FKey);
            $resource->related_terms(new => [$imageTermID]);
            $resource->add_location
              (location => $location,
               location_type => $location_type_id);
            $resource->commit;
            print "added (", $resource->id, ").\n";

      }

      else {

            # already got it
            print "already exists.\n";

      }

}

# done
print "\nDone\n";
exit;
```

While this was the longest example in this tutorial, this particular OAI to MyLibrary interface is very rudimentary. See the script named images2mylibrary.pl from the distribution to see how you can harvest OAI sets. See doaj2mylibrary.pl to see how you can more accurately classify incoming resources based on set names. The really enterprising reader will figure out ways to read the incoming Dublin Core subject fields and create facet/term combinations accordingly.

## Summary

Unlike version 2.x of MyLibrary, version 3.x is more like a toolbox and less like a turn-key application. Developers are expected to read and write values to the MyLibrary database, manipulate these values to create sets of information services.

The examples above point to terminal-based scripts implementing the described concepts. The distribution comes with another set of scripts implementing these ideas using a Web-based interface. They use all of the concepts outlined above but they are CGI scripts implemented in a more graphical interface.

Use MyLibrary in conjunction with other Perl modules. In a more traditional library you might consider reading sets of MARC records to create a sort of online catalog. Provide an SRU interface to your indexed content and then transform the XML returned from the SRU server into email messages or RSS feeds. Create CGI scripts that return Javascript that simply write to the document window. Then call these CGI scripts from within HTML <script> elements. This will enable HTML authors to incorporate MyLibrary content into their pages. You might harvest data from various but similar OAI repositories to create subject-specific collections. Index the collection and provide an

interface to it. You might create Web-based input screens allowing authors to submit information about publications thus implementing a sort of institutional repository. Use your imagination. Think a bit outside the box.

When you've got a hammer everything looks like a nail. While MyLibrary is not necessarily a perfect hammer, it can address many of the needs in libraries to create, maintain, and distribute classified lists of information resources. The key to success is discovering ways to re-purpose these lists meeting the expressed needs of library users.