Design and Demonstration

Chapter 5

The prototype needed to be simple enough so that it could be developed with minimal coding. It was decided to use only open source tools that are freely available.

Following three major steps were required in developing the prototype:

- Setting up a local metadata index
- Develop tools to harvest, crosswalk, and index the metadata
- Develop a search interface to the index

5.1 Indexing Setup

Structured data, such as metadata, are mostly stored in databases. The indexes are created for search from specific columns in a database. Since, the prototype was only focused on storing metadata and retrieving them for search it was decided to use Apache Solr (http://lucene.apache.org/solr/), a popular open source enterprise search platform based on the Apache Lucene
Solr Schema design

The primary step in setting up a Solr index is designing the schema, which is saved in the schema.xml file. In the schema, several properties such as name, data type, repeatability are defined for each field.

The sample schema.xml, which is installed by default, was customized to include the Dublin Core fields. The <fields> element was modified in the schema.xml as showed below:

```xml
  <fields>
    <!-- DC fields -->
    <field name="id" type="string" indexed="true" stored="true" />
    <field name="title" type="text" indexed="true" stored="true" />
    <field name="creator" type="text" indexed="true" stored="true" multiValued="true" />
    <field name="subject" type="text" indexed="true" stored="true" multiValued="true" />
    <field name="description" type="text" indexed="true" stored="true" />
    <field name="publisher" type="text" indexed="true" stored="true" />
    <field name="contributor" type="text" indexed="true" stored="true" multiValued="true" />
  </fields>
```
CHAPTER 5. DESIGN AND DEMONSTRATION

<field name="date" type="string" indexed="false" stored="true"
   multiValued="true" />
<field name="type" type="string" indexed="true" stored="true"
   />
<field name="format" type="string" indexed="true" stored="true"
   multiValued="true" />
<field name="identifier" type="string" indexed="true" stored="true"
   multiValued="true" />
<field name="source" type="string" indexed="true" stored="true"
   />
<field name="language" type="string" indexed="true" stored="true"
   multiValued="true" />
<field name="relation" type="string" indexed="false" stored="true"
   />
<field name="coverage" type="string" indexed="false" stored="true"
   />
<field name="rights" type="string" indexed="true" stored="true"
   />
<!— Other Fields —>
<field name="repository" type="string" indexed="true" stored="true"
   />
<field name="url" type="string" indexed="true" stored="true"
   multiValued="true" />
<field name="institution" type="string" indexed="true" stored="true"
   />
<field name="journal" type="text" indexed="true" stored="true"
   />
<field name="issn" type="text" indexed="true" stored="true"
   multiValued="true" />
<field name="doi" type="text" indexed="true" stored="true"
   />
<field name="author" type="text" indexed="true" stored="false"
   multiValued="true" />
<field name="allfields" type="text" indexed="true" stored="false"
   multiValued="true" />
</fields>
The steps of installation, configuration and startup details along with complete listing of the `schema.xml` are provided in the Appendix.

### 5.2 Harvesting and Indexing Scripts

It was decided to use Perl scripts to harvest, parse, and index metadata into Solr from various sources. Perl provides a number of built-in tools (known as *modules* in Perl idiom) for these purposes which are available for free from the Comprehensive Perl Archive Network (CPAN) (http://www.cpan.org/).

#### 5.2.1 Harvesting from OAI

The `Net::OAI::Harvester` [6] module provides the functions to harvest metadata from OAI-compliant sources. It uses standard XML parsers to convert the XML response from OAI sources into meaningful objects for manipulation and indexing. For example, harvesting all metadata from Librarian's Digital Library in `oai_dc` we can use the following code:

```perl
#!/usr/bin/env perl
use strict;
use Net::OAI::Harvester;

# Set the defaults
use constant BASEURL => 'https://clrtc.isibang.ac.in/oai/request';
use constant SET => 'Library_and-Information.Science';
use constant MPREFDC => 'oai_dc';

# Initialize the harvester object
my $harvester = Net::OAI::Harvester->new(
    baseURL => BASEURL,
    dumpDir => '/opt/oai.drtc',
);
```
CHAPTER 5. DESIGN AND DEMONSTRATION

```
my $records = $harvester->listAllRecords( set => SET,
    metadataPrefix => MPREFDC );
```

The above script will harvest metadata of all articles in the 'Library and Information Science' set from the Librarian's Digital Library in Dublin Core format. In this example, the $records object will store all the metadata, which can be passed on to the indexing function.

### 5.2.2 Collecting RSS/Atom feeds

The LWP::Simple [1] module provides a simple way to interact with web pages using HTTP, which can be used to download the RSS/Atom feeds. The XML::RSS [4] module provides methods to parse RSS/Atom feeds into Perl objects for further manipulation.

The following code listing is an example of collecting RSS feed from D-Lib Magazine:

```
#!/usr/bin/env perl

use strict;
use XML::RSS;
use LWP::Simple;

# define URLs
use constant RSSURL => 'http://www.dlib.org/rss/dlib.rss';

# initialize object
my $rss = new XML::RSS();

# get RSS data
print "trying to connect to ", RSSURL, "\n";
my $raw = get( RSSURL );

print "Now parsing data ...\n";
```
CHAPTER 5. DESIGN AND DEMONSTRATION

```perl
$srss->parse($raw);
print "RSS Feed title: " . $rss->channel('title') . "\n";
```

The above script shows the basic steps in collecting RSS feed into an object for manipulation. In this example, the $rss object can be passed on to indexer function.

### 5.2.3 Harvesting ToC Pages

For harvesting from HTML pages of the Table of Contents, WWW::Mechanize was used which simulates a browser to interact with web pages [5]. It provides powerful methods to automate not only the crawling of web pages, but navigating and parsing the contents as well. Following is a sample script for harvesting D-Lib Magazine, using this module:

```perl
#!/usr/bin/env perl
use strict;
use WWW::Mechanize;

# Define D-Lib archive page URL
use constant BASEURL => 'http://www.dlib.org/back.html';

# Initialize crawler
my $mech = WWW::Mechanize->new( autocheck => 1);
$mech->agent_alias( 'Linux Mozilla' );
$mech->get( BASEURL );

# Find links to all relevant archive pages
my @links = $mech->find_all_links(
    url_regex => qr/back((19|20)\d{2})\s\html$/
    # e.g., http://www.dlib.org/back2009.html
);
my ($url, $num);
```
CHAPTER 5. DESIGN AND DEMONSTRATION

# Find links to the ToC pages from archives pages
foreach my $link (@links) {
    $url = $link->url_abs; # find the 'absolute' URL
    print $url, "\n";
}

# This can included into previous 'foreach' loop
if ( $url ) {
    $mech->get( $url );
    my @toc_links = $mech->find_all_links(
        url_regex => qr/\d+contents\.html$/ );
    my $toc;
    foreach ( @toc_links ) {
        $toc = $-->url-abs;
        last if $toc;
    }
    print $toc, "\n";
    # e.g., http://www.dlib.org/dlib/january09/01contents.html
    $mech->get( $toc ); # download the HTML page
    # extract metadata contents into an object
    my @articles = get_toc( $mech->content );
}

# Function (subroutine) to extract metadata from ToC pages of D-Lib Magazine
sub get_toc {
    my $toc_html = shift;

    # Remove unwanted top area
    $toc_html =~ s/.+ARTICLES//si;

    # remove bottom stuff
    $toc_html =~ s/<div class="divider-full">\s*<\/div>\s*$/;
my @articles = split(/<div class="divider-full"/>, $toc.html);
my @papers;
for my $article (@articles) {
my ($paper, $author, $title, $abstract, $doi, $url) = ();
my @lines = split( /(</br\>]|</p>)/, $article);
for my $ln (@lines) {
    $ln =~ s/\s+/ /g;       # Squash whitespace
    $ln =~ s/^\ //;         # Strip leading space
    $ln =~ s/ $//;          # Strip trailing space
    $ln =~ s/#\d+/ /;       # HTML entities
    next if $ln =~ /'\s+$/;
    # Parse metadata
    if ($ln =~ /<a href="[^>]*">(.+)</a>/i && $title) {
        ($url, $title) = ($1, $2);
    }
    if ($ln =~ /Article by (.+)i && $author) {
        $author = $1;
    }
    if ($ln =~ /doi:/i && $doi) {
        $ln =~ s/\r\n//; $doi = $ln;
    }
    if ($ln =~ /Abstract:<(\s+.+)>/i && $abstract) {
        $abstract = $1;
    }
}
if ($url) {
    $paper->{'title'} = $title;
    $paper->{'doi'} = $doi;
    $paper->{'url'} = $url;
    $paper->{'author'} = $author;
    $paper->{'abstract'} = $abstract;
    push @papers, $paper;
}
return @papers;
The above listing is a basic code written for extracting metadata (title, author, URL, abstract, and DOI) from D-Lib Magazine archive pages. It can be enhanced to parse other details as well.

5.2.4 Index into Solr

For indexing metadata into the Solr index the program needs to connect to the web service of Solr. The WebService::Solr module provides the methods for various transactions with the Solr index using its web service.

This example shows indexing the oai_dc metadata harvested from OAI-compliant repositories into Solr index:

```perl
#!/usr/bin/env perl

use strict;
use WebService::Solr;

# Define Solr base URL
use constant SOLR => 'http://localhost:8983/solr';

# Create Solr object
my $solr = WebService::Solr->new(SOLR);
my $num = 0;

# Index metadata in a loop
foreach my $record (@records) {
    $num++;

    # Solr document
    my $doc = WebService::Solr::Document->new;

    # extract the desired metadata
    my $meta = $record->metadata;
    $doc->add_fields(id => $record->header->identifier);
    print $num, " => ", $record->header->identifier, "\n";
```
my @fields = qw( title creator subject description publisher
contributor
date type format identifier source language relation coverage
rights );

foreach my $field (@fields) {
    next if !$meta->{$field};

    my $solr_field;

    # Non-repeatable fields
    if ( my @elements = ($meta->{$field}) ) {
        foreach my $element (@elements) {
            if ( $field ='/identifier/' && $element !~ /http/i ) {
                $solr_field = WebService::Solr::Field->new( journal =>
                    $element );
            } else {
                $solr_field = WebService::Solr::Field->new( $field =>
                    $element );
            }
        }
    }

    $doc->add_fields( $solr_field );
}

# index to solr
$solr->add( $doc );

print "Indexed $num records.\n";
CHAPTER 5. DESIGN AND DEMONSTRATION

5.3 Search Interface Development

For the prototype, the search interface was developed in Perl using the Catalyst Web Framework. It provides the Model-View-Controller (MVC) architecture [8], which isolates the application logic (Controller) from the user interface (View) and the database (Model), permitting independent development, testing and maintenance of each. The Catalyst framework not only provides faster development, but also easier maintenance of application. [3]

The Catalyst framework also allows code re-use. The WebService::Solr [2] module was used as the Model for connecting to the Solr web service for searching the index. The Template::Toolkit [7] was used as the View for displaying the search results.

The heart of the search interface is the search controller which was created inside the Root.pm file with the following code:

```perl
sub search :Global Args(0) {
    my ( $self , $c ) = @_;

    my ($query, $qterm, $response, %options);

    # Parse search query
    if ( $c->req->param("q") ) {
        $qterm = $c->req->param("q");
        $query = $qterm;

        # Apply the template
        $c->stash(template => 'result.tt');
    } else {
        $c->stash(template => 'search.tt');
    }

    # Show the results
    if ( $query ) {
```

# get page size
$options{'rows'} = $c->req->param("rows") || 10;

# set the first page
$my $page = $c->req->param("page") || 1;

# set start position
$options{'start'} = ($page - 1) * $options{'rows'};

# Invoke search query against Solr index
$my $response = $c->model('Solr')->search($query, 
{options});

# Capture the raw JSON result
$my $json = $response->{raw_response}->{.content'};
$my $pager = $response->pager;

# Convert Solr response into JSON object
# (requires 'use JSON;')
$my $pjson = from_json($json);

# The complete Solr response as perl hash object
$my $content = $pjson->{response'};

# Post the search result
if ( $pager ) {
   $c->stash(
      query => $qterm,
      docs => $content->{docs'},
      pager => $pager,
   );
} else {
   $c->stash(
      query => $qterm,
      feedback => "No records found for 
      <b>$qterm</b>",
   );
}
The above listing uses the WebService::Solr module to connect to Solr index and download results in JavaScript Object Notation (JSON) format. It also converts it into an object before passing it to the View for display as HTML in the browser.

Further listings of codes, templates, and other details of the setup are explained in the Appendix.

5.4 Demonstration

The metadata schema used for building index in Solr software, shown in the Fig. 5.1, was based on Dublin Core.

![Figure 5.1: Solr index schema](http://localhost:8983/solr/admin/file/?file=schema.xml)
5.4.1 Harvesting and Indexing from DOAJ

Fig. 5.2 and Fig. 5.3 shows the indexing process of harvesting metadata records from DOAJ. For the purpose of demonstration, the harvesting was limited to 1000 records. Instead of the default Dublin Core, this script harvested the metadata in `doajArticles` schema since it was richer in metadata elements.

![Indexing from DOAJ: start screen](image)

Figure 5.2: Indexing from DOAJ: start screen
The OAI data provider module of DOAJ limits the harvesting to 50 records per request by issuing a resumptionToken. The above script includes the resumptionToken in subsequent requests to continue harvesting after every 50 records.

### 5.4.2 Indexing RSS feed

The following screenshot, Fig. 5.3, shows identifier of each article being extracted from the RSS feed of D-Lib Magazine for the purpose of indexing.
Figure 5.4: Indexing from RSS feed: D-Lib Magazine

5.4.3 Indexing Table of Contents pages

The following two screenshots in Fig. 5.5 and Fig. 5.6 show titles of each article being extracted from online table of contents of two e-journals for the purpose of indexing.
Figure 5.5: Indexing from ToC page: D-Lib Magazine

Figure 5.6: Indexing from ToC page: Ariadne Magazine
5.4.4 Search examples

Metadata from RSS

Fig. 5.7 and Fig. 5.8 are two screenshots showing search results for metadata indexed from RSS feeds of two e-journals.

Figure 5.7: Search results of RSS contents: D-Lib Magazine
Figure 5.8: Search results of RSS contents: Ariadne Magazine

Metadata from ToC

In Fig. 5.9 and Fig. 5.10, we can see search results for metadata indexed from online table of contents pages of two e-journals.
CHAPTER 5. DESIGN AND DEMONSTRATION

Figure 5.9: Search results of ToC contents: D-Lib Magazine

Figure 5.10: Search results of ToC contents: Ariadne Magazine
Metadata from OAI resources

The Fig. 5.11 and Fig. 5.12 are showing search results with metadata indexed from OAI-compliant resources of Librarian’s Digital Library (LDL) and DOAJ.

Figure 5.11: Search results of OAI data: DRTC LDL
CHAPTER 5. DESIGN AND DEMONSTRATION

5.5 Conclusion

From the demonstration of search results above it is obvious that the quality of metadata from OAI-compliant resources is richer than RSS feeds. However, the metadata extracted from online table of contents are remarkably of high quality. It demonstrates that web scraping and HTML heuristics can be effectively used to extract rich metadata from seemingly unstructured table of contents pages of e-journals.

5.6 Bibliography


