SolrPlugins

Solr Plugins

Solr allows you to load custom code to perform a variety of tasks within Solr – from custom Request Handlers to process your searches, to custom Analyzers and Token Filters for your text field, even custom Field Types.

Looking for examples of Solr Plugins? Checkout the website solr.cool!

- Solr Plugins
- How to Load Plugins
  - The Old Way
- Classes that are 'Pluggable'
  - Request Processing
    - SolrRequestHandler
      - SearchComponent
      - QParserPlugin
      - ValueSourceParser
    - Highlighting
      - SolrFragmenter
      - SolrFormatter
    - UpdateRequestProcessorFactory
    - QueryResponseWriter
    - Similarity
    - CacheRegenerator
  - Fields
    - Analyzer
    - Tokenizer and TokenFilter
    - FieldType
  - Internals
    - SolrCache
    - SolrEventListener
    - UpdateHandler
- Building Plugins
- Plugin Initialization
  - ResourceLoaderAware
  - SolrCoreAware
  - Lifecycle

How to Load Plugins

Plugin code can be loaded into Solr by putting your classes into a JAR file, and then configuring Solr to know how to find them.

If you want to use multiple SolrCores, and have a plugin available to all of them, you can place your JAR files in a directory specified using the "sharedLib" attribute in your solr.xml file prior to starting your servlet container. In version 4.3.0 and earlier, you can set the sharedLib attribute to "lib" to load from $SOLR_HOME/lib ... but in 4.3.1 and later, if you want to use $SOLR_HOME/lib, you must NOT configure sharedLib. The lib directory in the solr home is automatically loaded in newer versions.

For loading plugins in individual SolrCores, you have two options:

1. Place your JARs in a `lib` directory in the instanceDir of your SolrCore. In the example program, the location is example/solr/lib. *This directory does not exist in the distribution*, so you would need to do `mkdir` for the first time.
2. Use the lib directive in your solrconfig.xml file to specify an arbitrary JAR path, directory of JAR files, or a directory plus regex that JAR file names must match. Do not load jars in this way that have already been loaded from $SOLR_HOME/lib or $INSTANCEDIR/lib.

Loading plugins uses a custom Class Loader. It has been tested with a variety of Servlet Containers, but given the multitudes of servlet containers available in the wild it may not always work with *every* servlet container.

The Old Way

Another method that works consistently on any servlet container is to:

1. unpack the solr.war
2. add a jar containing your custom classes to the `WEB-INF/lib` directory
3. repack your new, customized, solr.war and use it.

Classes that are 'Pluggable'

The following is a complete list of every API that can be treated as a plugin in Solr, with information on how to use that configure your Solr instance to use an instance of that class.

Request Processing

SolrRequestHandler

Instances of SolrRequestHandler define the logic that is executed for any request. Multiple handlers (including multiple instances of the same SolrRequestHandler class with different configurations) can be specified in your solrconfig.xml...

```
<requestHandler name="foo" class="my.package.CustomRequestHandler" />
<requestHandler name="bar" class="my.package.AnotherCustomRequestHandler" />
<requestHandler name="baz" class="my.package.AnotherCustomRequestHandler">
  <!-- initialization args may optionally be defined here -->
   <lst name="defaults">
     <int name="rows">10</int>
     <str name="fl">*</str>
     <str name="version">2.1</str>
   <lst>
   <int name="someConfigValue">42</int>
</requestHandler>
```

for more info, see: SolrRequestHandler

SearchComponent

⚠️ Solr1.3

Instances of SearchComponent define discrete units of logic that can be combined together and reused by Request Handlers that know about them. (in particular: SearchHandler)

for more info, see: SearchComponent

QParserPlugin

QParserPlugin can be used to define customized user query processing instances of QParser .

First, implement a subclass of QParserPlugin. This consists primarily of implementing the parse() method to construct the appropriate Query objects.

Next register it in solrconfig.xml like this:

```
<queryParser name="myqueryparser" class="my.package.MyQueryParserPlugin" />
```

See SolrQuerySyntax and it's list of useful parser implementations for examples of writing a !QParserPlugin.

Having done this, you can choose to use your query parser on a one-time basis using the defType query parameter, like this:

```
http://mysolrmachine:8983/solr/select/?defType=myqueryparser&q=hi
```

You can also specify your query parser as part of the q parameter, like this:

```
http://mysolrmachine:8983/solr/select/?&q={!myqueryparser}hi
```

For more permanent use, you will likely want to define a separate SolrRequestHandler for your parser, like this:

```
<requestHandler name="dismax" class="solr.SearchHandler" >
 <lst name="defaults">
  <str name="defType">myqueryparser</str>
  ...
```

ValueSourceParser

Use this to plugin your own custom functions see FunctionQuery.

register in solrconfig.xml directly under the <config> tag

```
<valueSourceParser name="myfunc" class="com.mycompany.MyValueSourceParser" />
```

The class must implement org.apache.solr.search.ValueSourceParser

Highlighting

⚠️ :TODO: ⚠️ NEED DOCS

SolrFragmenter

⚠️ :TODO: ⚠️ NEED DOCS

SolrFormatter

⚠️ :TODO: ⚠️ NEED DOCS

UpdateRequestProcessorFactory

⚠️ :TODO: ⚠️ NEED DOCS

See UpdateRequestProcessor

QueryResponseWriter

Instances of QueryResponseWriter define the formatting used to output the results of a request. Multiple writers (including multiple instances of the same QueryResponseWriter class with different configurations) can be specified in your solrconfig.xml...

```
<queryResponseWriter name="wow" class="my.package.CustomResponseWriter" />
<queryResponseWriter name="woz" class="my.package.AnotherCustomResponseWriter" />
<queryResponseWriter name="woz" class="my.package.AnotherCustomResponseWriter" >
  <!-- initialization args may optionally be defined here -->
  <int name="someConfigValue">42</int>
</queryResponseWriter>
```

Similarity

The Similarity class is a native Lucene concept that determines how much of the score calculations for the various types of queries are executed. For more information on how the methods in the Similarity class are used, consult the Lucene scoring documentation. If you wish to override the DefaultSimilarity provided by Lucene, you can specify your own subclass in your schema.xml...

```
<similarity class="my.package.CustomSimilarity"/>
```

CacheRegenerator

The CacheRegenerator API allows people who are writing custom SolrRequestHandlers which utilize custom User Caches to specify how those caches should be populated during autowarming. A regenerator class can be specified when the cache is declared in your solrconfig.xml...

```
  <cache name="myCustomCacheInstance"
    class="solr.LRUCache"
    size="4096"
    initialSize="1024"
    autowarmCount="1024"
    regenerator="my.package.CustomCacheRegenerator"
    />
```

Fields

Analyzer

The Analyzer class is a native Lucene concept that determines how tokens are produced from a piece of text. Solr allows Analyzers to be specified for each fieldtype in your schema.xml that uses the TextField class – and even allows for different Analyzers to be specified for indexing text as documents are added, and parsing text specified in a query...

```
<fieldtype name="text_foo" class="solr.TextField">
  <analyzer class="my.package.CustomAnalyzer"/>
</fieldType>
<fieldtype name="text_bar" class="solr.TextField">
  <analyzer type="index" class="my.package.CustomAnalyzerForIndexing"/>
  <analyzer type="query" class="my.package.CustomAnalyzerForQuering"/>
</fieldType>
```

Solr also provides a SolrAnalyzer base class which can be used if you want to write your own Analyzer and configure the "positionIncrementGap" in your schema.xml...

```
<fieldtype name="text_baz" class="solr.TextField" positionIncrementGap="100">
  <analyzer class="my.package.CustomSolrAnalyzer" />
</fieldType>
```

Specifying an Analyzer class in your schema.xml makes a lot of sense if you already have an existing Analyzer you wish to use as is, but if you are planning to write Analysis code from scratch that you would like to use in Solr, you should keep reading the following sections...

Tokenizer and TokenFilter

In addition to specifing specific Analyzer classes, Solr can construct Analyzers on the fly for each field type using a Tokenizer and any number of TokenFilters. To take advantage of this functionality with any Tokenizers or TokenFilters you may have (or may want to implement) you'll need to provide a TokenizerFactory and TokenFilterFactory which takes care of any initialization and configuration, and specify these Factories in your schema.xml...

```
<fieldtype name="text_zop" class="solr.TextField" positionIncrementGap="100">
  <analyzer>
      <tokenizer class="my.package.CustomTokenizerFactory"/>
      <!-- this TokenFilterFactory has custom options -->
      <filter class="my.package.CustomTokenFilter" optA="yes" optB="maybe" optC="42.5"/>
      <!-- Solr has many existing FilterFactories that you can reuse -->
      <filter class="solr.StopFilterFactory" ignoreCase="true"/>
  </analyzer>
</fieldtype>
```

FieldType

If you have very specialized data type needs, you can specify your own FieldType class for each `<fieldtype>` you declare in your schema.xml, to control how the values for those fields are encoded in your index...

```
<fieldtype name="wacko" class="my.package.CustomFieldType" />
```

Internals

SolrCache

The SolrCache API allows you to specify custom Cache implementations for any of the various caches you might declare in your solrconfig.xml...

```
<filterCache       class="my.package.CustomCache"          size="512" />
<queryResultsCache class="my.package.CustomCache"          size="512"  />
<documentCache     class="my.package.AlternateCustomCache" size="512"  />
```

SolrEventListener

Instances of the SolrEventListener Interface can be configured in your solrconfig.xml to be be executed any time specific events occur within Solr.

`firstSearcher` and `newSearcher` events trigger the `newSearcher()` method with the appropriate args, `postCommit` and `postOptimize` events will trigger the `postCommit()` method...

```
<listener event="newSearcher" class="my.package.CustomEventListener">
  <-- init args for the EventListener instance can be specified here -->
  <lst name="arg1">
    <str name="q">solr</str> <str name="start">0</str> <str name="rows">10</str>
  </lst>
  <int name="otherArg">42</int>
</listener>
```

UpdateHandler

The UpdateHandler API allows you to specify a custom algorithm for determining how sequences of adds and deletes are processed by Solr. The UpdateHandler you wish to use can be configured in your solrconfig.xml, but implementing a new UpdateHandler is considered **extremely** advanced and is not recommended....

```
<updateHandler class="my.package.CustomUpdateHandler">
```

Building Plugins

To develop your own plugins, add the `apache-solr-*.jar` jar files to the classpath you use to compile your code. They contains all of the Solr Interfaces and Class files you may need. If you are developing plugins that know about lower level Lucene interfaces, you may need to also include the `lucene-*.jar` jar files from the `lib/` directory of your Solr distribution.

Plugin Initialization

⚠️ Solr1.3

Plugins are initialized either with init(Map%3CString,String%3E args) or init(NamedList args).

⚠️ Solr1.4

Plugins can also be initialized with init(PluginInfo info). These Plugins can have children (any Subnode with a "class" attribute is considered a child). If a pluginInfoInitialized Plugin wants to initialise these children in its `inform( core )` method via

```
core.create.createInitInstance()
```

, the children must not be SolrCoreAware. see PluginInfo.
⚠️ Solr3.1 As of Solr3.1 any subnode which is not a "lst","str","int","bool","arr","float" or "double" is considered a child. (Yes "long" is missing - see SOLR-2541)

ResourceLoaderAware

Classes that need to know about the ResourceLoader can implement ResourceLoaderAware. Valid ResourceLoaderAware include:

- TokenFilterFactory
- TokenizerFactory
- FieldType

SolrCoreAware

Classes that need to know about the [SolrCore] can implement SolrCoreAware. Classes implementing SolrCoreAware must not have a dedicated Constructor. Valid SolrCoreAware classes include:

- SolrRequestHandler
- QueryResponseWriter
- SearchComponent

Lifecycle

The initialization lifecycle is:

1. Constructor
2. `init( Map / NamedList / PluginInfo )`
3. ResourceLoaderAware classes call: `inform( ResourceLoader );`
4. Before the first request is made and after all plugins have been created and registered, SolrCoreAware plugins call: `inform( SolrCore );`