

Developer Guide / Luigi

- 1. Luigi overview
- 2. Lucene overview
- 3. POJO to Document translation
 - 3.1. Example Spring configuration
- 4. Index configuration
- 5. The indexing service
- 6. The Searching Service
 - 6.1. Searching
 - 6.2. Background searcher

Luigi Javadoc

Talos authorisation

Back developer overview

TOP

1. Luigi overview

Luigi is a Java library that aims to provide Lucene's full-text search in a highly configurable and loosely coupled library.

Lucene is very easy to use, but very difficult to reuse. The application code that handles indexing is very tightly coupled with Lucene's code, and most of the indexing configuration ends up being hardcoded. Luigi provides a layer of services on top of Lucene that can be configured and managed through Spring beans.

The power of POJOs

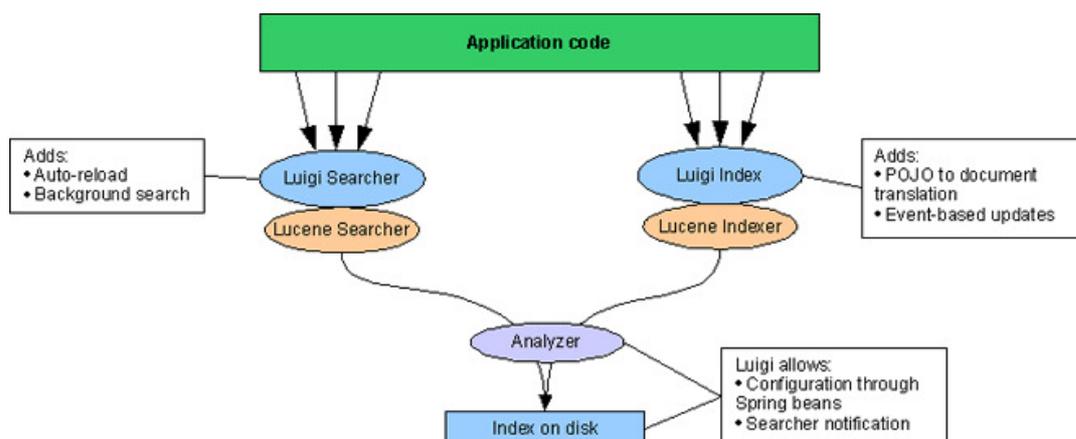
Luigi uses Plain Old Java Objects as input as well as output of searches, using descriptors provided at assembly time to transform the data. Assembly and configuration are done through Spring's bean definitions, and the code that matches your data objects to Lucene documents uses reflection to decouple one from the other completely. This provides a very high level of interoperability and extensibility.

Background indexing through the Connector API

We developed an event library to enable indexing in the background and concurrent searching. When your application changes data, it only needs to fire an event passing the correct object, and the index will be updated in the background. When the index finishes updating, it will notify all listening Searchers, which will hot-swap the Lucene searcher, so new searches already see the updated data, while ongoing searches (at the moment of update) finish their work cleanly.

2. Lucene overview

A search in Lucene is defined by three components: a data storage (usually one or more files on disk), the configuration for the indexing, and the configuration for searching. The indexer component uses an Analyzer, an object responsible for parsing and tokenizing the input text before indexing, and a document structure. It also needs a reference to the data storage. The searching component contains a reference to the storage as well, and also needs an Analyzer to handle the search parameters. Luigi concentrates on the most common case, when the indexer and the search use the same data storage and the same analyzer.



TOP

3. POJO to Document translation

The items that Lucene indexes for search are called *documents*, and are represented by the class `Document`. The document structure used by Lucene is composed of *fields*. Each field has a *name* and a text *value*, which can be provided by a `String` or `java.io.Reader` instance. All documents within an index must have the same structure, such as this:

- Title
- File name
- Author
- Content (the document's real data)
- Creation Date

Each Lucene index must be configured with rules that define what happens to each field. A field can be tokenized or not, saved or not, and indexed (which means "made available to search") or not. For more details, refer to Lucene's documentation.

Luigi expects Java objects as input, and uses the same rules for each object property as the rules for document fields in Lucene. A simple but complete translation from object graphs into documents is implemented by default, so that the following structure is also possible (`obj.prop` is equivalent to `obj.getProp()`):

- `doc.title`
- `doc.file.name`
- `doc.author.fullName`
- `doc.content`
- `doc.creationDate`

In the above case, the result of `doc.getAuthor()` can be any object for which a method `getFullName()` is defined. Reflection is used to access the property. As of Luigi 1.1, you can also use a `java.util.Map` on the path, and the `ReflectiveBuilder` will call `object.get("title")` when it sees `"object.title"`.

3.1. Example Spring configuration

This is a `Spring` bean definition taken from a real application (names were changed to protect the innocent):

```
<bean id="QuickSearchDescriptor" class="net.cw.luigi.spring.DefaultObjectDescriptorFactory">
  <property name="key">
    <value>key</value>
  </property>
  <property name="defaultField">
    <value>content</value>
  </property>
  <property name="text">
    <set>
      <value>attributes.name</value>
      <value>attributes.description</value>
      <value>attributes.shortdescription</value>
      <value>attributes.longdescription</value>
      <value>type</value>
    </set>
  </property>
  <property name="unindexed">
    <set>
      <value>id</value>
    </set>
  </property>
  <property name="unstored">
    <set>
      <value>content</value>
    </set>
  </property>
</bean>
```

This factory is the preferred way to define object descriptors. For more information, refer to the javadoc of the `DefaultObjectDescriptorFactory`, and the `Lucene` documentation.

4. Index configuration

Luigi defines an object called `IndexConfiguration`, which keeps a reference to the index storage and to the analyzer, and allows other objects to listen for changes in the storage device. The preferred way to configure an index is through the `IndexConfigurationFactory`.

```
<bean id="QuickSearchIndex-en" class="net.cw.luigi.spring.IndexConfigurationFactory" >
```

```

<property name="dispatcher">
  <bean class="net.cw.connector.event.MulticastDispatcher"/>
</property>
<property name="analyzer">
  <bean class="org.apache.lucene.analysis.standard.StandardAnalyzer"/>
</property>
<property name="location">
  <value>d:/search/quick_en</value>
</property>
</bean>

```

We'll not get into details about what this **dispatcher** means, but it has to do with automatic notification of searchers. Searchers will get notified when the index is changed on disk, so that new searches see the change.

5. The indexing service

If you have the information about your index (through the [IndexConfiguration](#)) and about the objects that will be given to your indexer (through the [ObjectDescriptor](#)), you can configure your indexer through the [DefaultIndexingService](#), which can in turn be created by the [IndexingServiceFactory](#):

```

<bean id="QuickIndex-en" class="net.cw.luigi.spring.IndexingServiceFactory">
  <property name="indexConfiguration" ref="QuickSearchIndex-en"/>
  <property name="objectDescriptor" ref="QuickSearchDescriptor"/>
</bean>

```

The [IndexingServiceFactory](#) has another property called *channel*. If you provide an [EventChannel](#), the Index can be notified of changes through the Connector API. More details on this later.

6. The Searching Service

Luigi also builds a layer over searching to provide automatic reloading of indexes – something that is a pain, for Lucene newbies and experts alike. We have also implemented a background searcher – which you probably won't need unless your indexes are *really big* and your queries take very long time.

A Searcher must know where to search and how to analyze query terms. This information is provided by the [IndexConfiguration](#) object. Details on how to build an index configuration are provided [above](#).

```

<bean id="CWShopSearch-en" class="net.cw.luigi.defaults.DefaultSearchingService">
  <constructor-arg type="net.cw.luigi.index.IndexConfiguration">
    <ref local="QuickSearchIndex-en"/>
  </constructor-arg>
  <constructor-arg type="net.cw.luigi.search.ObjectDescriptor">
    <ref bean="QuickSearchDescriptor"/>
  </constructor-arg>
</bean>

```

Note that the [DefaultSearchingService](#) is not a factory. When this bean is initialized, it will register itself as a listener to the [IndexConfiguration](#), so that it gets notified when the index changes. There's no extra configuration you must supply.

6.1. Searching

The Searching service bean will give you a [Searcher](#) object, which you will in your applications to do lucene searches. As of version 1.1, all searches are provided through a [SearchOptions](#) instance. The method with a query string was deprecated.

Usually, you will only search with a query string, like this:

```

SearchOptions opts = new SearchOptions();
opts.setText("attributes.name:john type:user");
ResultBuilder rbuilder = new DefaultResultBuilder();
List results = rbuilder.build(searchingService.getSearcher().search(opts));

```

The code above will search for records of type "user" containing "john" in the `attribute.name` field. The resulting hits will be converted to a list of Map instances, each one containing the fields declared for the document in the [ObjectDescriptor](#). Another example:

```

SearchOptions opts = new SearchOptions();
String[] sortFields = {"attributes.name", "attributes.description"};
opts.setSort(new Sort(sortFields));
opts.setText("john");
ResultBuilder rbuilder = new DefaultResultBuilder();
Iterator it = rbuilder.iterate(searchingService.getSearcher().search(opts));

```

```

while (it.hasNext()) {
    SearchResult sr = (SearchResult) it.next();
    Map data = sr.getAttributes();
    print("ID: " + data.get("id"));
    print(" (Relevance: " + (sr.getScore() * 100.0f) + "%)");
    print("; Name: " + data.get("attributes.name"));
    println("; description: " + data.get("attributes.description"));
}

```

The code above will search for records containing "john" in the `content` field – which was declared as the `defaultField` in the `ObjectDescriptor` [above](#). The Search options also indicate that you want the results sorted first by `attributes.name`, then by `attributes.description`. The `ResultBuilder` will not create one `Map` instance for each record, it will create one each time the iterator's `next()` method is called. This might give you a smaller memory footprint.

6.2. Background searcher

If you want a searcher which can do background search, you will need to give it a `CommandExecutor`:

```

<bean id="CWShopSearch-en" class="net.cw.luigi.defaults.DefaultSearchingService">
    <constructor-arg type="net.cw.luigi.index.IndexConfiguration">
        <ref local="QuickSearchIndex-en"/>
    </constructor-arg>
    <constructor-arg type="net.cw.luigi.search.ObjectDescriptor">
        <ref bean="QuickSearchDescriptor"/>
    </constructor-arg>
    <constructor-arg type="net.cw.luigi.commands.CommandExecutor">
        <bean class="net.cw.luigi.commands.CommandExecutorImpl" >
            <constructor-arg>
                <bean class="EDU.oswego.cs.dl.util.concurrent.QueuedExecutor"/>
            </constructor-arg>
        </bean>
    </constructor-arg>
</bean>

```

If you do need to run searches in the background, you might be interested in the progress of the issue [LG-7](#).



Please send us comments, questions, criticism!

[USER GUIDE](#) | [DEVELOPER GUIDE](#) | [DOWNLOADS](#) | [FAQ](#) | [PEOPLE](#) | [FORUM](#)

Non-commercial, non-derivative use of this documentation is permitted.

