Programmer's Guide: Using LucidWorks Enterprise to add search to your Web Application

Easily index web pages and
other content for powerful, flexible,
extensible search

© 2010 by Lucid Imagination, Inc. under the terms of Creative Commons license, as detailed at http://www.lucidimagination.com/Copyrights-and-Disclaimers/. Version 1.02, published 6 June 2010. Solr, Lucene, Apachecon and their logos are trademarks of the Apache Software Foundation.

Abstract

Creating search functionality can be the most daunting part of putting together an application. Even if you can get past the notion of indexing every single piece of content you want to make available, there's still the issue of performing the actual search and providing results based on relevancy. It's a complex task, and one many developers are loath to tackle. A search server application such as Apache Solr makes searching easier, but indexing unstructured content can still be a bit of a challenge.

LucidWorks Enterprise, built on Apache Solr/Lucene, goes the extra mile to make that part easy too. This step-by-step guide shows takes a programmatic approach to working with LucidWorks Enterprise, the search solution from LucidWorks Enterprise built on Apache Solr. We'll show how to add content, index it, create a web page (with example PHP code) to send search requests and display results, add search fields using the REST API, and then add search features such as faceted search. Code samples and screen shots are included.

Table of Contents

Introduction 1
Setting up 2
  Install LucidWorks Enterprise 2
  Prepare the web application environment 2
How a search engine works 3
  Search application development workflow 4
Indexing content with LucidWorks Enterprise 4
Adding search to your site 13
Getting better control: advanced searching 18
  Adding fields using the LucidWorks Enterprise UI 18
  Adding fields using the LucidWorks Enterprise ReST API 20
  Indexing the data 21
  Performing the search 24
Faceting and other enhancements 32
Summary 37
Next Steps 38

Introduction

Today's computer users have been conditioned to "navigate by search". They know what they want – or at least, they think they do – and they want sites and applications to simply give it to them. That can make it hard on the developers of applications – especially those building web applications – because implementing search can be a difficult and time consuming process. It requires the ability to not only analyze your content, which may be in many different forms, but also the ability to understand a user's query and determine the relevance of the different content items that satisfy that query. And that's just for basic search capabilities! Once you start adding in advanced searches, faceting, and other expected features, you've committed yourself to quite a task.

Fortunately, search doesn't have to be that difficult. For several years now, developers have been relying on the power of Apache Solr and Apache Lucene to build the foundation for their search functionality, but the mechanics involved have remained error-prone and labor intensive.

Enter LucidWorks Enterprise. Built on Solr, LWE gathers some of the most useful (and potentially difficult) enhancements, such as indexing web or database content, adds in additional features such as user management and click-scoring to improve results, and wraps it all up in one easy to use, web-based bundle.

All of this takes search from an nearly impossible task requiring very deep pockets to a basic functionality that's within reach of virtually any developer or organization.

In this paper, we're going to look at how to use LucidWorks Enterprise to easily add search capabilities to your web application. We'll start by looking at how to search existing static content, move on to indexing raw data, and then look at providing advanced search for your users, enabling them to search different attributes of your content. We'll finish up with a quick look at how you can potentially use your search function to generate content, and the potential advantages that could bring.

Code in this tutorial is shown using PHP, but the concepts apply to any programming language. To follow along, you should have basic programming experience.

Setting up

In order to see our search in action, we're going to add search capabilities to a fictional site called Pop Culture Journal, or PCJ. We'll be working on PCJ in future articles and tutorials, building it into a site where users can share recommendations on movies, books, television shows, and other, well, pop culture, so you can see how to perform basic tasks such as adding or editing documents within the search index, or more advanced features such as user management. But for now, we're just going to add the ability to search movies.

If you want to follow along, please make sure to do the following first:

Install LucidWorks Enterprise

To install LucidWorks Enterprise, execute the following steps:

1. LWE is a Java application, so if it's not installed, you will need Java 1.6 or higher. You can download the Java 1.6 SDK from http://www.oracle.com/technetwork/java/javase/downloads/index.html.
2. Download LucidWorks Enterprise from http://www.lucidimagination.com/lwe/download.
3. Run the LucidWorks Enterprise installer. You can find instructions at http://www.lucidimagination.com/developers/whitepapers/getting-started-with-lucidworks-enterprise, but in general, the default parameters will work just fine.
4. After LWE starts, point your browser to http://localhost:8989/search to verify that the software is installed and working properly.

Prepare the web application environment

Ultimately, of course, you're going to be adding search to your own application, but for the sake of demonstration, you can install the Pop Culture Journal files to see how everything fits together with the sample data. To install Pop Culture Journal, follow these steps:

1. Install a PHP environment. You can work with LWE in virtually any programming language, but for the examples in this paper we'll be using PHP. The XAMPP package, which includes Apache and PHP (as well, as MySQL and other tools) is available for Linux, Windows, MacOS, and Solaris, and includes the cURL utility, which we'll also need. You can download it from http://www.apachefriends.org/en/xampp.html.

2. Enable cURL in your application environment. We'll be using cURL later, in order to take advantage of LWE's ReST APIs, so you'll need to have that functionality available. If you're using XAMPP, all you need to do is uncomment `extension=php_curl.dll` in the `php/php.ini` file and restart the server.
3. Download the Pop Culture Journal files from http://www.lucidimagination.com/files/ASTYWA_PCJ.zip.
   This download includes the SolrPHPClient package, downloadable from http://code.google.com/p/solr-php-client/downloads/list, as well as files using the Open Source Web Design Template "ditto ditto", downloadable in its raw state from http://www.oswd.org/design/information/id/1996. It also includes data from the Freebase project, originally found at http://download.freebase.com/datadumps/latest/browse/film.tar.bz2.
4. Unzip the PCJ project files and place them in the document root for your web server. For XAMPP, this is `XAMPP_DIR/htdocs`.

This download includes a basic static version of the web site, and all of the code files.

How a search engine works

Before we actually build anything, it's helpful to understand exactly what happens when your users enter a keyword and press "Search".

The first thing to understand is that a search platform such as LWE does not, as a rule, search your content. Instead, it searches an index of the content. For example, if we had a document such as:

Document #THX1138:

```
A long time ago, in a galaxy far, far away…
```

The index might look something like this (at least on a conceptual level):

```
THX1138: a
THX1138: long
THX1138: time
THX1138: ago
THX1138: in
THX1138: a
THX1138: galaxy
THX1138: far
```

```
THX1138: far
THX1138: away
```

So when a user enters a search for

```
galaxy
```

The search platform finds "`THX1138: galaxy`" and knows to return document THX1138.

The advantage here is that any manipulations that need to be done, such as converting to lowercase, or chunking the data into various size pieces, or any other manipulations, are already done.

The information LWE returns about that document depends on what information was stored, as opposed to indexed. For example, for a simple document like this, we would likely have stored the entire text. For a web page, we might store the title and a description so we can display them on the results page.

Search application development workflow

The process of using a search platform generally involves the following steps:

1. Define the data structure, and what pieces of information you want to be searchable. In the case of Pop Culture Journal, we will ultimately want to enable searches on virtually all aspects of a film, such as its title, genre, language, and so on.
2. Make sure your data conforms to the schema you defined in step 1. For example, if you want users to search on a price, your search platform will need to know what data represents the price field. We'll see an example of that later, in Indexing the data.
3. Add your fields to your search platform. We'll see more of that in Adding fields using the LucidWorks Enterprise UI.
4. Index your data.
5. Perform user searches.

In our case, we're going to start very simple, indexing the existing content of the web site in order to provide a basic search.

Indexing content with LucidWorks Enterprise

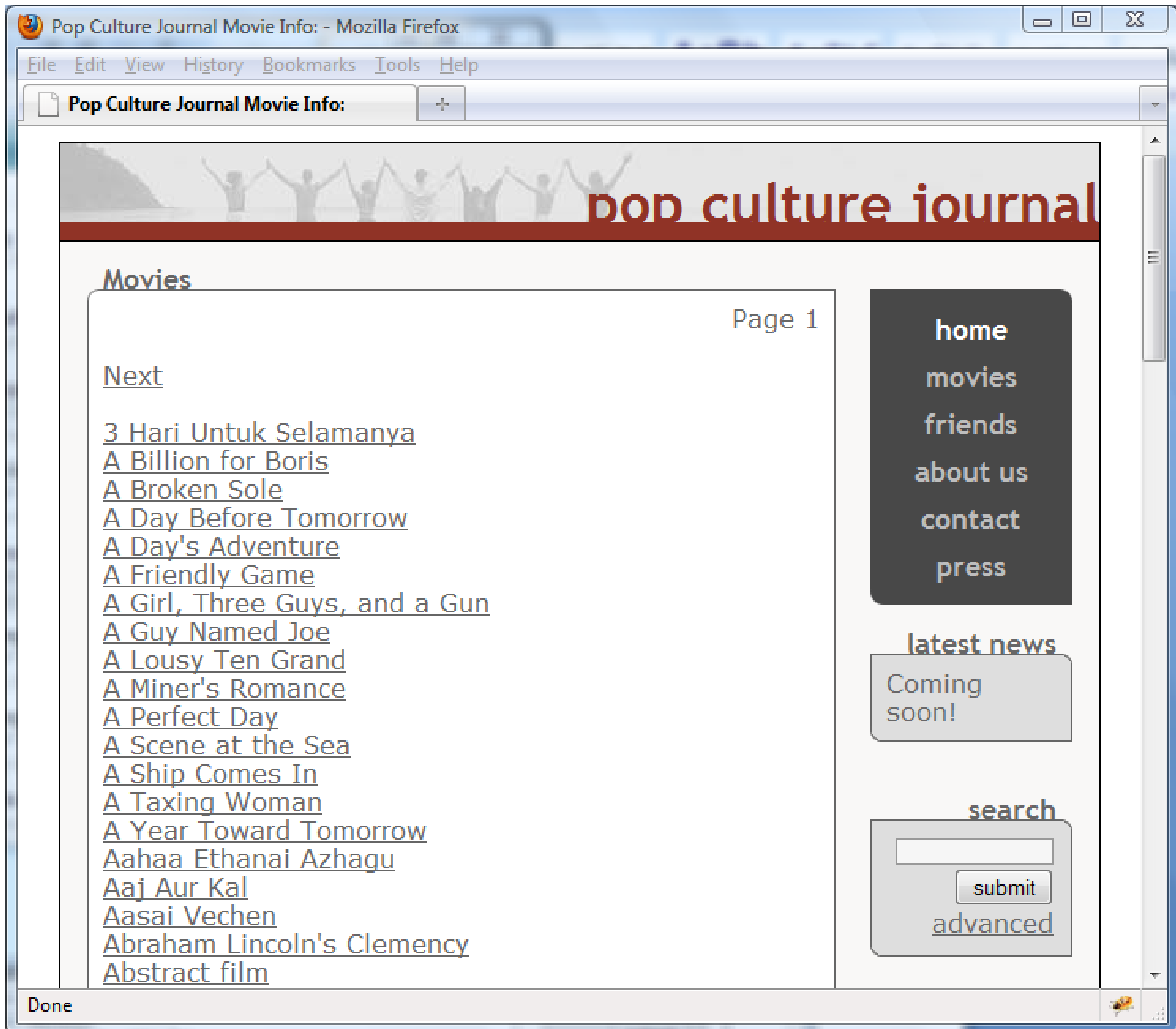
Perhaps the most basic search that you can add to your site is a simple keyword search that looks at the content of your existing web pages. That's where we're going to start with Pop

Culture Journal. We've got just under 1000 pages that we're going to index. Once we've done that, we'll create a simple page that calls LucidWorks Enterprise to perform a search and display the results.

If you've set up your web application environment as specified in Prepare the web application environment, you can see the list of movies in the static Pop Culture Journal site by pointing your browser to

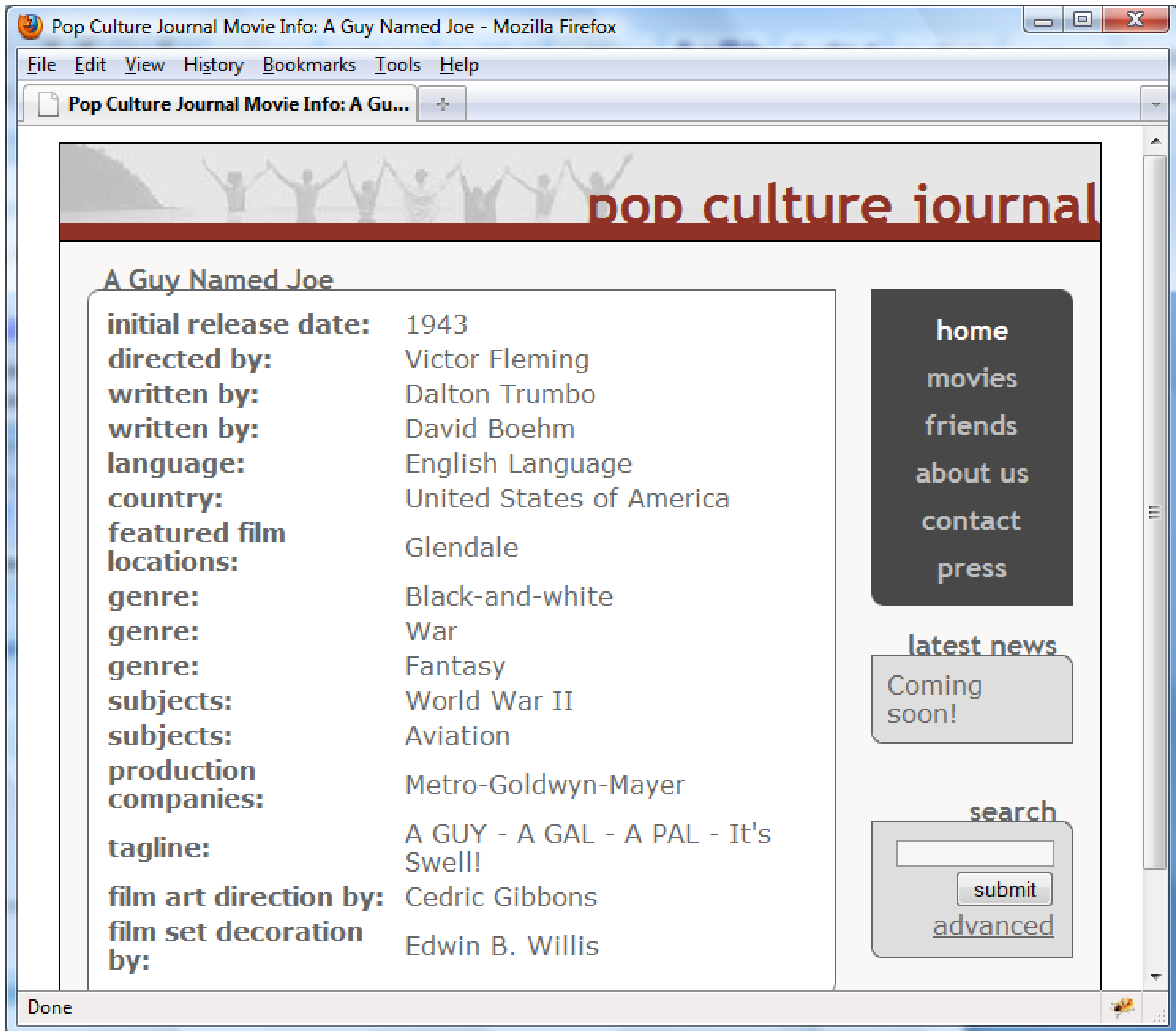
http://localhost/movies.php

You should see a list something like:



The main movies page on Pop Culture Journal

As you can see, we have an alphabetical list of movies, and if you click on one of the movies, you'll see a simple page with basic data about it, taken directly from the original data dump. Some of these pages will be fairly complete, while others won't because the data was missing in the original source. For example, clicking "A Guy Named Joe" shows a fairly complete record:

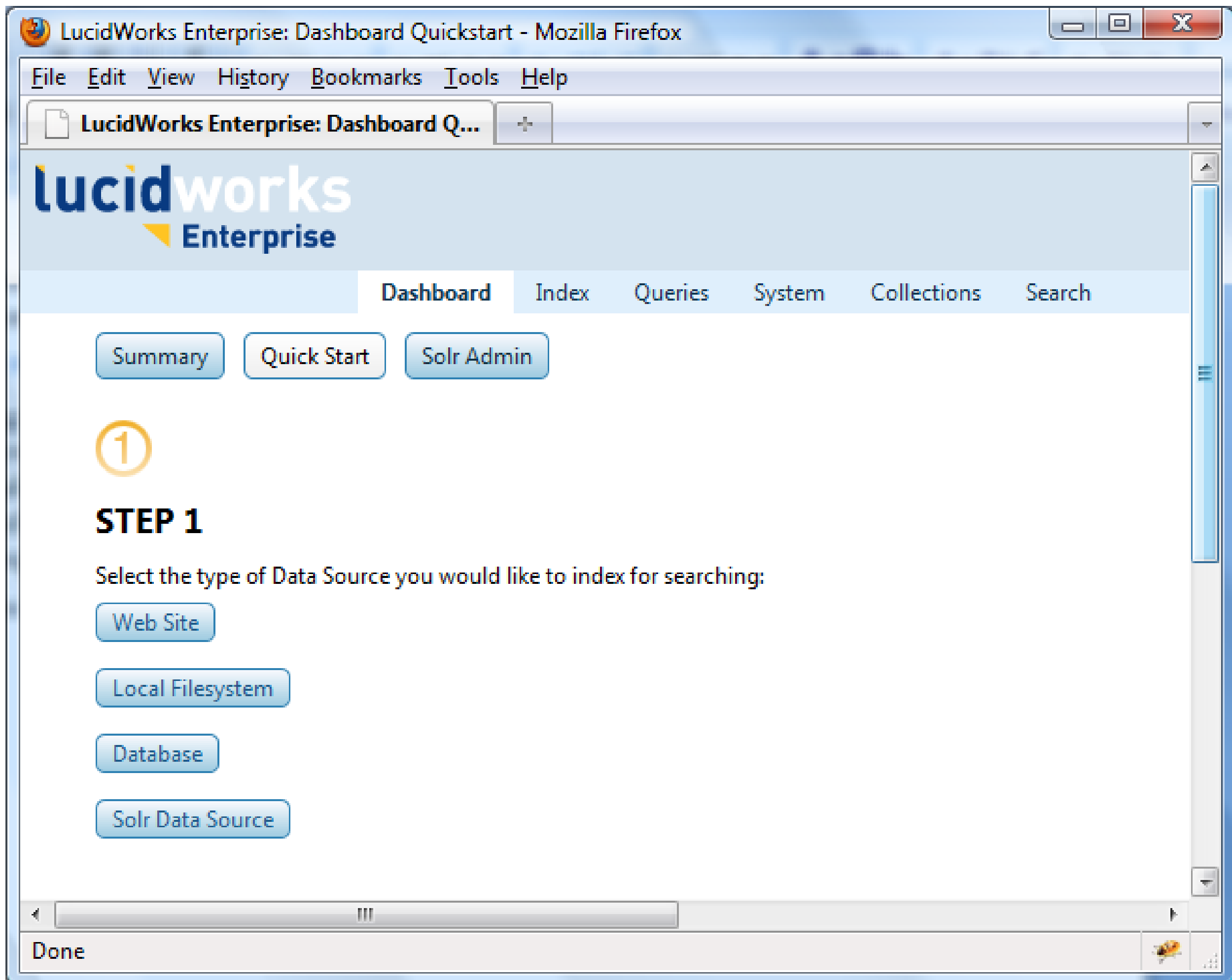


A sample movie information page

In order to index this data, you'll need to first log in to LucidWorks Enterprise. Point your browser to

http://localhost:8989/login

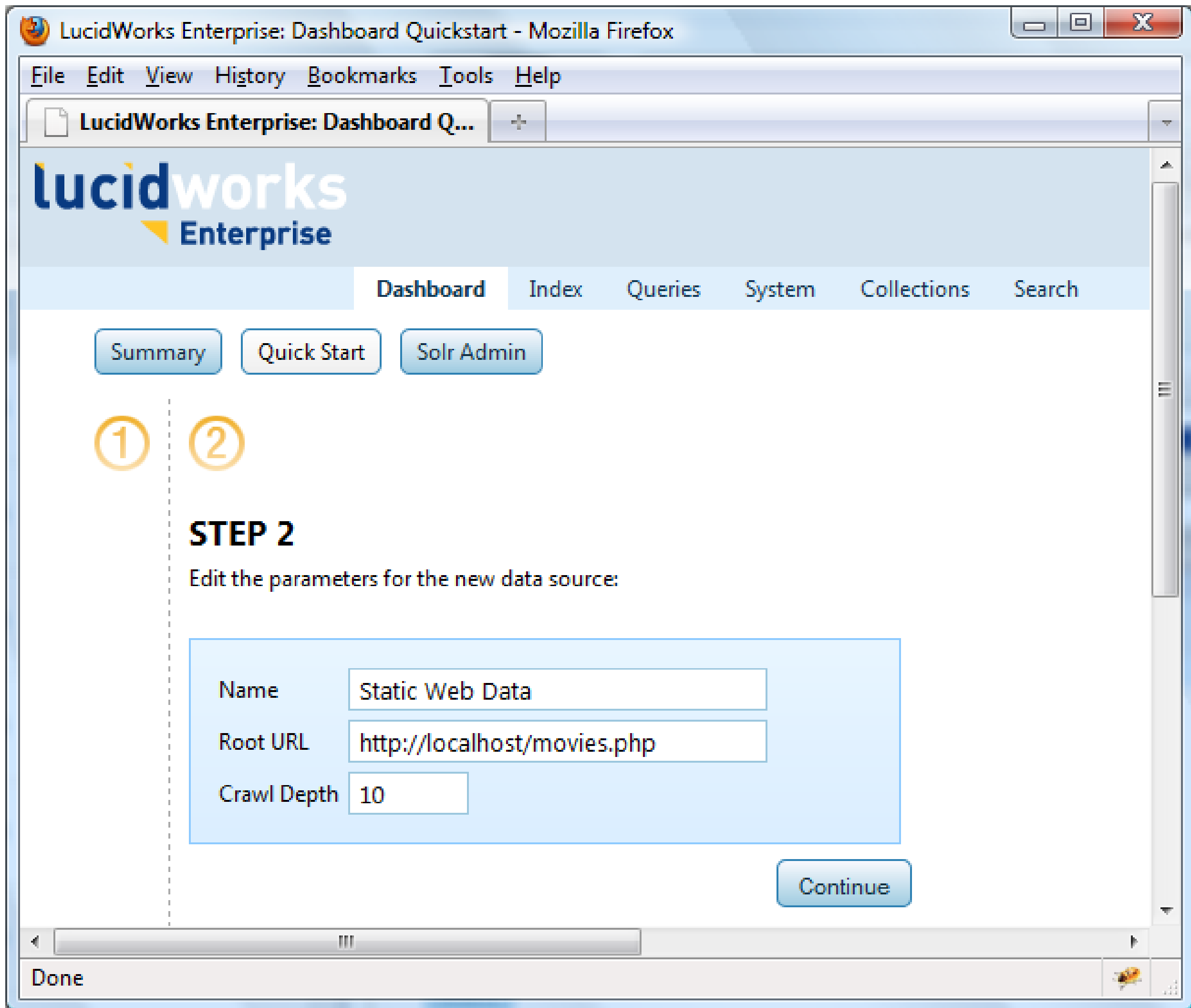
Check the installed README file for the username and password. Once you've logged in, you'll see the Quick Start page:



The LWE Quick Start page

What we're going to do here is create a data source. Data sources are useful not only because they provide you the opportunity to pull data from a variety of places, but also because they enable you to manage different types of data differently. For example, you might provide unfettered access to your web data, but include your database content in a collection that requires user authentication, or that is updated more frequently.

In this case, we're choosing to index a Web Site:



Indexing a web site

A web data source needs three pieces of information:

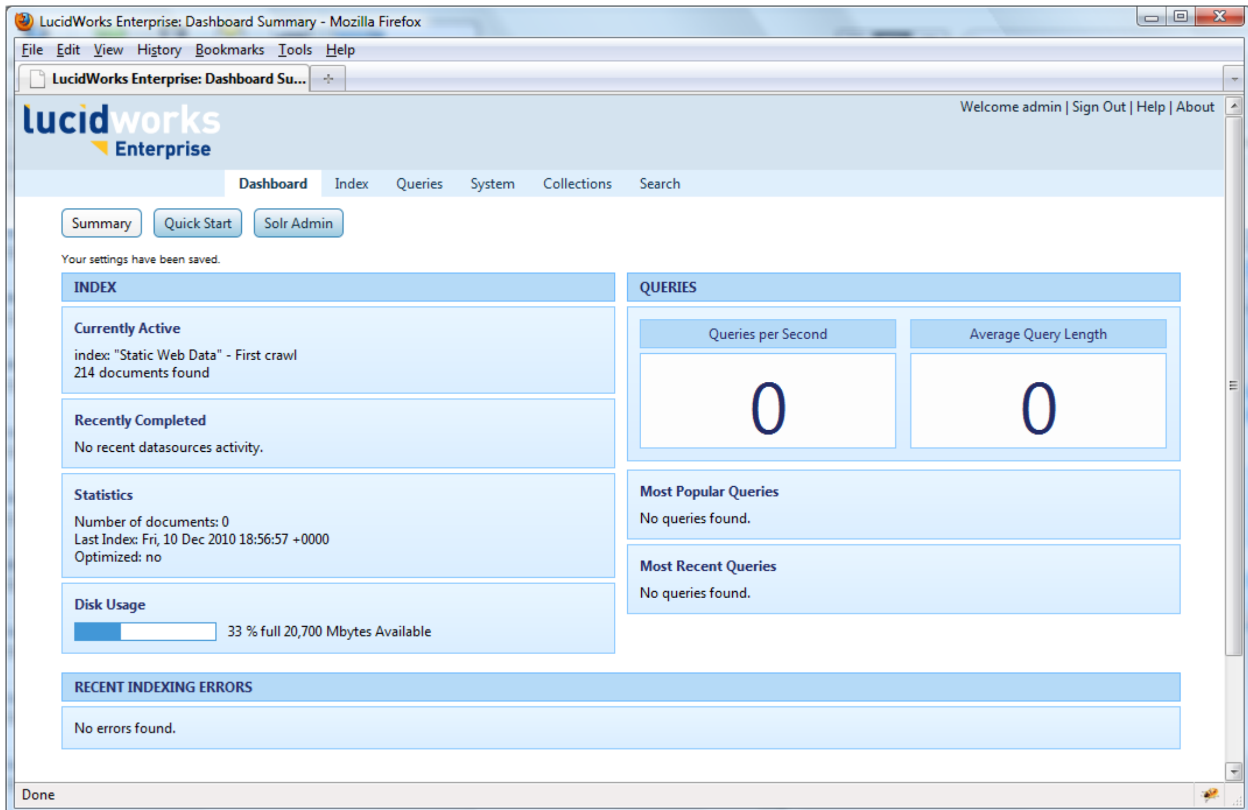
Name: The name of the data source is partly for convenience, so we can manage the data through the interface, and partly informational; you can limit a search to only data within a single data source.

Root URL: This is the page the crawler starts off on. It might be the root of your site, or as in this case, it might be a link to all of our data.

Crawl Depth: This is the link "depth" that the crawler will follow. For example, if you specify a crawl depth of 0, the indexer will look only at the specified page. For a crawl depth of 1, it would index the specified page, and any pages directly linked from that page. For a crawl depth of 2, it would index the links on those pages, and so on. In this case, we want the indexer to follow all of our "Next" links to get all of the movies, so I've specified a crawl depth of 10.

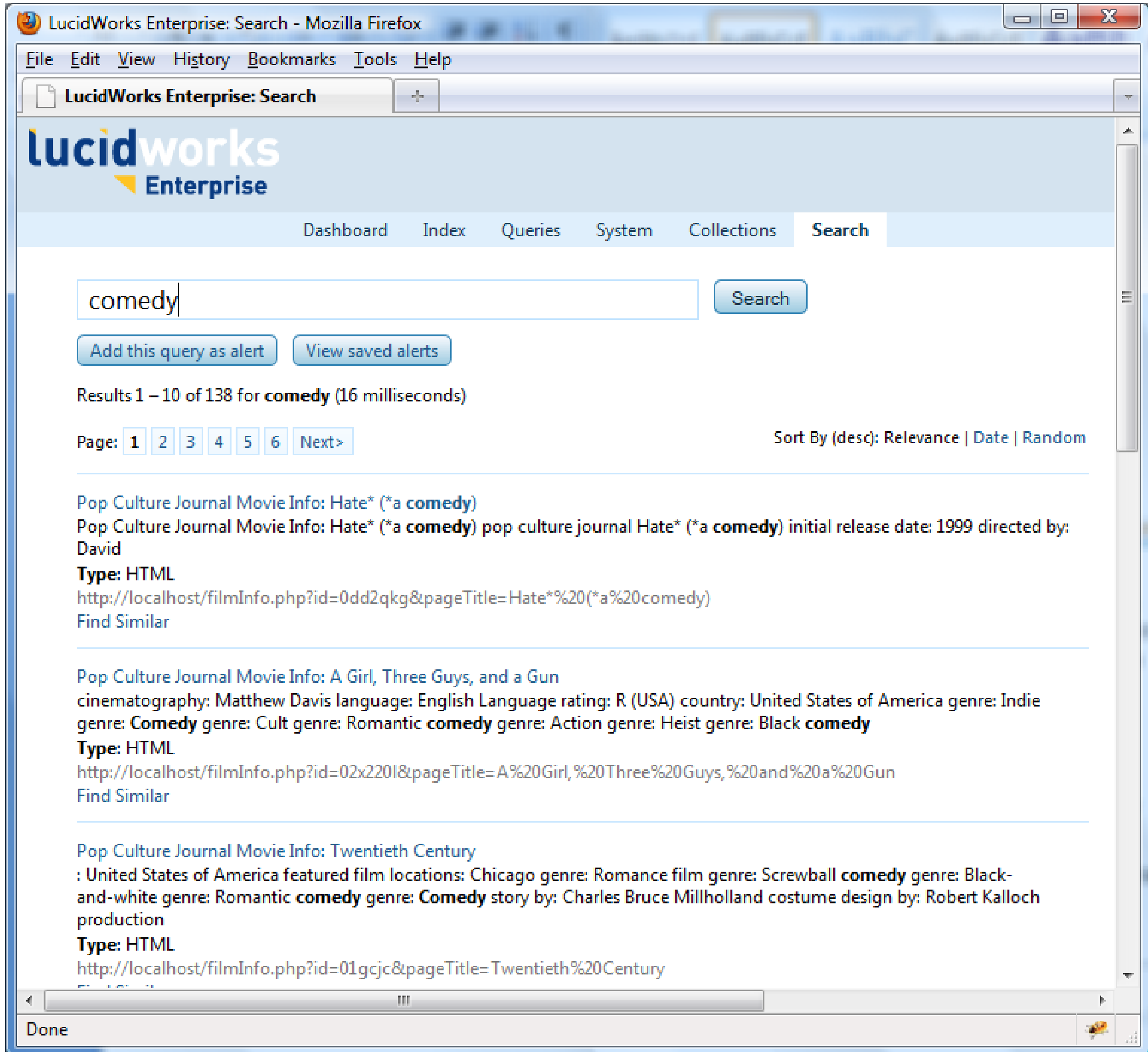
This page is a "quick start", but the full indexing page also enables you to specify URLs to allow or disallow. So you might limit your search to only your own files, or disallow searching for tweets in order to keep your index size down.

Click Continue to start LWE indexing your data. You can follow its progress on the Dashboard:



Indexing progress shows in real time on the dashboard page

Once it's done, we can do a quick check to make sure by clicking the Search tab. Over on the side, you'll see links that are called search facets, but they're not very interesting right now, so we'll get back to them later, in Faceting and other enhancements. But if you do a search for "comedy", you can see the results:



A simple search on our indexed data

Be default, the most "relevant" items appear first, but in the UI, you also have the option to sort by date, or to provide a random order. In our own application, we can sort by various fields if we choose.

So here we see a list of pages that include the term "comedy" on them. Notice that each result includes highlighted results that show the term in context. This is information that we can use in our own application. Also, each result includes the URL of the page, so we can link to it if we choose.

In the case of a web document, LucidWorks Enterprise stores meta-information such as the content-type, any links on the page, and the URL, which is stored as the document's unique identifier. (More on those later.)

LucidWorks Enterprise also stores the actual content of the page in a field called "body". When we create an advanced search page in Getting better control: advanced searching we'll look at getting finer-grained control, but for now let's go ahead and actually add a basic search.

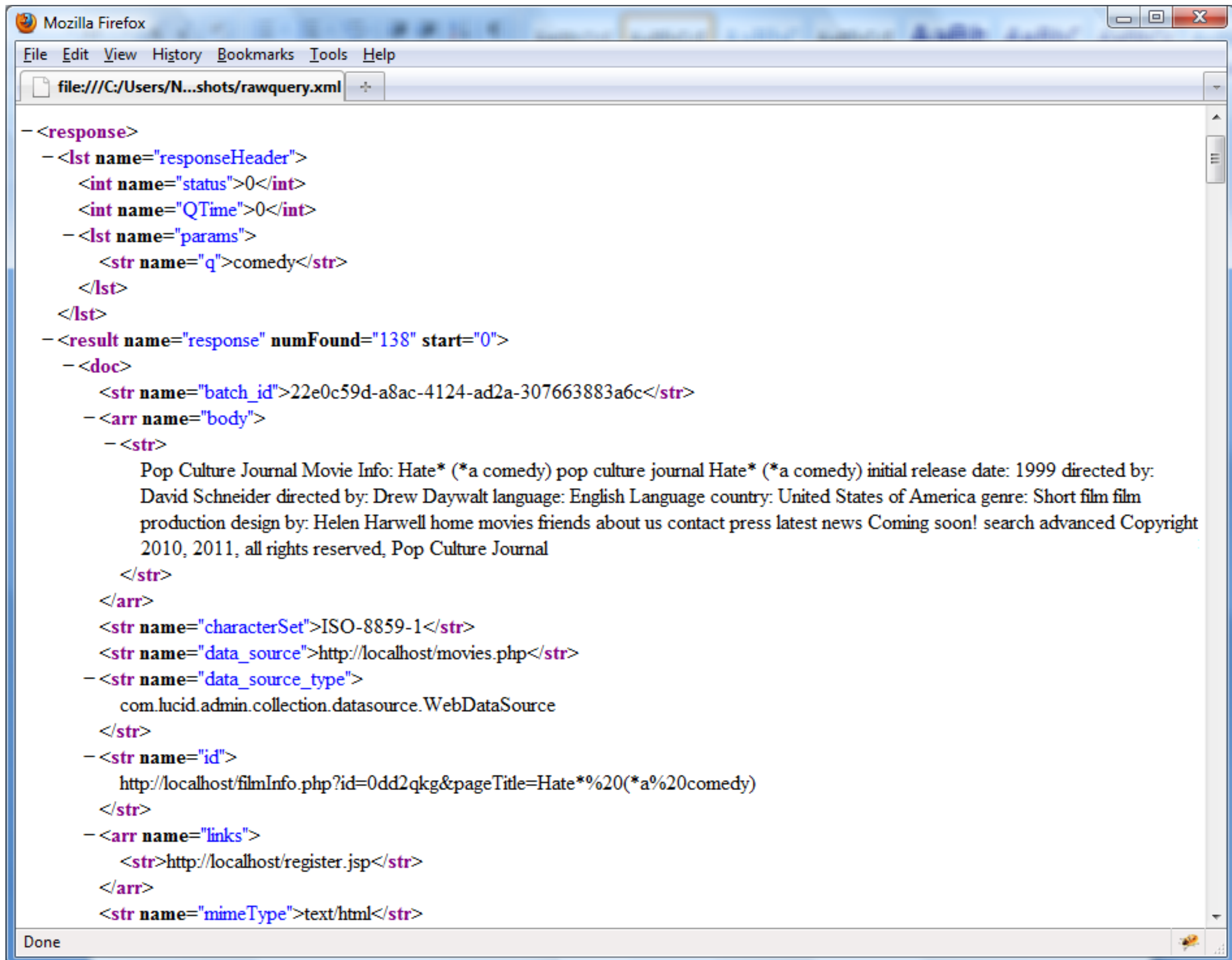
Adding search to your site

At this point we've indexed our content and shown that we can search on it, but that doesn't do us any good for adding search to the Pop Culture Journal site itself. To do that, we're going to create a page that uses PHP to send a search request and display the results.

Before we do that, though, let's look at the actual data. The important thing to understand here is that the actual mechanics behind a LucidWorks Enterprise search are pretty straightforward. If you point your browser to

http://127.0.0.1:8888/solr/collection1/select/?q=comedy

You'll see a result that looks like this:



A basic search result

Notice that this is just straight XML – LWE offers other formats, such as JSON, but XML is the default, and my personal preference – with information such as how many results it found, and then the specific data about the first ten results.

If you wanted, you could simply analyze that information directly, but we're going to simplify matters a little and use the SolrPHPClient, which was included with the PCJ download. We'll start by creating the `searchres.php` file, which is called by the search form on every PCJ page:

```
<?php
```

```
require_once( 'SolrPhpClient/Apache/Solr/Service.php' );

$query = $_REQUEST["query"];

$boxTitle = "Search Results for \"".$query."\"";
include("includes/top.php");

echo "<div id='pageTitleValue'>Search Results</div>";

$LWE = new Apache_Solr_Service(
               'localhost', '8888', '/solr/collection1' );

...
```

The basic search

The first thing we need to do is get access to the SolrPHPClient, which we can do with a simple include. (In your own application, make sure the files are accessible!) Then, after we display the top of the page, we're creating the actual service object by feeding it the server, port, and path for LWE's search function. What you see here are the default values; if you changed anything on setup, or if you're using a different collection, you'll want to adjust accordingly.

Now we can perform the actual search.

```
…
  $LWE = new Apache_Solr_Service(
               'localhost', '8888', '/solr/collection1' );

  $offset = 0;
  $limit = 100;
  $searchParams = array(
            'hl' => 'true',
            'hl.fl' => 'body'
  );

  $response = $LWE->search($query, $offset, $limit, $searchParams);

  if ($response->response->numFound > 0) {

     foreach ( $response->response->docs as $doc ) {
```

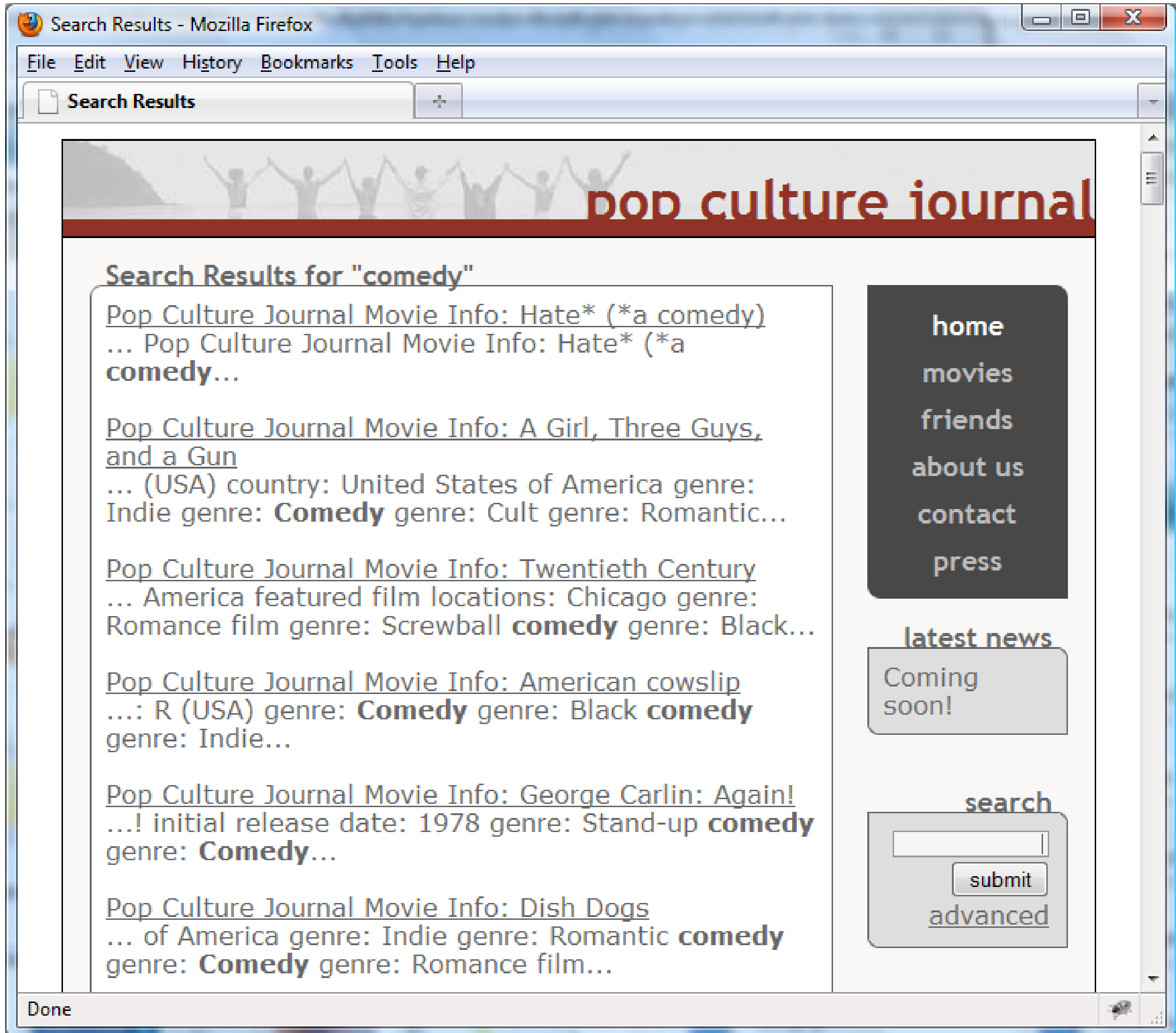
```
            echo "<a href='{$doc->id}'>{$doc->title}</a><br />";
            echo "...".
                 $response->highlighting->{$doc->id}->body[0]
                 ."...<br /><br />";
        }
        echo '<br />';
    } else {
        echo "No results for '".$query."'";
    }

    include("includes/bottom.php");

?>
```

First we set the offset and limit parameters, which determine where in the results we start, and how many results we display. In a production application, we could use that for paging. Next we add any search parameters. For example, we want highlighted results, so we'll turn that on, and specify the field we want highlighted. We can also use parameters for sorting, facets, and other purposes, as we'll see later.

Next, we're checking for any results, and if the engine's found any, we loop through them. Remember, the unit of information here is a "document", with each `$doc` object having properties that represent each attribute, such as the `title` or the `id`, which in this case is also the URL.

For each document, we're also displaying the highlighted content, which comes back in the response in the `highlighting` section. So we are requesting the highlighting for this particular document, and specifying the body field.

Now if we enter the "comedy" search term into the search box on one of the movie pages, we see a result of:



Displaying the search results on the page

As you can see, we get our list of movies, sorted by relevance, with the term highlighted in context and a link to each movie page. Not bad for just a dozen or so lines of code!

Now let's ramp things up a bit, and give our users a bit more control over what they're searching for.

Getting better control: advanced searching

Now that you've seen how easy it is to add search to your application, let's tackle something that may seem even more overwhelming: advanced search, in which users get to search on specific fields. For example, rather than just pulling any page that has the word "comedy" on it, maybe they only want films that have the word "comedy" in the title, or are in the comedy genre.

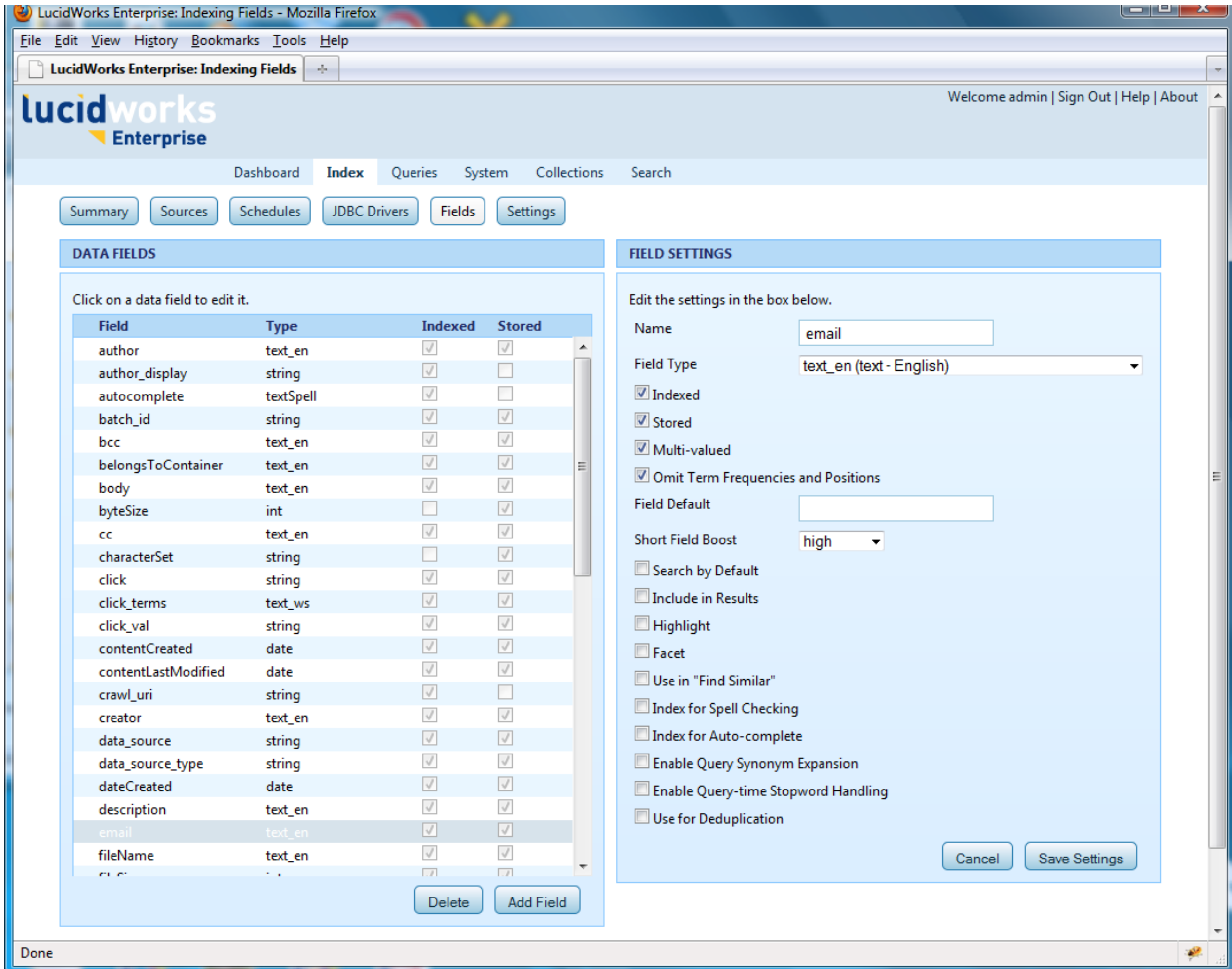
In this section, we'll look at what's involved in preparing the system for advanced queries, and at how we can implement them.

Adding fields using the LucidWorks Enterprise UI

In order for our users to be able to search on specific fields, LucidWorks Enterprise has to know what those fields are. One way that LWE makes it easy to add fields is by providing a web-based UI. To get there, make sure you're logged into LWE at

http://localhost:8989/login

and choose Index/Fields. LWE gives you a list of all of the current fields, and if you click one you can edit information about them:



LucidWorks Enterprise provides a web UI for editing fields

As you can see here, the web-based UI enables you to set properties such as the field type and whether data in the field should be `indexed` or `stored` (or both) or whether it's `multi_valued`, meaning that a single document can have more than one instance.

The UI also enables you to decide how LWE uses the data in searches. Should this field be searched by default? Should it be returned in the results? Available for highlighting? Should this data be used to provide suggestions or help check spelling?

The LucidWorks Enterprise web-based UI makes it simple to check the current parameters for a field, or even to add new fields without having to restart the search server. But if you have a large number of new fields (as we do for PCJ's film data) there's a faster way.

Adding fields using the LucidWorks Enterprise ReST API

One of the advantages of using LucidWorks Enterprise is that most of the things you can do through the UI are also available through the LucidWorks Enterprise ReST API, so they're scriptable. It's beyond the scope of this paper to describe the entire API, but basically, we're going to send a POST request to the server telling it what fields we want to add, and what values we want for important parameters that differ from their defaults.

The LucidWorks Enterprise download includes examples of how to use these functions in Perl, Python, and CSharp, but since we're working in PHP, I've put together a simple script that will use the ReST API to add our fields:

```
<?php
function addField ($fieldName, $multiValued = "false"){

  $data = array("name"=>$fieldName, "field_type"=>"text_en",
                "indexed"=>"true", "stored"=>"true",
                "search_by_default"=>"true",
                "multi_valued"=>$multiValued);

  $curl = curl_init();
  curl_setopt($curl, CURLOPT_URL,
      "http://localhost:8888/api/collections/collection1/fields");
  curl_setopt($curl, CURLOPT_POST, 1);
  curl_setopt($curl, CURLOPT_PORT , 8888);
  curl_setopt($curl, CURLOPT_HTTPHEADER,
               array("Content-Type: application/json"));
  curl_setopt($curl, CURLOPT_POSTFIELDS, json_encode($data));

  $result = curl_exec($curl);
}

addField("initial_release_date");
addField("directed_by", "true");
...
?>
```

Our field adder script

[NOTE: To keep things simple for this example, I've left out response parsing, error handling, and other important features included in the LWE examples. Make sure you follow best practices when you build your own application.]

As you can see, we're creating an array with certain basic properties for each field. For simplicity's sake, we're going to define all of our fields as English text, but in your application you'll likely want to use more specific types for dates, numeric data, and so on. We're also setting all fields as indexed, stored, and searched by default. Again, in a real production application, we'd be more discriminating about what we did with each field.

We can then create the cURL request itself, pointing it at the `fields` entry point of the Collections API. Because this is a POST request, the system knows we're trying to add a field, which it does with the data we've provided in the form of a JSON object.

To add all of the fields we'll need for the film data, point your browser to:

http://localhost/rest/addFilmFields.php

You should see a page full of messages in the form of JSON objects, and no errors. If you refresh the Fields listing in the UI, you'll see a few dozen new fields, such as `soundtrack`, `genre`, and `film_series`.

(Earlier versions of LWE have a UI issue when fields are added through the API. If you suddenly find yourself getting 500 errors after running the script to add your fields, clear your cookies and restart the browser.)

Now that we've defined our data, it's time to index it.

Indexing the data

LucidWorks Enterprise has the ability to index multiple data formats, but one way we can get complete control over how the data's interpreted is to use the Solr XML format, so our data looks something like this:

```
<doc>
    <field name='id'>/m/02z99_p</field>
    <field name='objectType'>film</field>
    <field name='name'>Olive Oyl for President</field>
    <field name='directed_by'>Isadore Sparber</field>
    <field name='music'>Winston Sharples</field>
```

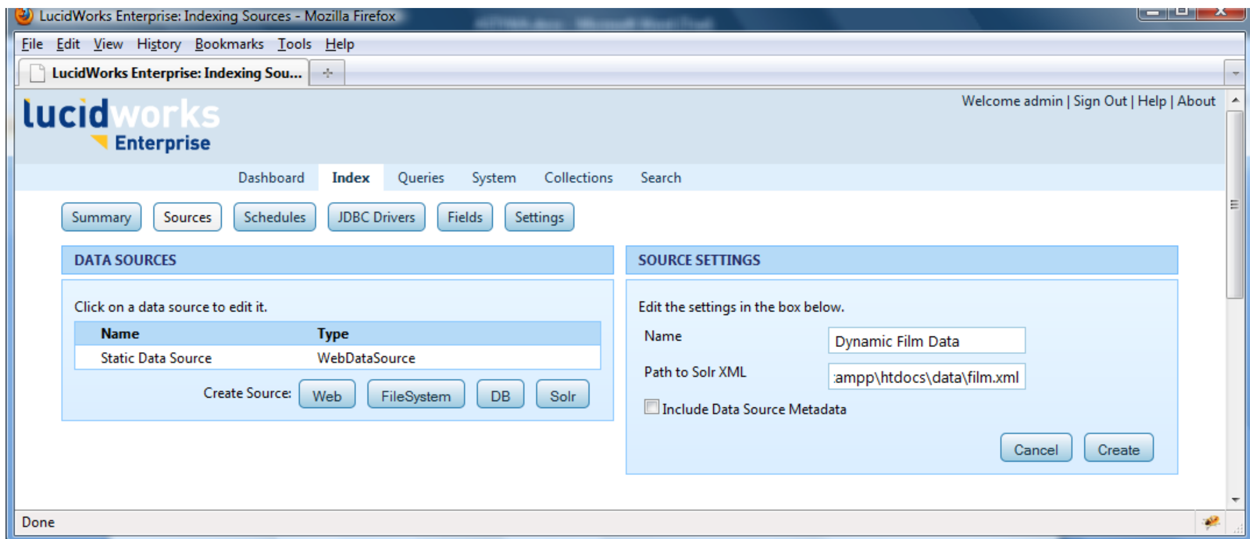
```
        <field name='language'>English Language</field>
        <field name='starring'>/m/052d489</field>
        <field name='starring'>/m/052d484</field>
        <field name='genre'>Comedy</field>
    </doc>
```

A sample document

Each `doc` element is a collection of `field` elements, each of which has a `name` attribute and data. By default, the `id` field is the unique identifier for the document; if you update this document and provide the same `id` value, LWE will update the original document rather than adding a new one.

Notice that some fields, such as `starring`, contain `id` values rather than data; these fields refer back to other documents. For now, though, we've included just film data -- we'll build out the rest in other projects – in the film.xml file in the PCJ data directory.

So to add our film data, we need to index the `film.xml` file. To do that using the UI, make sure you're logged in and click Index, then Sources.

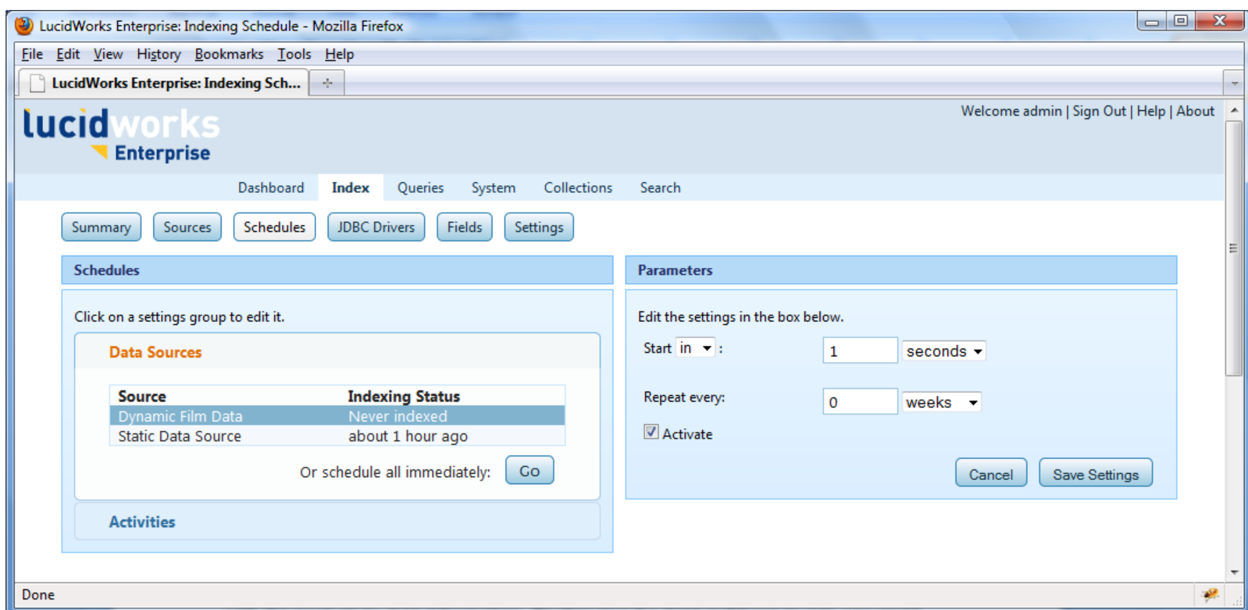


Adding a new data source

What we want to do here is to add a new Solr data source, so click the Solr button. As you can see, we just need the name of the data source (Dynamic Film Data) and the path to the `film.xml` file, which is located in the `data` directory of the PCJ download. (If you have a

collection of files, you can provide just the directory, and LWE will index all the files in it.) Click Create to save the new data source.

Now we've created the source, but nothing will actually happen until we tell LucidWorks Enterprise to index its data. To do that, click the Schedules button, then Data Sources. Click the new data source to get the option to schedule it.



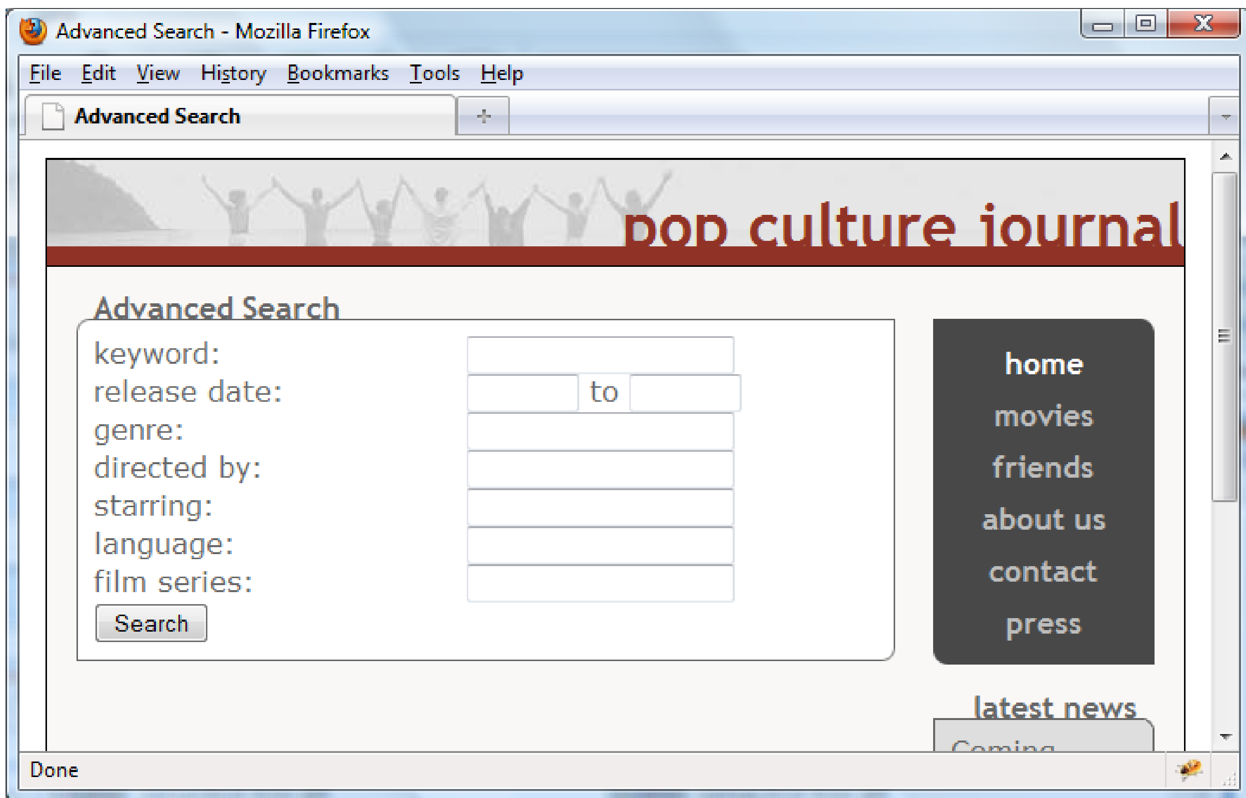
Running the new data source

Because we want the data to be indexed right now, we can tell LWE to index it in one second. We can also schedule any data source to be reindexed on a regular basis. For example, if this data changed frequently, we might want to index it every day, or even every hour or so, so that the index was always up to data, and users would always be working with the most up-to-date information. By changing the Start pulldown to "at", you could even schedule the source to be indexed at a particular data and time.

In our case, we just want to run it once, and right now, so choose to index in one second and click Save Settings. Just as it did before, the system goes off and indexes all of the documents, and we can keep tabs on it from the Dashboard until it's finished. In the meantime, let's look at building the advanced search.

Performing the search

We'll start by creating the advanced search page. In this case, we're looking at straight HTML code, which you can find by clicking the "advanced" link in the search box. Here's what it looks like:



The advanced search page enables users to choose a field on which to search

As you can see, users get to decide what fields to search for each piece of information. So when users enter criteria for a specific field, we'll need to take that into account when we build the query, in `advancedres.php`:

```
<?php
require_once( 'SolrPhpClient/Apache/Solr/Service.php' );

$searchQuery = $_REQUEST["query"];
$advQuery = array();
```

```
$date_from = "*";
$date_to = "*";
if ($_REQUEST["date_from"] != ""){
     $date_from = $_REQUEST["date_from"];
}
if ($_REQUEST["date_to"] != ""){
     $date_to = $_REQUEST["date_to"];
}
if ($date_from != "*" OR $date_to != "*"){
     array_push($advQuery, "initial_release_date:[".$date_from." TO
".$date_to."]");
}
if ($_REQUEST["genre"] != ""){
     array_push($advQuery, "genre:".$_REQUEST["genre"]);
}
if ($_REQUEST["directed_by"] != ""){
     array_push($advQuery, "directed_by:".$_REQUEST["directed_by"]);
}
if ($_REQUEST["starring"] != ""){
     array_push($advQuery, "starring:".$_REQUEST["starring"]);
}
if ($_REQUEST["language"] != ""){
     array_push($advQuery, "language:".$_REQUEST["language"]);
}
if ($_REQUEST["series"] != ""){
     array_push($advQuery, "series:".$_REQUEST["series"]);
}
array_push($advQuery, "objectType:film");
array_push($advQuery, "tagline:[* TO *]");

for ($i = 0; $i < count($advQuery); $i++){
     if ($searchQuery != "") { $searchQuery .= " AND "; }
     $searchQuery .= $advQuery[$i];
}

$boxTitle = "Search Results for \"".$searchQuery."\"";
include("includes/top.php");
echo "<div id='pageTitleValue'>Search Results</div>";
```

…

Building the query

Now, that seems like a lot of code, but all we're really doing at the start of this page is determining what fields the user was specifically searching on, and building them into a single query. To search for information in a specific field, we use the colon notation, so looking for English language films would include:

```
language:English
```

We can also request a range, as you can see from the initial_release_date, such as

```
initial_release_date:[1951 TO 2000]
```

Finally, we're specifically limiting ourselves to film documents, and giving preference to films that have more complete information by requiring a tagline. Note that "*" can't be the first character of a query, so to request any tagline, we're using the range:

```
tagline:[* TO *]
```

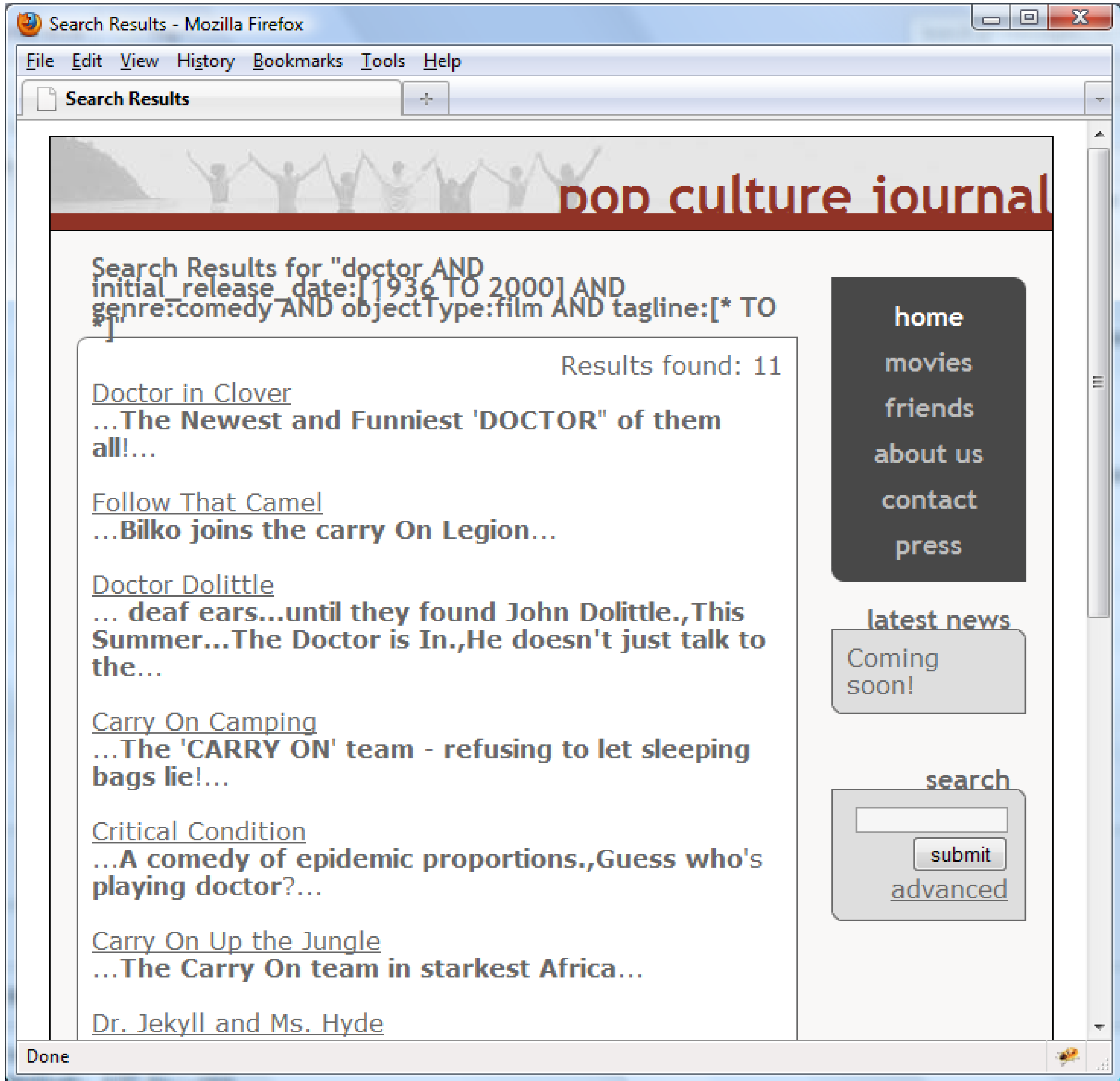
Now we execute the search, as before:

```
...
$LWE = new Apache_Solr_Service( 'localhost', '8888',
                                '/solr/collection1' );

$offset = 0;
$limit = 100;
$searchParams = array(
              'hl' => 'true',
              'hl.fl' => 'tagline',
);

$response = $LWE->search( $searchQuery, $offset, $limit,
                          $searchParams );

if ( $response->response->numFound > 0 ) {

    echo "<div id='resultsFound'>Results found: ".
                $response->response->numFound."</div>";

    foreach ( $response->response->docs as $doc ) {
        echo "<a href='film.php?id={$doc->id}'>".
                  $doc->name."</a><br />";
        echo "...".
             $response->highlighting->{$doc->id}->tagline[0].
             "...<br /><br />";
```

```
    }
    echo '<br />';
} else {
    echo "No results for '".$query."'";
}

include("includes/bottom.php");
?>
```

Executing the search

The basics are the same as in the original search. So if I do a query for films from 1936 to 2000, in the comedy genre, using the keyword “doctor”, I get 11 results:



The search results

In a real production application I'd clean up the query display and provide paging, but you get the basic idea.

Notice that we've changed the URL we're linking to slightly. Rather than linking to our static pages, we're going to link to a dynamic page that pulls the film data directly from the index using only the id passed in with the link:

```
<?php

require_once( 'SolrPhpClient/Apache/Solr/Service.php' );
$filmId = $_REQUEST["id"];
$searchQuery = "id:".$filmId;

$LWE = new Apache_Solr_Service( 'localhost', '8888',
                                '/solr/collection1' );

$offset = 0;
$limit = 1;

$searchParams = array();
$response = $LWE->search( $searchQuery, $offset, $limit,
                          $searchParams );

if ( $response->response->numFound > 0 ) {

    $film = $response->response->docs[0];
    $boxTitle = $film->name;
    include("includes/top.php");

    echo "<div id='pageTitleValue'>".$film->name."</div>";
    echo "<div class='filmtitle'>{$film->name}</div>";
    echo "<div class='taglineDynamic'>{$film->tagline}</div>";

    if ($film->initial_release_date != ""){
        echo "<div class='initial_release_date'>";
        echo "<div class='filmLabel'>Originally released:</div>";
        echo "<div>{$film->initial_release_date}</div>";
        echo "</div>";
    }
    if ($film->directed_by != ""){
        echo "<div class='directed_by'>";
        echo "<div class='filmLabel'>Directed By:</div>";
        echo "<div>".getArrayItems($film->directed_by)."</div>";
        echo "</div>";
```

```
        }
        if ($film->written_by != ""){
            echo "<div class='written_by'>";
            echo "<div class='filmLabel'>Written by:</div>";
            echo "<div>".getArrayItems($film->written_by)."</div>";
            echo "</div>";
        }
    } else {
        echo "No such film.";
    }

    include("includes/bottom.php");

    function getArrayItems( $objectArray){

      $returnStr = "";
      if (is_array($objectArray)){
        foreach ($objectArray as $thisObject) {
          if ($returnStr != ""){$returnStr .= ", ";}
          $returnStr .= $thisObject;
        }
      } else {
        $returnStr = $objectArray;
      }
      return $returnStr;
    }
?>
```

Pulling film information directly from the index

Most of this should seem pretty familiar by now; we create the server, create the query, and display the results. Again, to keep things simple I've just displayed a few fields. Multi-valued fields come in as arrays, so at the bottom we have a function to display them nicely.

A film information page has the basic data

Now, this looks simple, but don't underestimate the power of it. We just created an entire page based on information returned by the search engine. That means that any time we update the index, the site gets updated.

"Big deal," you might be thinking. "I get the same thing with my database."

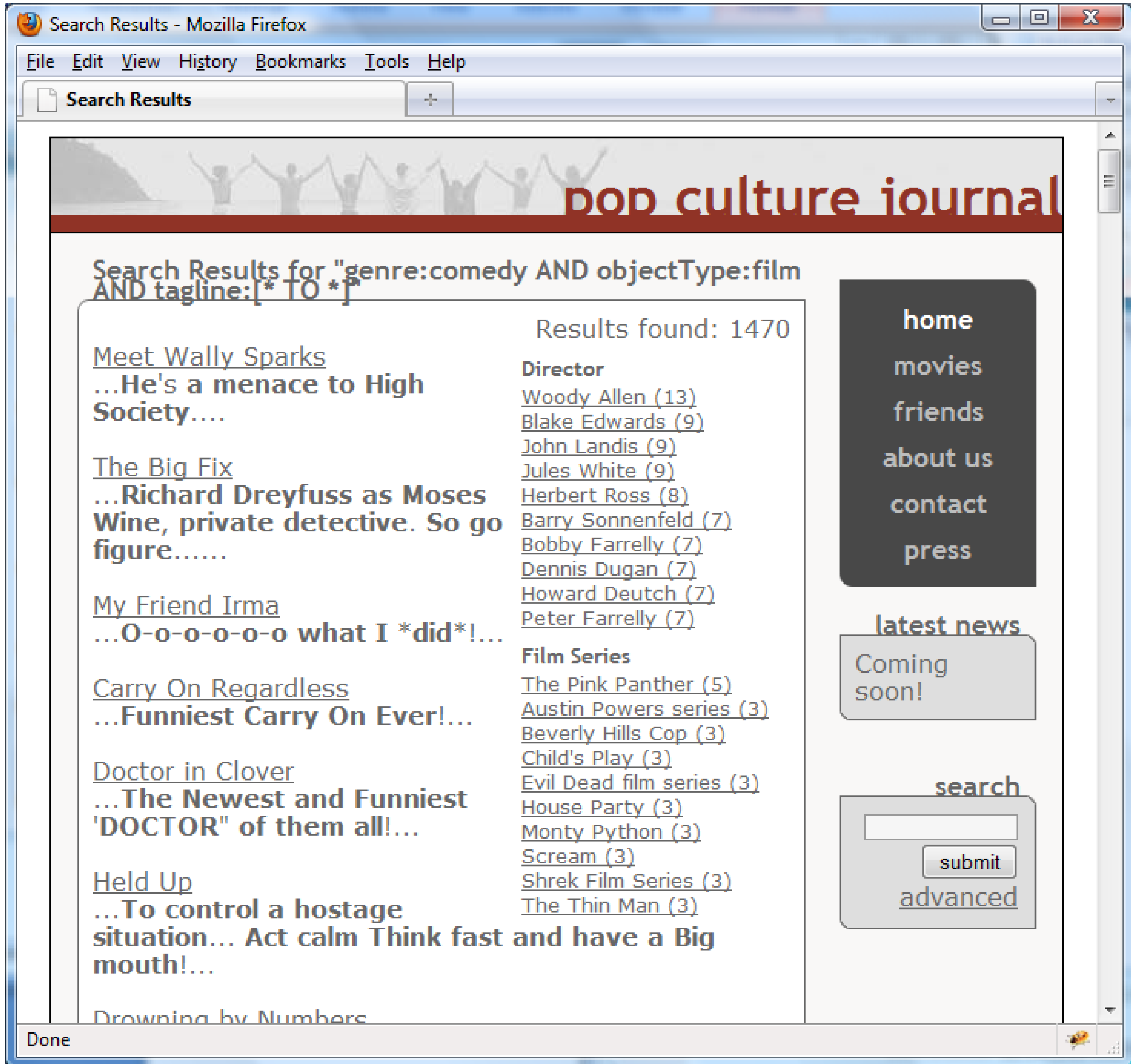
And that's true. But what if you built a page that included sub-queries? For example, film data includes IDs that link back to performances and other data we haven't yet loaded. What if you had an actor's page that listed some of his most prominent movies?

When I say the site is updated with the search index, I don't just mean the data. LucidWorks Enterprise includes features that gradually improve search results over time. So if, for example, we used LWE's click scoring API to track what movies were clicked when the user searched for an actor, we'd know which movies he was best known for, and we could display them first. So without doing any extra programming, your content gets better and more relevant as your search engine does.

With the advanced search in place, let's take a quick look at one more handy feature for adding search to your web application.

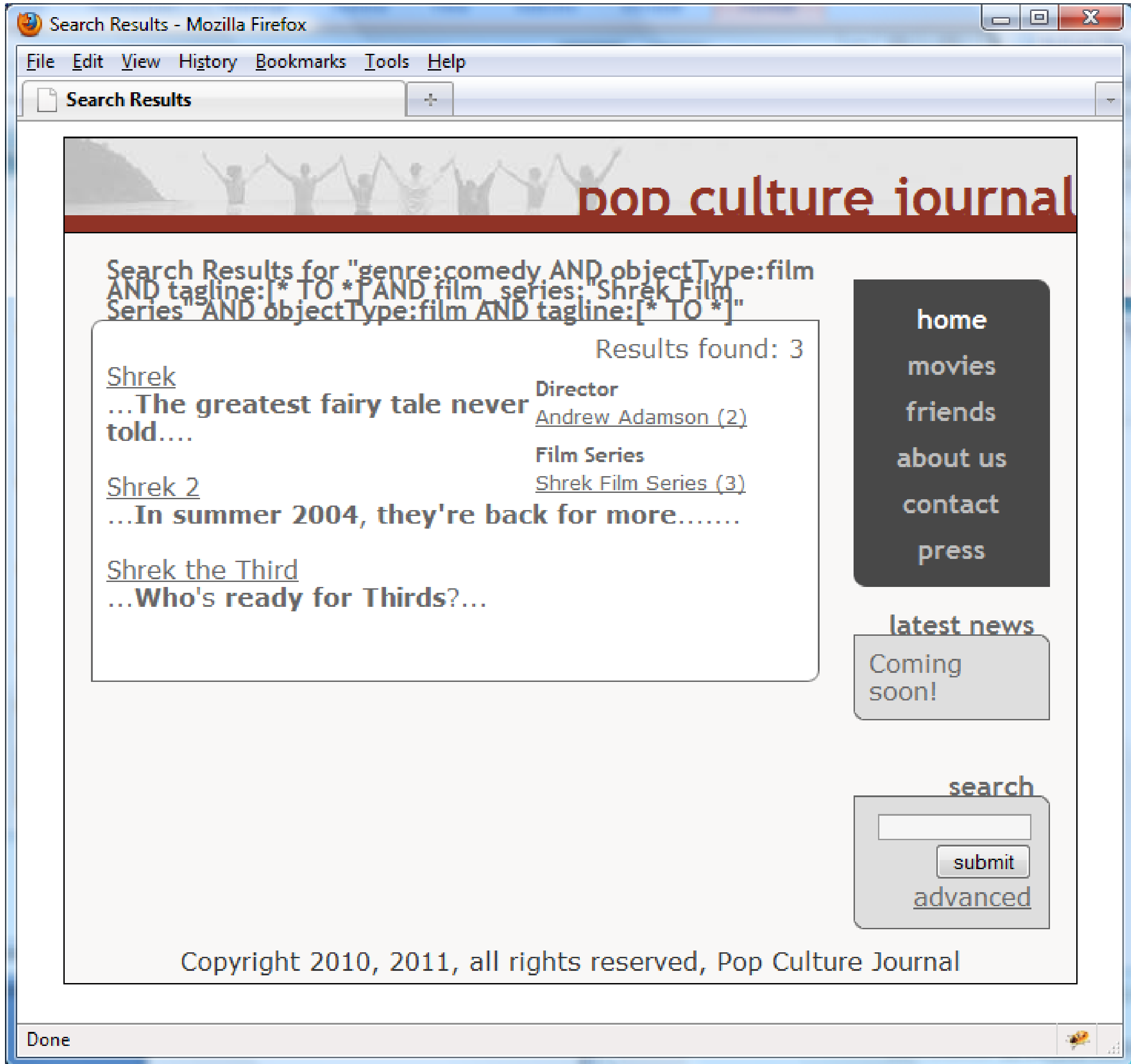
Faceting and other enhancements

Before we close there are a few very handy features of LucidWorks Enterprise you should know about when planning search functionality for your own application. One of the most exciting is faceted search. For example, if I enable faceting for our search – I'll show you how in a moment – and just do a search on "comedy", I get the ability to narrow down my search by director or film series:



Facets enable our users to narrow their searches

If I then click on the Shrek series, I can narrow things even more:



Facets provide progressively more narrow information

Normally, this would take some pretty heavy lifting, but faceting is one of the built-in features of the engine at the heart of LucidWorks Enterprise. To use it, we simply add the appropriate parameters to the search in `advancedres.php`:

```
...
$limit = 100;

    $searchParams = array(
              'hl' => 'true',
              'hl.fl' => 'tagline',
              'facet' => 'true',
              'facet.field' => array('film_series','directed_by'),
              'facet.mincount' => 2,
              'facet.limit' => 10
    );

  $response = $LWE->search( $searchQuery, $offset, $limit,
                            $searchParams );

  if ( $response->response->numFound > 0 ) {

       echo "<div id='resultsFound'>Results found: ".
                      $response->response->numFound."</div>";

       include("facets.php");

  foreach ( $response->response->docs as $doc )
…
```

Turning on facets

As you can see, we've simply added four parameters to the search. First we turn faceting on, then we specify the fields for which we want facets. Next we specify a minimum number of items before a facet is displayed, so if a director only directed one film in the results, his name won't be listed. Finally, we specify the maximum number of items to return.

The actual facets are displayed by the code in `facets.php`:

```
<?php
if (count($response->facet_counts->facet_fields->directed_by) > 0){
    $directorString = "";
    $directorString .= "<div class='facets'><h3>Director</h3>";
    $display = false;
    foreach ($response->facet_counts->facet_fields->directed_by
                  as $key => $val){
```

```
            $directorString .=  "<a href='advancedres.php?".
                                  "query=".$searchQuery.
                                  "+AND+directed_by:\"".$key."\"'>";
            $directorString .=  $key." (".$val.")";
            $directorString .=  "</a><br />";
            $display = true;
        }
        $directorString .=  "</div>";
        if ($display){ echo $directorString; };
    }
    if (count($response->facet_counts->facet_fields->film_series) > 0){
        $seriesString = "";
        $seriesString .= "<div class='facets'><h3>Film Series</h3>";
        $display = false;
        foreach ($response->facet_counts->facet_fields->film_series
                                 as $key => $val){
            $seriesString .= "<a href='advancedres.php?".
                                  "query=".$searchQuery.
                                  "+AND+film_series:\"".$key."\"'>";
            $seriesString .= $key." (".$val.")";
            $seriesString .= "</a><br />";
            $display = true;
        }
        $seriesString .= "</div>";
        if ($display) { echo $seriesString; }
    }
    ?>
```

Displaying facets

I won't spend a lot of time on this – the examples are in the PCJ code, so feel free to play with them – but the basic idea is that we're getting the facet counts from the response just as we retrieved the highlighted content, then displaying it. For each link, we're incorporating the current query and adding an additional query to narrow the results by the given link.

LucidWorks Enterprise also provides other features that you can easily provide your users. They include:

- Autocomplete, which provides a dropdown of indexed terms based on what the user's already typed into the search box

- Spell-check, which detects words that are not found in the index and provides suggestions to help users who might not be quite sure what they're looking for
- User alerts, which enable users to request notification when new content is added for their favorite searches
- Click scoring, which tracks which links are clicked for a query and automatically tunes the search engine so that those results show greater relevance

These functions are available to you with only a tiny fraction of the work that would have been required had you decided to build them yourself.

Summary

In this paper, we've looked at what's required to add search to your web application using LucidWorks Enterprise. We looked at LWE's built-in web crawler for static content, and saw that just a few lines of code can add sophisticated search results to your app.

We also looked at indexing more structured data in order to be able to provide "advanced" search functions, enabling users to search based on specific fields. In that context, we discussed how building our content right out of the search index, as we did here, enables you to automatically leverage relevance improvements within your actual content.

Finally, we looked at some of the more convenient features of LucidWorks Enterprise, such as faceted searching, autocomplete, and click scoring.

All of these features are designed to lower the barrier to entry for search in your web application, making it possible for almost any programmer to provide sophisticated results. And when you're not bogged down in the details of search and its bells and whistles, you can concentrate on building your own new and innovative applications.

Next Steps

For more information on how Lucid Imagination can help search application developers, employees, customers, and partners find the information they need, please visit www.lucidimagination.com to access blog posts, articles, and reviews of dozens of successful implementations.

LucidWorks Enterprise is the search solution development platform built on the power of Apache Solr/Lucene, developed by the enterprise search experts at Lucid Imagination. LucidWorks Enterprise leverages the disruptive innovation of the leading open source search technology, to deliver unmatched scalability to billions of documents, with subsecond query and faceting response time.

By building and extending the scalable power of Solr open source with vital new features, the search experts at Lucid Imagination have created an integrated platform that simplifies and empowers predictable, reliable search application development.

With LucidWorks Enterprise, you get the search results that matter – with a dynamic, tight between fit your business goals, processes, and the valuable information your customers and users need – today and as your search and business grow.

Or call: 1.650.353.4057