

Nemoy Search Module for Lasso

Inspired by the first Lasso Programming Challenge¹, this little tool is an easy to implement, yet flexible system for adding single-field searches to your web site. A single input field accepts keywords with modifiers to search multiple fields of a single data table at the same time, then perform a relevance ranking algorithm of the found records.

Relevance ranking can be mathematically intensive, so a dynamic language like Lasso isn't going to provide the greatest performance when using advanced algorithms. However, by using some "good enough" adjustments to reduce the computational demand, we can create some relevance ranking with some user adjustability that will provide useful results for a variety of applications.

Meeting the "Challenge"

Nemoy meets the following Challenge goals:

- Find records which contain any keyword
- Find records which contain all required keywords prefixed by +
- Find records which contain keywords except those excluded by - prefixes
- Find records which contain all words together in a phrase wrapped by quotes
- Find records which contain any combination of the above terms
- Calculate relevance for found records
- Display records sorted by relevance

It also meets these additional goals:

- Find records contain "begins with" keywords
- Find records contain "ends with" keywords
- Find records based on searching multiple fields per record
- Allow weighting of fields for relevance
- Make use of lemmatization*, and weighting of lemmatized terms
- Abstract query interface for multiple database engines
- Abstract query interface for multiple data table sources
- Abstract relevance interface for multiple ranking formulas
- Highlight search terms in results

** current lemma dictionary does not distinguish parts of speech, and word list being used is derived from the contents of the entire ldml8 reference table*

¹ <http://www.lassosoft.com/Community/Challenge/index.lasso?9269>

Source Code Organization

Processing begins with `nemoy.lasso` which is a typical HTML template with Lasso code. The first significant task is to include `_nemoy/setup/_nemoyStartup.lgc`. The path to this file is hard coded in the `nemoy.lasso` demo file as it assumes no other setup files have been loaded yet to declare a variable for the path. (Inside `_nemoyStartup.lgc` paths are defined for all other source file locations.)

All resource code files are in the `/_nemoy/` folder which groups resources in the following sub folders:

- `/adaptors/` — each file is a custom type designed to accept query parameters and write an SQL statement for a specific database server. The only task for an adaptor is to construct the SQL statement itself, not perform the query. The adaptor is called by the ctype `fwpSrch_nemoyQuery` found in `/core/`.
- `/core/` — these are core logic files which remain constant to all `nemoy` deployments (unless the PageBlocks framework is being used, in which case, the core files are already loaded with the PageBlocks API libraries).
- `/linguistics/` — language related resource files for advanced searches. Example: `lemmata` includes words which are related by a root (lemma) that could be considered when searching.
- `/rankEngines/` — each file here is an implementation of a ranking algorithm within a single custom type. The `Nemoy Search Module` accepts multiple algorithms so it can apply uniquely tuned ranking formulas to unique data sets and purposes.
- `/setup/` — these are either standard files that need edited for the application, or are application-specific files to define searches (specific combinations of tables and fields to be searched along with some algorithm variables)
- `/style/` — these are misc GUI files for the demo files.

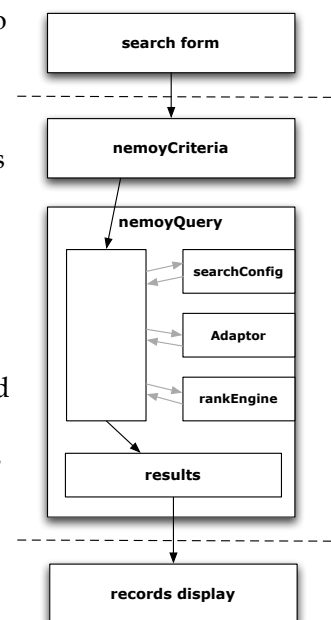
Source Code Design

The `Nemoy Search Module` code is object oriented except for the demo page startup process and a few utility custom tags. When a search is submitted, a criteria object is created which contains the search string, a map of parsed search terms, and a couple states of the terms as a collection. That criteria is passed to a query object which performs the task of searching, ranking the results, and containing the final result dataset.

The query object uses composition to integrate a search configuration object, a database adaptor object, and a rank engine object. Each of these are written to a standardized API.

A search configuration object defines the database, table, fields and other characteristics of a search. Each configuration is named. This search configuration name becomes part of the search criteria which is passed to the query object.

Likewise, the search ranking algorithm to be used for a search is a part of the search criteria object. It is independent of the search configuration.



The query object uses Lasso's `lasso_datasourceModuleName` tag to determine which database engine is being used to serve the database declared in a search configuration. The query object uses this to instantiate the appropriate database adaptor as a helper to create an SQL string to execute. Additionally, the query object instantiates a rank engine object to subsequently calculate a relevance value for each record in the found data set. The query object then sorts the results according to the relevance number.

A display object is created to generate HTML code for the resulting record data. Like the search form HTML, this piece is not integral to the nemoy engine, and can be replaced by application-specific code (although the demo code provides a generic, reusable routine for this functionality).

Search Configuration with `/setup/srchConfig_xxxx`.

A search configuration custom type declares settings for a search. This custom type is just a collection of values, there are no member tags. The example below identifies the required instance variables.

```
define_type: 'srchConfig_lassoTagRefc';

  local:
    'qryUser'      = '',
    'qryPswd'     = '',
    'db'           = 'ldml8_reference',
    'tbl'          = 'tags',
    'keyfield'     = 'id',
    'selectFields' = 'id, tag_name, tag_category, tag_type, tag_description',
    'requiredPhrase' = "tag_basic='Y' AND tag_display='Y'",
    'contentFields' = 'tag_name, tag_category, tag_description',

    'contentWeights' = (map:
      'tag_name'      = 5,
      'tag_category'  = 2,
      'tag_description' = 1),

    'lemmaWeight' = .6,

    'displayColumns' = (array:
      'tag_name'      = 'Tag',
      'tag_category'  = 'Category',
      'tag_type'      = 'Type',
      'tag_description' = 'Description'),

    'highlightColumns' = (array: true, true, false, true);

/define_type;
```

Each instance variable is described below:

- `qryUser` — the `-username` value to use for the query inline
- `qryPswd` — the `-password` value to use for the query inline
- `db` — the database name
- `tbl` — the table name
- `keyfield` — the name of the primary key field

- `selectFields` — a comma separated list of fields to be returned for display or other application logic purposes
- `contentFields` — a comma separated list of fields to be searched for content
- `requiredPhrase` — an SQL WHERE clause fragment that is to be included to qualify which records can be searched. For example, if only records where `rcrdApproved='Y'` can be searched, define that phrase here. The instance var must be defined, but a value is not necessary.
- `contentWeights` — for each field name in `contentFields`, that field must be listed here along with a value which identifies the relative value/importance of finding text in that field. The size of the numbers is irrelevant. The relative difference is what matters. If finding text in the field `tag_name` is 5x more relevant/important than finding that same text in the field `tag_description`, use an integer for `tag_name` that is 5x bigger than for `tag_description`.
- `lemmaWeight` — defines the relative weighting of a lemmatized term compared to a user entered term. Should be a decimal between 0 and 1.
- `contentColumns` — for each field from `selectFields` to be displayed on screen, list that field here along with a display title for that column. the data is an array, not a map, and the fields listed must be in the same order as listed in `selectFields` (though fields can be skipped).
- `highlightColumns` — for each field from `selectFields` to be displayed on screen declare a value of true or false to indicate whether text in that display should include highlighting of the search terms.

/setup/srch_querySetup

This simple custom type declares application-wide settings for the Nemoy Search Module. All searchConfigs that Nemoy will use are declared here. Additionally, the available database adaptors and rank engines are declared.

/setup/_nemoySetup

This file has three sections to it, and could be implemented in separate files depending on the application code organization.

The first section declares some path variables which are used throughout the other source code files. Technically for better OOP design, a object with these values would be better, but page vars for paths is a typical Lasso idiom.

Following the path definitions, the required libraries are loaded. There are some files which are native to the Nemoy Search Module, and some which are utility tags pulled from the PageBlocks framework (which will work fine anywhere). One utility used throughout the code is the `fwUtil_timer` which makes it very easy to add timers to code and then extract the results. This is embedded through out the code to aid in tracking the performance of searches and ranking engines.

The second section declares the main objects: `searchCriteria`, `searchQuery`, and `resultsTable`. these vars can be renamed in application-specific code as desired. The default values for `searchCriteria` will be application dependent.

The final section is a typical handler for search submissions. The way the demo files are written, searches can be submitted by clicking a form or through a GET directly from a URL. The

three-step process of doing criteria ->init, a query ->init, and then a ->searchFor on the query object is the standard way to use the search module.

/setup/srch_resultsTable

This is simply an application-specific way to draw the table for the demo files. Use it as an example for creating a table display object, or replace it however you prefer.

Rank Engines

A rank engine custom type has a simple structure. The file name and the class name must be the same, and there must be one method (member tag) called ->calcRelevanceOf. Implementing the algorithm will be specific to the formula, but there's some things which are common to all algorithms.

The data available to the calculator include the following:

- #records — all records returned by the search. The data structure is an array of pairs where the first value of the pair is the rank of the record. This starts out as zero, and is then updated by the rank engine. The second value of the pair, is the data for the record which is a map of field names and contents.

```
array:
  pair: rank = map:
    fieldName = fieldContent
    ...
    fieldName = fieldContent
  ...
  pair: rank = map:
    fieldName = fieldContent
    ...
    fieldName = fieldContent
```

- #allCount — the total number of records in the data table
- #searchWords — an array of individual words in the search string (they'll be stripped of term modifiers like quotes and + and such, but will be in the same order that they were originally typed)
- #contentFields — a comma separated list of fields that were searched (copy this value or weird things will happen if you change it).
- #contentWeights — the weighting factors of each of the content fields

It is possible that as more complex ranking formulas are considered that the interface between the criteria, query, and rank engines will have to evolve. The existing code should probably be updated to pass all these values as a map so that implementation is more abstracted.

The file rankEngine_wordCount.ctyp is an example of a minimal formula that simply counts the number of terms. It shows the typical minimum processing that a rank engine is likely to do in iterating through records, comparing field data to search terms, and updating the rank value of the record. The other rank engines can be reviewed for more complex formulas.

Database Adaptors

The database adaptors generate SQL query strings. While it's possible to build an array of search terms for Lasso's inlines, direct SQL is preferred for control, and it's possible that rankEngine-specific adaptors could be used to implement more complex (and faster) rank formulas.

Each adaptor must implement two methods `->buildQuery` and `->buildAllCountQuery`. The former would build and return the search query, while the latter would build and return a simpler query to determine an "all records" count (accounting for the searchConfig requiredPhrase).

Generally speaking, the code in each adaptor should be nearly identical except to replace the text that would generate the query string itself. We're pretty much constructing simple queries with some moderate boolean logic in the WHERE clause. This logic is tightly integrated into the supported search term modifiers (+, -, quotes etc). The patterns of the boolean logic should not be changed when creating adaptors for datasource that don't have one yet.

As it stands now, the adaptors for MySQL and Sqlite are actually the same. If this turns out to be true for additional adaptors, then the need to have separate ones probably goes away. For now, keeping them separate helps define and test the abstraction layers for adaptors.

Search Criteria

The `fwpSrch_nemoyCriteria` file acquires and parses search parameters. Inputs include the searchString typed by the user, a rankEngine value, and a srchConfig value. The latter two allow for a UI which gives the user a choice of multiple options, or they can be hard coded.

The `->init` method loads values from `client_getParams`. (Better OO code would pass `client_getParams` to the init method, but `client_getParams` is such a common Lasso idiom it makes sense to relieve the programmer of that task.)

The `->onCreate` method can optionally be used to declare default values for rankEngine and srchConfig using `-defaultSrchConfig` and `-defaultRankEngine` if none are passed via `client_getParams`.

The internal `->parseTerms` method will parse the searchString into components. It generates two data structures: one for search terms (search string words including their modifiers), and one for search words (search string words stripped of modifiers).

`searchTerms` is primarily for use by the query object for constructing the SQL query string. Whereas, `searchWords` is primarily for use by the rank engine to compare search words to the field contents, and for the display code to highlight search words in the displayed content.

Search Query

The `fwpSrch_nemoyQuery.ctyp` file is the heart of the logic for the search process. It uses composition to generate helper objects for the srchConfig data, the database adaptor, and the rank engine. It does this by looking up values from `srch_querySetup.ctyp` to identify source code files for these objects. They are loaded as raw source code then invoked into custom types dynamically.

The `->searchFor` method invokes the database adaptor to create the SQL query strings, initiates the searches, uses the internal `->makeRowMaps` method to convert the standard Lasso

recordsarray data into the data structure required for the rank engine, and finally calls the rank engine to calculate the relevance value for each row in the dataset.