

LucidWorks Search Custom Connector Guide

2.8 Documentation

Table of Contents

How to Use this Documentation	5
Audience and Scope	5
Conventions	5
Customers of LucidWorks Search on AWS or Azure	7
Getting Support & Training	8
Custom Connector Guide	9
Example Crawler	9
Introduction to Lucid Connector Framework	10
How To Create A Connector	12
Integrating Google Connectors	26
Integrating New Crawlers with LucidWorks	34
About LucidWorks	36

LucidWorks Search Documentation

The LucidWorks Search Documentation is organized into several guides that cover all aspects of using and implementing a search application with LucidWorks Search, whether on-premise or hosted on AWS or Azure.

Installation & Upgrade Guide

- Installing LucidWorks Search
- System Directories and Logs
- Upgrade instructions for v2.8
- Review changes from LucidWorks v2.7 to v2.8

System Configuration Guide

- Troubleshooting crawl issues
- Alerts configuration
- Query options
- Custom fields, field types, and other index customizations
- Performance considerations and system monitoring
- Distributed search and indexing
- Security options

Lucid Query Parser

- How the default query parser handles user requests
- Customization options

LucidWorks REST API Reference

- Configure data sources and administer crawls
- Set system settings
- Manage fields, field types, and collections
- Example clients in C#, Perl and Python

Custom Connector Guide

- [Introduction to Lucid Connector Framework \(see page 10\)](#)
- [How To Create A Connector \(see page 12\)](#)

How to Use this Documentation

Audience and Scope

This guide is intended for search application developers and administrators who want to use LucidWorks Search to create world class search applications for their websites.

While LucidWorks Search is built on Solr, and many of its features are implementations of Solr and Lucene features, this Guide does not cover basic Solr or Lucene configuration. We do, however, point out where LucidWorks Search deviates from Solr or Lucene standard configuration practices, and have provided links to Solr and Lucene documentation where possible for further explanation if the functionality in LucidWorks Search is identical to Solr or Lucene.

One important note to remember is that LucidWorks is multi-core enabled by default, with `collection1` as the default core. This means that standard Solr paths such as `http://localhost:port/solr/*`, as shown in Solr documentation, would be `http://localhost:port/solr/collection1/*` in LucidWorks Search.

Topics covered on this page:

- [Audience and Scope \(see page 5\)](#)
- [Conventions \(see page 5\)](#)
- [Customers of LucidWorks Search on AWS or Azure \(see page 7\)](#)
- [Getting Support & Training \(see page 8\)](#)

Conventions






Paths

Server paths are described in relation to the base LucidWorks Search installation path, indicated by `$LWS_HOME`. For example, if LucidWorks Search was installed at `/var/lucidworks`, then the path to the 'app' directory shown as `$LWS_HOME/app` will be `/var/lucidworks/app` on the server.

Notes

Special notes are included throughout these pages.

Note Type	Look & Description
-----------	--------------------

Note Type	Look & Description
Information	 Notes with a blue background are used for information that is important for you to know.
Notes	 Notes are further clarifications of important points to keep in mind while using LucidWorks.
Tip	 Notes with a green background are Helpful Tips.
Warning	 Notes with a red background are warning messages.
Cloud	 Information for LucidWorks Search in the Cloud Users Information specifically for LucidWorks Search customers on the AWS or Azure Platform.

REST API Conventions

Many of the LucidWorks Search REST APIs support several methods (such as POST, GET, PUT, DELETE) and each is documented with detailed attribute descriptions and examples of inputs and outputs. Each description includes the path to the API endpoint, parameters for input, and the attributes returned as a result of the request.

Windows users should take care when copying the examples as they assume that you are familiar with how to modify unix-based curl commands for the Windows environment.

Parameters

Several of the paths shown in the API documentation include parameters that need to be modified for your installation and specific configuration. These are indicated in *italics*.

For example, getting the details of a data source is shown as:

```
GET /api/collection/collection/datasources/id.
```

If you were using 'collection1' and data source '3', you would enter:

```
GET /api/collection/collection1/datasources/3.
```

Server Addresses

The LucidWorks Search REST API uses the Core component, installed at <http://localhost:8888/> by default in LucidWorks Search. Many examples in this Guide use this as the server location. If you have installed LucidWorks Search locally, and you changed this location on install, be sure to change the destination of your API requests accordingly.

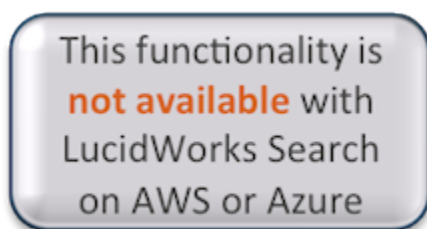
Customers hosted on AWS or Azure should see the section for [#Customers of LucidWorks Search on AWS or Azure \(see page 7\)](#) below.

Customers of LucidWorks Search on AWS or Azure

All of the preceding information on this page applies to customers who have LucidWorks Search hosted on either AWS or Azure Platforms, with a few small exception which are detailed below.

Configuration Options

Certain configuration options are available with on-premise installations only (such as installation options, manual configuration file changes, etc.). The following panel will appear on any page or section that does not apply or is not available for LucidWorks Search on the AWS or Azure platforms:



API Conventions for LucidWorks Search on AWS or Azure

Nearly all of the documented REST APIs will work for customers on AWS or Azure, but the example API calls must be modified to include either the Access Key or the API Key and used as authentication credentials. Customers are being transitioned from a simple Access Key to a more secure Basic authentication system that requires a unique API Key.

1. Customers who only have an Access Key can see the key on the My Search Server page and the main Collections Overview page of your instance (click the REST API button above the usage graphs). Example URLs for API calls used in this documentation would then be changed from `http://localhost:8989/api/...` to `http://access.lucidworks.io/<access key>/api/...`. This access key is specific to your instance and should be treated as securely as possible to prevent unauthorized access via the APIs to your system.

2. Customers with Basic authentication have instances which use an URL with "`https://s-XXXXXXXX.lucidworks.io`" where XXXXXXXX is 8 characters (letters or numbers). So, if your instance URL is "`https://s-9sdff10b.lucidworks.io`" you would use that in place of any example API calls that used "`http://localhost:8888`". For example, this call to get all collections:

```
curl 'http://localhost:8888/api/collections'
```

would be changed to:

```
curl -u 'API_Key:password' 'https://s-9sdff10b.lucidworks.io/api/collections'
```

The API_Key can be found by logging in to your LucidWorks Search instance, and clicking "My Account" at the upper right of the screen. Click "API Access" on the left to view the API key. The password is 'x' by default. There is not currently a way to change the default password. You should take care not to expose this key when posting to our forums, as that information could be seen by other LucidWorks Search customers.

For users on LucidWorks Search for Windows Azure, the above URL would be: `'https://s-9sdff10b.azure.lucidworks.io/api/collections'`.

Getting Support & Training

There are several options to get answers to questions besides this documentation:

- The [LucidWorks Search Forum](#) is a place to ask questions and share information about your implementation.
- The [LucidWorks Search KnowledgeBase](#) has articles written by our support and consulting staff around common issues and questions.
- [Training Videos](#) produced by the LucidWorks training team.
- Premium support is also available, providing access to a help desk ticketing system. For more information see [Lucene/Solr Support](#).

Custom Connector Guide

This guide discusses how to build a custom connector with the LucidWorks Connector Framework. It contains the following sections:

- [Introduction to Lucid Connector Framework \(see page 10\)](#): Provides a technical overview of how connectors work in LucidWorks Search and an introduction to the various components of the Framework.
- [How To Create a Connector \(see page 12\)](#): Provides detailed information about each component, including how to build a custom component and which parts of the example connector to reference while creating a connector.
- [Integrating Google Connectors \(see page 26\)](#): If there is a Google Connector Manager connector that you'd like to use with LucidWorks Search, here are some tips on how to integrate it.
- [Integrating New Crawlers with LucidWorks \(see page 34\)](#): Once the custom connector is made, LucidWorks Search needs to be able to discover it; this section describes how to do that.

Example Crawler

There is an example implementation of a crawler provided in the `$LWS_HOME/app/examples/java` directory of each LucidWorks installation. That directory also contains the source code for the example crawler, which can be used as a basis for any custom implementations.

The example crawler provides three sample data sources:

- a very simple crawler for local file system.
- a "random" data source that produces a random number of example documents.
- a "secure" data source that allows you to implement "security trimming" feature.

The examples also include a nested library to illustrate the concept of packaging crawlers with all dependent jars, and to show that crawlers can actually use such libraries.

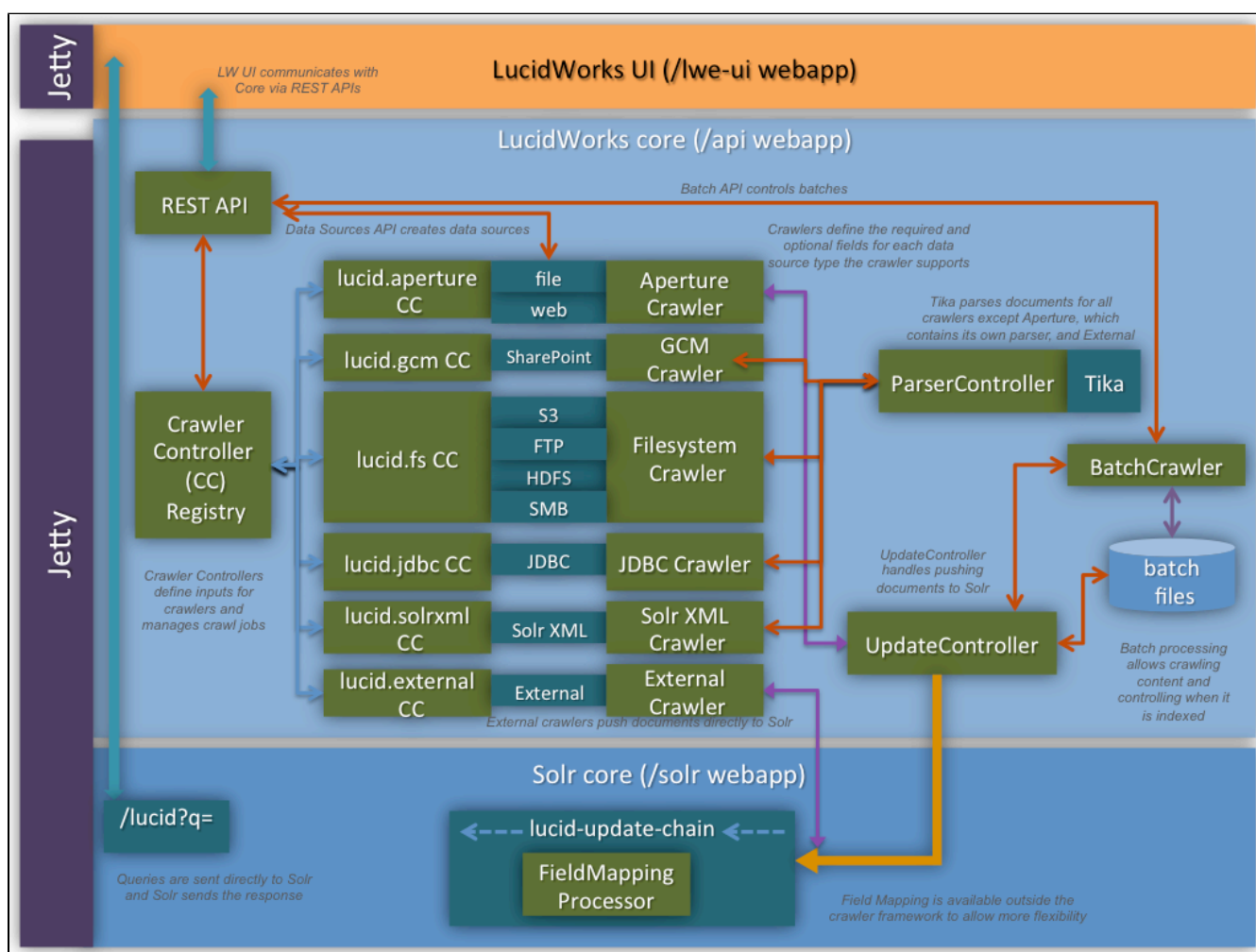
Details on how to build and use the example crawler can be found in the `README.txt` file in the `$LWS_HOME/app/examples/java` directory.

Introduction to Lucid Connector Framework

LucidWorks Search offers an open API for managing the process of content acquisition (crawling) from document repositories, and for adding new implementations for content acquisition (crawlers).

The main principle behind the design of this API was to isolate the core of LucidWorks Search both from the details of each crawler component implementation and the details of the indexing platform, and to allow for integration of externally-developed crawler components. Currently this API is in use and provides the integration for each crawler integrated with LucidWorks (whether developed by LucidWorks or as a 3rd party integration).

This graphic gives an overview of the Crawler architecture:



LucidWorks Crawler architecture

In general terms, each Crawler defines possible data source types and their parameters.

Each data source is defined by a `DataSourceSpec`, which lists all possible properties that a `DataSource` can take, their default values, and whether they are mandatory or optional. The `DataSourceFactory` for each `CrawlerController` defines the valid `DataSourceSpec(s)` for a particular crawler. The `DataSourceSpec(s)` for a crawler are also known as a type of data source, as a type must be unique for each `CrawlerController`. The `DataSourceFactory` may also contain validation rules for data source properties; for example, requiring that input to a property is a string instead of numeric.

The `CrawlerController` handles the scheduling and execution of crawl jobs. Each job has a unique identifier and the job definition is reusable. The job definition may include information to initiate a crawl at a specific time each day, for example. The status of each job is also managed by the `CrawlerController` with defined states such as `RUNNING`, `FINISHED`, `STOPPED` and others. Each crawl job definition also includes instructions for how to handle the output of the job.

All `CrawlerControllers` are created and managed by the `CrawlerControllerRegistry`.

Once data has been acquired from content sources, the `CrawlProcessor` defines how it is further processed. The `ParserController` is a document parsing and content extraction service and the `UpdateProcessor` represents the output for `SolrInputDocument(s)` to be indexed. The default LucidWorks `CrawlProcessor` currently uses Apache Tika v1.2 for content extraction and parsing from the raw data. The `UpdateProcessor` uses a SolrJ connection to the LucidWorks instance of Solr.

It's possible to handle the output of a crawl job as a batch, meaning that no parsing or indexing can take place (or a combination: no parsing but indexing or parsing but no indexing). The output of the job is stored for later processing - either to be parsed by a separate process or indexed at a more convenient time. Batches allow the crawl process to be split into three stages: fetching raw content; parsing content; and indexing content. A `BatchManager` handles these processes and stores the fetched content in a hierarchy of folders on the filesystem, which may consume a great deal of space depending on the content. Stored content is not automatically cleaned after indexing (if the content is ever indexed) to allow multiple indexing runs, so care must be taken to remove unwanted batches when they are no longer needed.

After raw documents have been fetched and their text and metadata extracted, an initial version of a `SolrInputDocument` is prepared. Each data source has a property called "mapping" which defines how to handle the incoming fields of each document. The mappings can specify a field to use as the unique key, map incoming fields to other previously defined fields, add fields to documents, or convert field types to other field types. The `FieldMapping` definitions are sent to Solr in JSON format to update the rules for the `FieldMappingProcessor`.

After `FieldMapping` has been completed, the documents are input to Solr and indexed.

How To Create A Connector

As described in the [introduction \(see page 10\)](#), the Lucid Connector Framework is an open API for managing the process of content acquisition, known as crawling, from document repositories and for adding new implementations of content acquisition, referred to as crawlers.

This section and the ones following it, describe how to create a custom connector. Each section will start with a technical overview of the component and then discuss the classes that are required to create a custom component for your own needs. An example connector is also provided and can be found in `$LWS_HOME/app/examples/java`.

The following sections will discuss each of the Connector Framework components:

- `CrawlerController` facade, responsible for defining and controlling the crawl jobs executed by a given crawler platform. `CrawlerController` instances are created by `CrawlerControllerRegistry`. This is the central class in the API, and users interact with crawler platforms via methods defined in this class.
- `DataSource` which describes in an abstract way the configuration parameters for accessing a content repository and the set of documents to retrieve. `DataSource`-s are created by a `DataSourceFactory` specific to a concrete crawler platform.
- Closely related to `DataSource`-s are `DataSourceSpec`-s, which are descriptors that define what properties can be set, their default values and how the values set by a user can be validated.
- Crawl jobs run by `CrawlerController`-s are represented by `CrawlState`, and their status can be obtained from `CrawlStatus`.
- `CrawlProcessor` is an abstraction for processing the output of a crawl job. This in turn uses a `ParserController` and `UpdateController`, responsible for parsing and indexing the results of a crawl job.
- `BatchManager` handles batch jobs, which are crawl jobs that don't immediately send their output to a `CrawlProcessor` for parsing and indexing, but instead store the raw content for later processing (either parsing + indexing, or just indexing).
- `FieldMapping` and `FieldMappingUtil` constitute a metadata mapping facility to map metadata extracted from documents to the target index fields. This facility includes some rudimentary type conversion.

The following graphic shows the architecture of the components in detail:



A simple example crawler is provided in the `$LWS_HOME/app/examples/java` directory of a LucidWorks installation. That directory also contains the source code for the example crawler, which can be used as a basis for a custom crawler.

To build a custom crawler you need to write the following classes. In each example, the crawler name should begin each class name; replace `crawler` with the crawler name when creating the files.

- `crawlerDataSourceSpec`: Derived from `DataSourceSpec`. Defines data source properties, their default values and validations.
- `crawlerDataSourceFactory`: Derived from `DataSourceFactory`. It is responsible for reporting the list of supported data source types and their specifications. The actual creation of data source instances from a map of parameters is done in `DataSourceFactory`.
- `crawlerCrawlerController`: Derived from `CrawlerController`. Defines the startup, stop and reset of the crawler.
- `crawlerCrawlState`: Derived from `CrawlState`. Defines stop and start for a daemon-thread the crawler should run with.

-
- `crawlerCrawler`: Derived from `Runnable`. The traversal of the `DataSource` is done here.

DataSource and DataSourceSpec

Overview

The `DataSource` class is essentially a wrapper for a set of properties, with some specialized methods and constructors that enforce providing some details. `DataSourceFactory` (an abstract class, whose implementation is specific to a specific crawler platform) knows what kind of data sources are supported by the platform, and it also knows how to validate a set of provided properties to verify whether they can define a valid `DataSource` for this platform. For this purpose the `DataSourceFactory` also keeps a registry of `DataSourceSpec`-s. These "specs" list all possible properties that a `DataSource` can take, their default values and whether they are mandatory. `DataSourceSpec` also provides a limited conversion facility (casting) of input data to the formats and types expected by a corresponding `DataSource` (e.g., it's common for numeric parameters to be supplied by a UI as strings - the casting then uses `Validator` subclasses to convert such strings to a numeric format, so that other parts of the API will deal only with the expected types of the properties).

`DataSource` type is an arbitrary string identifier that must be unique in the scope of a given crawler platform. `DataSource` category is a purely informative string that may provide hints to the user about how to present documents retrieved from this source (e.g. files, database records, ...).

`DataSource` instances should be viewed as purely passive data containers. Any state related to crawl jobs should be kept in implementation-specific subclass of `CrawlState`.

The properties of a `DataSourceSpec` are those that are entered by a user when creating a new data source of this type (either through the UI or via the API). Careful consideration should be given to which properties should be required and each property should have as good a default as possible. Validators that match the expected type of the value input for a property should be called to catch configuration errors as early as possible (e.g., before they produce errors in the crawl). Several predefined validators are available which match many common types (such as int, float, URL, etc.) and they should be used where appropriate.

LucidWorks uses underscores for property names in each of our crawler implementations instead of CamelCase (so, a property is called "ignore_robots" instead of "ignoreRobots"). Following the same standard may help your custom crawlers appear to work the same as the included set of crawlers.

The properties are also used by the Admin UI to dynamically create an entry form for users to configure new data sources. The flexibility of this approach allows new crawlers to be added, or properties of a data source modified, without having to create or update forms for each data source type. There are some conventions the UI uses when reading the properties for a specific data source. First, property names are normalized and used as form labels. The normalization removes any underscores and the first word is capitalized. A property name such as "access_token" is transformed to display as "Access token". Property descriptions are used to display information to the user about what kind of information is expected for that attribute. Descriptions are optional, but if they are used, they should be kept short.

Custom Classes for This Component

In the example crawler provided with LucidWorks, you can find examples of customized classes for this component in

`$LWS_HOME/app/examples/java/crawler/src/java/com/lucid/examples/crawl.`

To build a custom crawler you need to write the following classes for the DataSource component. In each example, the crawler name should begin each class name; replace `crawler` with the crawler name when creating the files.

- `crawlerDataSourceSpec`: Derived from `DataSourceSpec`. Contains crawler-specific properties. The super-constructor from `DataSourceSpec` has to be called with the Category of the Crawler (if there's no appropriate Category please use `Category.Other` instead).
- `crawlerDataSourceFactory`: Derived from `DataSourceFactory`. Registers supported crawler types and their `DataSourceSpec`-s. The `CrawlerController` has to be set as a parameter in the super-constructor. The `DataSourceSpec`-s should be registered using the following calls:

```
public ExampleDataSourceFactory(CrawlerController cc) {
    super(cc);
    // map type names to specifications
    types.put("xfile", new XFileSpec());
    types.put("xrandom", new XRandomSpec());
    types.put("xsecure", new XSecureSpec());
}
```


CrawlerController, CrawlStatus and CrawlerControllerRegistry

Overview

The abstract class `CrawlerController` models interactions with a crawler implementation. `CrawlerController` instances are created and managed by a `CrawlerControllerRegistry`. Each crawler provided in a separate jar (including some built-in crawlers and all third-party crawlers) is loaded using its own classloader - this way crawler implementations are isolated and can use conflicting versions of dependencies.

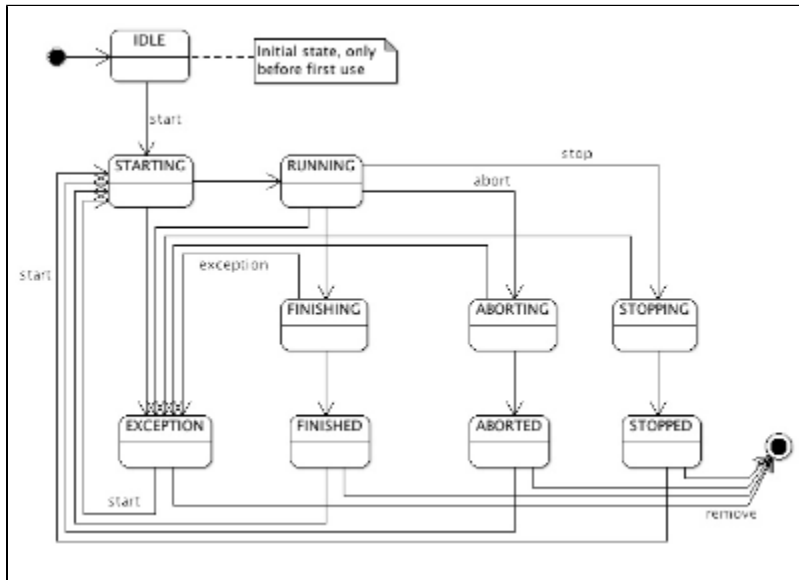
Although the singleton pattern for `CrawlerController`-s is not enforced, it is followed in practice through the use of `CrawlerControllerRegistry` (which is a singleton in LucidWorks Search).

A crawl job is the process of going to a document repository and retrieving documents.

The life-cycle of a crawl job consists of the following states:

- The crawl job is defined and registered with the crawler platform using `CrawlerController.defineJob(...)`. This is also a chance for the crawler platform to perform additional verification of the crawl parameters. A defined job gets a unique identifier, and the job definition with this identifier is reusable. An internal component `CrawlerController.jobStateMgr` manages this definition in memory. For crawler platforms that run in separate processes there may be a need to synchronize this internal job status with the external process; the crawler API makes no such provisions, since the details of this process are implementation-specific.
- The crawl job is started using `CrawlerController.startJob(...)`. The job itself is then being executed asynchronously (in a separate thread). This call should return quickly and must not block for the duration of the crawl job. In the current API there is provision only for one running crawl job per job definition.
- A transitory state is associated with the process of starting a job called "STARTING", which is expected to last relatively shortly, after which the job transitions to a "RUNNING" state or to one of the final failure states (see below). Most of the work for the crawl job is expected to take place in the RUNNING state. When the work is finished in an orderly manner, the job transitions to a FINISHING state, where the necessary commits and cleanups are performed, and then finally transitions to a FINISHED state.
- A running crawl job can be stopped or aborted. When a crawl job is stopped, the `CrawlerController` will make an attempt to preserve as much of the partial crawl results as possible, and stops the job in an orderly manner. When the crawl job is aborted there are no such guarantees; however, partial results may still become visible and committed to the index. If this is not desirable the `CrawlerController` implementation may track documents by adding a run identifier (`batch_id`) and then issue a delete request with this `batch_id`. There are two transitory states associated with these actions: STOPPING and ABORTING. During these states it's expected that the controller will perform necessary cleanups. Final states after these actions are STOPPED and ABORTED, respectively.
- In case of a non-recoverable error the job goes into a final EXCEPTION state.

The following shows the various crawl job states and how they interact with one another:



Crawl Job States

Crawl job status can be retrieved using `CrawlerController.getStatus(...)` or all running and recently finished crawl jobs can be listed using `CrawlerController.listJobs(...)`. This list is maintained in memory, so it's cleared on restart. There is also a persistent job history that is maintained with `CrawlerController.getHistoryRecorder().record(crawlStatus)`.

Crawler platforms that want to support batch operations should return a non-null implementation of `BatchManager`, such as the provided `SimpleBatchManager` that stores intermediate crawl results in local files or another method.

[Back to Top \(see page \)](#)

Custom Classes for This Component

In the example crawler provided with LucidWorks, you can find examples of customized classes for this component in

`$LWS_HOME/app/examples/java/crawler/src/java/com/lucid/examples/crawl`.

To build a custom crawler you need to write the following classes for the `CrawlerController` component. In each example, the crawler name should begin each class name; replace `crawler` with the crawler name when creating the files.

- `crawlerCrawlerController`: Derived from `CrawlerController`. Defines the startup, stop and reset of the crawler.
- `crawlerCrawlState`: Derived from `CrawlState`. Defines stop and start for a daemon-thread the crawler should run with.

For every derived class there are some methods that can be overridden:

- `crawlerCrawlerController`:
 - `reset(String collection, String dataSourceId)`: tells the crawler to reset the used data source, to clean up timestamps used to reinitialize the crawl state etc.
 - `resetAll(String collection)`: tells the crawler to reset all data sources
 - `CrawlId defineJob(DataSource ds, CrawlProcessor processor)`: Initializes the `CrawlState`. The `DataSource` and the `CrawlProcessor` have to be registered like this:

```
ExchangeCrawlState state = new ExchangeCrawlState();
state.init(ds, processor, this.historyRecorder);
this.jobStateMgr.add(state);
```

- `startJob(CrawlId descrId)`: starts the job with `CrawlState.start()`
 - `stopJob(CrawlId jobId)`: stops the job
 - `abortJob(CrawlId jobId)`: aborts the job (can be the same as `stopJob`)
- `crawlerCrawlState`:
 - `start()`: starts a Crawler over a daemon-thread

[Back to Top \(see page \)](#)

Crawler

Overview

The `Crawler` does the work of traversing the Data Source and collecting the data for input.

Custom Classes for this Component

In the example crawler provided with LucidWorks, you can find examples of customized classes for this component in

`$LWS_HOME/app/examples/java/crawler/src/java/com/lucid/examples/crawl.`

To build a custom crawler you need to write the following classes for the Crawler component. The crawler name should begin each class name; replace `crawler` with the crawler name when creating the files.

- `crawlerCrawler`: Derived from `Runnable`. The traversal of the `DataSource` is done here.

All classes have to override some methods from the interfaces they are derived from or call a method of the super class to get a successful registration in the LucidWorks framework.

- `crawlerCrawler`:
 - `run()`: Overridden by the interface `Runnable`. Here the traversal of the data source is done.
 - `stop()`: Overridden by the interface `Runnable`.

CrawlProcessor, ParserController and UpdateController

Overview

Each crawl job definition should specify a sink for the output of the crawl job. A null value may be provided to mean the default `CrawlProcessor` implementation that uses the `TikaParserController` and `UpdateController` depending on the setting of the datasource (see below). This default implementation can be also configured to persist raw results of the crawl job as a batch when the "caching" property is true for a datasource. Document parsing and indexing can also be turned off (and the data stored in a batch) when a "parsing" property of a data source is set to false.

The `CrawlProcessor` exposes a minimal API to simply consume the raw output documents plus the protocol-level metadata. Alternatively, it can consume a `SolrInputDocument` if the parsing process was already performed. Usually only one of the methods is called; specific implementations of `CrawlProcessor`-s pass data between these methods as automatically as necessary.

The default `CrawlProcessor` that comes with LucidWorks (the one that is instantiated when no `CrawlProcessor` is specified) uses internally two other abstractions:

- `ParserController` represents a document parsing and content extraction service. The default implementation of this abstract class in LucidWorks is called `TikaParserController` and uses Apache Tika v1.2 for content and metadata extraction.
- `UpdateController` represents the output for `SolrInputDocument`-s to be indexed. The default implementation of this component obtained from `UpdateController.create(...)` uses a SolrJ connection to the Lucidworks instance of Solr as output when the "indexing" property is set to true, otherwise it stores parsed documents in batch data.

Custom Classes for This Component

None required, unless the default `CrawlProcessor` is not sufficient.

BatchManager

Overview

`BatchManager` is a component that is responsible for persisting and managing intermediate crawl results. This allows the crawling process to be split into up to three stages:

- fetching raw content from remote repositories
- parsing content, as well as text and metadata extraction
- indexing the extracted data as `SolrInputDocuments`

The advantage of this functionality is that the fetching of the data is usually the most costly step in the crawling pipeline, and it's sometimes better to execute the parsing and indexing at a different time than the fetching. It can also be used to re-do some of the steps e.g., after fixing configuration errors (either in field mapping or in the `schema.xml` file). The disadvantage is that it complicates the data flow and consumes additional disk space, but that may be an acceptable tradeoff.

LucidWorks comes with an implementation of this API called `SimpleBatchManager`. This implementation stores batch data in a hierarchy of folders that in turn contain record-oriented files. `SimpleBatchManager` creates separate folder hierarchies for each crawler, each crawl job, and each crawl job run that resulted in some batch data. The files are the following:

- `batch.status` - describes the status of the batch, e.g. how many raw documents are present, how many parsed documents, timestamp, etc.
- `content.raw` - contains the raw content of retrieved documents together with protocol-level metadata.
- `solr.json` - contains `SolrInputDocument`-s ready for indexing, in JSON format.

A `CrawlerController` implementation supports batch operations when it provides a non-null instance of a `BatchManager`. The following operations can be performed on batch data:

- Batch data can be created by setting appropriate options in `DataSource`-s, when the default implementations of `CrawlProcessor`, `ParserController` and `UpdateController` are used. There is also a lower-level API available for writing individual records to a specified batch, see the `ContentFileWriter` and `SolrFileWriter` javadoc for more details.
- `BatchManager.listBatchStatuses(...)` (or a REST API under `/api/collections/<collection>/batches`) can be used to retrieve a list of available batch data sets.
- A batch processing job can be started with `CrawlerController.startBatchJob(...)`, where the user can specify the output of the batch processing, whether the content should be parsed (or re-parsed) and/or indexed, using the supplied `CrawlProcessor` instance or a default one when `null` is provided.

- running batch jobs can be listed with `CrawlerController.listBatchJobs()`, or otherwise controlled in the same way as regular crawl jobs, i.e. stopped or aborted. They also undergo the same state changes as regular crawl jobs.

Obsolete or no longer needed batch data can be deleted using `BatchManager.deleteBatch(...)` or using the bulk delete `BatchManager.deleteBatches(...)`. Batch data is not deleted by default, so must be managed to ensure the batches do not consume too much disk space.

Custom Classes for This Component

None. Customizations to support batch operations are defined in the custom `CrawlerController`.

FieldMapping and FieldMappingUtil

Overview

After the raw documents are parsed and their text and metadata are extracted, an initial version of `SolrInputDocument` is prepared (this usually takes place somewhere in a `CrawlProcessor` or a `ParserController` that it uses). Since the metadata names in documents coming from various sources can be pretty arbitrary it is necessary to normalize their values and to map their names to Solr fields valid for the current index schema. This is the role of the `FieldMapping` and the `FieldMappingUtil` helper class.

Each `DataSource` has a property "mapping" that contains an instance of `FieldMapping` (if there was none specified on `DataSource` creation then a default one with the default mappings will be provided). You can examine the details of the default field mapping by looking through the response of Data Sources REST API.

Field mappings are specified per data source, and then passed to `UpdateController`. The process of field mapping is performed by the `UpdateController` so usually it should not be invoked explicitly. `UpdateController` implementations may further verify the mappings using the current Solr schema so that the mappings produce valid fields.

The mappings consist of the following main areas:

- `uniqueKey` - this property specifies the name of the uniqueKey in the Solr schema. This value is verified with the current schema when a new crawl job is defined.
- `mappings` - this is a map of source metadata names to target field names. Source names are case-insensitive. A value of null means that this metadata should be discarded. See below for the details of the mapping algorithm.
- `literals` - this is a map of key/value pairs that define fields to be added to every document.
- `types` - defines any special field types if a conversion is necessary from the default STRING type. Currently recognized types are STRING, INT, LONG, FLOAT, DOUBLE and DATE. If a type is not specified then STRING is assumed.
- `multiVal` - this is a map of field names and boolean values. True means that the target index field with this name supports multiple values. False (or a missing key/value) means that the field is single-valued.
- `dynamicField` - this property specifies a prefix for dynamic fields if a more specific mapping is missing. See also below for the details of the mapping algorithm.
- `defaultField` - this property specifies a name of the default field if a more specific mapping is missing. See also below for the details of the mapping algorithm.
- `datasourceField` - this property specifies a prefix of the fields that preserve data source id, data source type and data source display name.
- `lucidworks_fields` - this boolean property (defaults to true) indicates that LucidWorks-specific fields should be automatically added to incoming documents (such as `data_source id`, `data_source_name` and `data_source_type`).

Field mapping is usually created and initialized in `DataSourceFactory.create(...)` method when the datasource is initially created. During job startup this mapping is passed to `UpdateController`, which can optionally verify field mappings with the current index schema for the collection specified in the data source. If a field is missing in the schema then it's mapped to null (i.e., discarded). The arity and the type of the field is checked and set appropriately. This process is repeated each time `CrawlerController.startJob(...)` is called.

The process of mapping source metadata names to the target field names works like this (note: this process is already encapsulated in one of the existing `UpdateController` implementations):

- first a case-insensitive match is tried with the source names present in the "mappings". If a value is found then it's returned (a value of null should be interpreted as "discard").
- then if `dynamicField` is non-null the source name is converted to a dynamic field name like this: to the value of `dynamicField` an underscore is appended, and then an escaped version of the source name is appended (the escaping replaces any non-word character with an underscore).
- then if `defaultField` is non-null then the value of `defaultField` is returned
- finally, the source name is returned.

Internally, this process uses a helper class `FieldMappingUtil`. This class contains methods to:

- initialize field mappings with values suitable for Aperture or Tika parsing
- verify the mapping with the current index schema
- normalize fields - this normalization step should always be performed to make sure that the `SolrInputDocument` instances contain only fields valid for the current schema, with correct multiplicity and correct type. The normalization works like this:
 - if a field is defined as type DATE then the value is checked - if it's an instance of `java.util.Date` then it's left unchanged, otherwise the string representation of the value is parsed using Solr's `DateUtil` to obtain a valid `Date` instance. In case of parsing errors the offending value is discarded.
 - if a field is defined as single-valued but multiple values are present then:
 - if a field is type DATE only the first value is retained, all other values are discarded
 - otherwise a set of unique string representations of values is concatenated using single space character, and the original multiple values are replaced with this single concatenated value.
 - there is also some other special treatment for the "mimeType" field to avoid common Tika and Aperture parsing errors.
- `addLucidworksFields` - this method ensures that some fields necessary for the LucidWorks UI are populated.

Custom Classes for This Component

None necessary or required.

Integrating Google Connectors

LucidWorks ships with a connector for SharePoint repositories that uses the Google Connection Manager (GCM), which has been integrated with LucidWorks Search. SharePoint is only one of several available connectors, however, and others can be integrated by following the process defined below. Other repositories that can be crawled using GCM connectors include Documentum, IBM FileNet, LDAP, and Lotus Notes.

The process to integrate a new connector is simpler than developing a crawler from scratch, but requires understanding many of the same concepts. Before proceeding, please review the [Introduction to Lucid Connector Framework \(see page 10\)](#). With this procedure we'll create a `DataSourceSpec` and then register it with LucidWorks. Once created, the new connector will be a type of data source for the `lucid.gcm` crawler (this would only be important if you use the Data Sources API - if you use the Admin UI only, you won't notice a difference between the other data source types).

It may also be helpful to review the [Connector Developer's Guide](#) from Google. LucidWorks is using Google Connector Manager v2.8.6.

Covered in this section:

- [Overview of the Development Process \(see page 26\)](#)
- [Preparing for development \(see page 27\)](#)
- [Writing the DataSourceSpec \(see page 28\)](#)
 - [Extract Fields from the Connector HTML Form \(see page 28\)](#)
 - [Writing the DataSourceSpec \(see page 31\)](#)
- [Registering the Extension \(see page 32\)](#)
- [Deployment \(see page 33\)](#)

Overview of the Development Process

There are several steps described below, summarized here:

1. Prepare the development environment by checking the requirements.
 - a. Download and install LucidWorks. Do not start it during installation, or stop it if it is already running.
 - b. Download a compatible Google Connector and extract the files.
 - c. Start LucidWorks with a special port.
2. Use that sample Google Connector project included with LucidWorks to extract the connector configuration form and use that information to create an appropriate `DataSourceSpec`.
3. Implement a `DataSourceSpec` class, extend it to register the `DataSourceSpec`, then write a configuration file for the java service loader.
4. Deploy the `.jar` file.

5. Start LucidWorks and use the new connector.

Preparing for development

The included sample project requires the following installed on the system where development will be completed:

- Ant 1.8+
- Java 1.6+
- LucidWorks Search

To properly prepare the development environment, perform these steps:

1. Download and install LucidWorks. More information on installing LucidWorks is available in the section on Installation. If you have already installed it and have it running, stop it.
2. Download a compatible Google connector. Connectors can be found at <https://code.google.com/p/google-enterprise-connector-manager/>. Review the release notes for the candidate connectors to make sure they work with GCM v2.8.
3. Extract the connector files from the downloaded Connector archive and copy the `.jar` files from `lib` directory to `$LWS_HOME/app/webapps/connector-manager/WEB-INF/lib/`. If there are duplicate `.jars` in the target directory, you may need to resolve any conflicts. For example, you can test it with the example google ldap connector (see `$LWS_HOME/app/examples/google-connector/google-connector-ldap-2.8.4.jar`)
4. Start LucidWorks with following command:

```
app/bin/start.sh -lwe_connectors_java_opts "-DlucidworksGCMPort=10000"
```
5. After starting, verify there are no errors in the logs (see also System Directories and Logs for more information on logs). Pay particular attention to possible errors in the `connectors-<date>.log`.
6. Go to the `$LWS_HOME/app/examples/google-connector` directory. This folder contains an ant buildable project that can be used as a starting point when implementing new integrations.
7. Localize the sample project by editing the `build.xml` file, and modifying the `gcm-url` property. The port must be changed to the port that was specified when starting LucidWorks.
8. Run `ant dist` to verify that the environment is successfully configured. If the build does not succeed, correct any errors in `build.xml`.

At this point, you should be ready to follow the rest of these instructions to extract the required information for the custom connector for LucidWorks.

[Back to Top \(see page \)](#)

Writing the DataSourceSpec

When integrating Google connectors into LucidWorks Search, the majority of the work is to make sure the new data source type can be used with Admin UI. By itself, a Google Connector provides a UI through a HTML form. In LucidWorks, however, the way to provide the configuration UI is based on `DataSourceSpec-S`. A `DataSourceSpec` is a class that provides sufficient information for the UI so that it can render the configuration screen (data source configuration screens in LucidWorks are dynamically generated based on information provided to it by the data source). Sometimes making the Admin UI work with the new connector is a straightforward task, but sometimes it requires some additional effort.

The configuration screen should provide the user with information about which fields are required and if possible, provide some guidance about what format to enter information. Unfortunately, at this point the process of figuring out what fields a Google Connector requires a little bit of reverse engineering (to figure out the required parameters and their format from the HTML form) and some trial and error.

There are several steps to writing the `DataSourceSpec` for the connector, explained in each of the sections below:

1. [#Extracting Fields from the Connector HTML Form \(see page \)](#)
2. [#Writing the {{DataSourceSpec}} \(see page \)](#)

[Back to Top \(see page \)](#)

Extract Fields from the Connector HTML Form

Once the environment is properly configured and the sample project is set up for development, you can use the ant target `ant list-connectors` to show the GCM connectors installed in an embedded Google Connector Manager web application in LucidWorks. You'll need the name of the connector in order to retrieve the HTML form. The output of this command looks like this (the WARN messages are fine):

```
sh> ant list-connectors
...
[java] Available connectors:
[java]
[java] - LDAPConnectorType
[java] - sharepoint-connector
...
```

These lines show the GCM connectors installed on the system.

Next, retrieve the connector configuration HTML form by executing `ant get-connector-form -Dconnector=<name of the connector> (for example, ant get-connector-form -Dconnector=LDAPConnectorType)`. The output of this command is a file called `connector-form.html` that you can find in the example project main directory. You will need to examine this file for the required fields and their valid formats. Then you can use that information to write the `DataSourceSpec` for the connector.

For example, the HTML form for LDAP connector looks like this:

```
<tr>
  <td><br />
    <b>LDAP Connector Configuration<span style="color:
#FF0000"><sup>Preview</sup></span></b></td>
</tr>
<tr>
  <td valign="top"><label for="hostname">LDAP Directory
    Server Host</label></td>
  <td><input name="hostname" id="hostname" type="text"></input></td>
</tr>
<tr>
  <td valign="top"><label for="port">Port number</label></td>
  <td><input name="port" id="port" type="text" value="389"></input></td>
</tr>
<tr>
  <td valign="top"><label for="authtype">Authentication Type</label></td>
  <td><select name="authtype" id="authtype">
    <option value="ANONYMOUS" selected="selected">Anonymous</option>
    <option value="SIMPLE">Simple</option>
  </select></td>
</tr>
<tr>
  <td valign="top"><label for="username">LDAP Binding
    Distinguished Name (DN)</label></td>
  <td><input name="username" id="username" type="text"></input></td>
</tr>
<tr>
  <td valign="top"><label for="password">LDAP Binding
    Password</label></td>
  <td><input name="password" id="password" type="password"></input></td>
</tr>
<tr>
  <td valign="top"><label for="method">Connection Method</label></td>
  <td><select name="method" id="method">
    <option value="STANDARD" selected="selected">Standard</option>
    <option value="SSL">SSL</option>
  </select></td>
</tr>
<tr>
  <td valign="top"><label for="basedn">LDAP Search Base</label></td>
```

```
<td><input name="basedn" id="basedn" type="text"></input></td>
</tr>
<tr>
  <td valign="top"><label for="filter">User Search Filter
    (only these users will be indexed)</label></td>
  <td><input name="filter" id="filter" type="text"></input></td>
</tr>

<tr style='display: none'>
  <td><input type='hidden' id='schemavalue' name='schemavalue'
    value='[]' />
  <script type="text/javascript">
    function getIndexOf(arr, value) {
      for ( var i = 0; i < arr.length; i++) {
        if (arr[i] == value)
          return i;
      }
      return -1;
    }
    var schemaList = new Array();
    function appendToSchema(chkbox) {
      if (schemaList.length == 0) {
        schemaList = JSON
          .parse(document.getElementById('schemavalue').value);
      }
      if (chkbox.checked) {
        schemaList.push(chkbox.value);
        document.getElementById('schemavalue').value = JSON
          .stringify(schemaList);
      } else {
        if (getIndexOf(schemaList, chkbox.value) >= 0) {
          schemaList.splice(getIndexOf(schemaList, chkbox.value), 1);
          document.getElementById('schemavalue').value = JSON
            .stringify(schemaList);
        }
      }
    }
  </script>
</td>
</tr>
```

From that it is possible to see that there are 8 parameters: hostname, port, authtype, username, password, basedn, filter and schemavalue statically defined in the HTML form. Closer inspection shows that the schemavalue field is dynamically build with javascript (it is actually based on the content of the LDAP server) and the format for that field is ["<field1>", "<field2>"].

[Back to Top \(see page \)](#)

Writing the DataSourceSpec

The next step is to write a `DataSourceSpec` definition based on this information. Specs provide information for the crawler to know which attributes are required, how the attributes should be defined, and how the Admin UI should create a form to allow GUI configuration of the data source.

For all Google Connector-based data sources there is a base class called `GCMSpec` that can be extended to the needs of the new Google Connector.

From the example project the class implementing the UI spec for the LDAP connector is called `GCMLDAPSpec` and it looks like this:

```
...
public class GCMLDAPSpec extends GCMSpec {
...
    public GCMLDAPSpec(LWEGCMA adaptor) {
        super(adaptor);
    }

    @Override
    protected void addCrawlerSupportedProperties() {
        super.addCrawlerSupportedProperties();
        addSpecProperty(new SpecProperty(HOSTNAME, "Hostname",
            String.class, "", Validator.NOT_BLANK_VALIDATOR, true));
        addSpecProperty(new SpecProperty(PORT, "Port",
            Integer.class, 389, Validator.NON_NEG_INT_STRING_VALIDATOR, true));
        // all the other properties
    }

    @Override
    public FieldMapping getDefaultFieldMapping() {
        FieldMapping fieldMap = new FieldMapping();
        fieldMap.defineMapping("GCM_dn", "dn");
        fieldMap.defineMapping("GCM_cn", "cn");
        fieldMap.defineMapping("GCM_uid", "uid");
        return fieldMap;
    }
}
```

When deployed to LucidWorks, the data source will look like this in the Admin UI:

Dashboard > collection1 Help

Status

Indexing

Querying

Access Control

Tools

Advanced

Settings

Data Sources

Fields

Dynamic Fields

Field Types

JDBC Drivers

Schedules

Ldap

* Name

Test OpenDS Server

* Hostname

localhost

Hostname

* Port

1389

Port

Authtype

SIMPLE

Authtype (ANONYMOUS/SIMPLE)

Username

cn=Directory Manager

LDAP username

Password

.....

LDAP password

Method

STANDARD

LDAP connection method (STANDARD/SSL)

* Basedn

dc=example,dc=com

LDAP Search base

* Filter

(objectclass=inetOrgPerson)

LDAP Search filter (Only these entries will be indexed)

* Schemavalue

["dn","cn","uid"]

A list of indexed LDAP attributes in the form of ["cn","uid"]

Advanced

show

Create

Cancel

[Back to Top \(see page \)](#)

Registering the Extension

Finally the implemented classes needs to be registered into LucidWorks Search so that the new Google Connector is recognized. This is done with a specific class that implements `GCMExtension`:

```
...
public class GCMLDAPEExtension extends LWEGCMAdaptor implements GCMExtension {

    @Override
    protected void customizeProperties(Map<String, Object> dsProperties, HashMap<String,
String> gcmProperties) {
        gcmProperties.put(GCMSpec.CONNECTOR_TYPE, "LDAPConnectorType");
    }

    @Override
    public void register(Map<String, com.lucid.crawl.datasource.DataSourceSpec> types) {
        LWEGCMAdaptor.register("ldap", this);
        types.put("ldap", new GCMLDAPSpec(this));
    }
}
```


There are two methods in this class. The first is `register`, which registers the new connector. In the other method, `customizeProperties` allows modifications to the parameters sent from the Admin UI so they match what GCM expects (this may just be desirable, or it may be mandatory).

Some examples of transformations that might be needed are:

- Set values that are required by the connector or GCM but not exposed in the Admin UI (for example, the `CONNECTOR_TYPE`)
- Build (sometimes cryptic) strings that are normally built with javascript by the connector HTML form from static set of fields specified in the Spec

One final step is required, because the extensions are implemented by using the Java Service Loader. A file in the example project called

`src/main/resources/META-INF/services/com.lucid.crawl.gcm.GCMExtension` lists the available GCMExtensions. You need to add the name of the class implementing GCMExtension in that file.

[Back to Top \(see page \)](#)

Deployment

During the development phase the ant target `ant deploy` can be used. When everything is working as expected the `.jar` for the integration glue can be created by running `ant dist` that will create a `.jar` file in the `dist` directory. This `.jar` file can then be added inside `gcm-crawler.jar` in directory `$LWS_HOME/app/crawlers/` (the `deploy` target does this automatically).

Once the connector has been deployed, restart LucidWorks Search and the new connector should be available to use via the Admin UI or with the Data Sources API.



During development it might be necessary to cleanup the GCM configuration and state in `$LWS_HOME/conf/gcm/connectors/<connector-name>` by running `"rm -rf"` inside that directory.

[Back to Top \(see page \)](#)

Integrating New Crawlers with LucidWorks

Register the Crawler

Once a new crawler has been created, it needs to be registered with the `CrawlerControllerRegistry` in order to work with LucidWorks. In simple terms the way to do this is to follow these steps:

1. Create a `MANIFEST.MF` file for the plugin `.jar` with the required properties.
2. Place the `.jar` in the `$LWS_HOME/app/crawlers` directory.
3. Restart LucidWorks.

Create `MANIFEST.MF` File

LucidWorks requires the following entries in the crawler plugin's jar `META-INF/MANIFEST.MF` file. These properties may be placed either in the main section of the manifest, or in multiple sections (e.g., one section per crawler implementation):

- **Crawler-Alias:** (required) this is a symbolic name under which this crawler implementation will be known to LucidWorks. For example, the built-in Aperture crawler is registered under alias "lucid.aperture". Implementors should pick a meaningful name that is unique.
- **Crawler-Class:** (required) this is a fully-qualified class name of the `CrawlerController` implementation. For example, the built-in Aperture crawler's class is "com.lucid.crawl.aperture.ApertureCrawlerController".
- **Crawler-Exclude:** (optional) this is a whitespace-separated list of packages and fully-qualified class names that are excluded from loading using this class loader, instead their look-up and loading will be delegated to the parent classloader. In some cases nested jars may provide classes that conflict with other classes loaded from the parent classloader. Names of packages and classes on this list are treated as plain string prefixes and regular expressions are not supported.

An example `MANIFEST.MF` file for the included Aperture-based crawler looks like this:

```
Manifest-Version: 1.0
Ant-Version: Apache Ant 1.8.2
Created-By: 1.6.0_29-b11-402-11M3527 (Apple Inc.)
Crawler-Alias: lucid.aperture
Crawler-Class: com.lucid.crawl.aperture.ApertureCrawlerController
Crawler-Exclude: javax.xml.namespace
```

Loading the JAR

Each crawler class is loaded in a separate classloader, together with its dependencies. Crawler controller implementations are loaded by `CrawlerControllerRegistry` from JAR files, typically found in `$LWS_HOME/app/crawlers`. These jar files contain both the main `CrawlerController` implementation class, all other related classes (such as `DataSourceFactory` subclass), and may also contain nested .jar-s with dependencies (libraries) used by the implementation. A special class loader is used to load these classes, which unlike the default classloader:

- can discover and load classes from nested jar files.
- prefers classes found in the crawler jar over classes found in the parent classloader. This means that you can implement crawlers that use different, possibly mutually conflicting versions of dependencies.
- processes the crawler .jar's META-INF/MANIFEST.MF file looking for specific entries, and initializes the crawler plugin.

This process is executed during Lucidworks start-up as a part of `CrawlerControllerRegistry` initialization. This means that it's sufficient to just put a crawler plugin jar in `$LWS_HOME/app/crawlers` for LucidWorks to discover it and initialize it.

How the Admin UI Reads Crawlers

The LucidWorks Admin UI has been designed to dynamically read available crawler and data source types and display the list based on the currently enabled crawlers. When a specific data source type has been selected by the user, the UI also dynamically draws the screen with the latest available properties. So, once a new crawler is completed and properly registered (as above), then it's enough to restart LucidWorks to see the data source in the UI.

There are some conventions the UI uses when reading the properties for a specific data source. First, property names are normalized and used as form labels. The normalization removes any underscores and the first word is capitalized. A property name such as "access_token" is transformed to display as "Access token". Property descriptions are used to display information to the user about what kind of information is expected for that attribute. Descriptions are optional, but if they are used, they should be kept short.

About LucidWorks

LucidWorks (formerly known as Lucid Imagination) is the trusted name in Search, Discovery and Analytics, delivering the only enterprise-grade embedded search development solution built on the power of the Apache Lucene/Solr open source search project. Founded in 2008, the company initially provided support, consulting services, documentation and training for the Apache Lucene/Solr open source search project.

Within a few years, the LucidWorks team realized the need to add value to the open source search platform by developing an extensive layer of services which made Lucene/Solr secure and easier to use and manage. The company shipped the first version of its flagship product, LucidWorks Search, in 2011, followed by LucidWorks Big Data in May 2012. LucidWorks continues to offer support, documentation, consulting services and training products for Lucene/Solr.

LucidWorks remains committed to giving back to the Apache Lucene/Solr community. Out of the 37 Core Committers to the Apache Lucene/Solr project, 9 individuals work for LucidWorks, making the company the largest supporter of open source search in the industry. Further, LucidWorks hosts the Lucene Revolution, a conference dedicated to sharing ideas and promoting the Apache Lucene/Solr open source search project.

For more information on product and support options for LucidWorks Search, please write to: sales@lucidworks.com or visit our [website](#). Support inquiries can be submitted to our [Support group](#).



[LucidWorks](#)

3800 Bridge Parkway, Suite 101
Redwood City, CA 94065

Tel: 650.353.4057

Fax: 650.525.1365