

SEARCH SubSystem

Table of Contents

1. [Information](#)
2. [Internals](#)

ADEI has a modular search subsystem. The search capabilities are provided by the search engines which providing one-or-more search modules. Besides the search term, the search string may specify the search modules to perform search and number of limits to filter results. The module parameters could be specified along with the modules.

The search subsystem is implemented around three main classes:

- *SEARCHEngine* - Search engine providing one-or-more search modules, defined in *classes/searchengine.php*
- *SEARCHFilter* - Search filter providing an interface to restrict search results (like google's *site:kernel.org*), defined in *classes/searchfilter.php*
- *SEARCHResults* - Provides the results of search

The user supplies to the search subsystem:

- A list of search modules to perform search along with parameters
- Global search options
- Search string
- A set of limits to restrict search results

For each specified module, the search subsystem identifies a search engine providing this module and executes the *Search* function of the engine. The function returns a *SEARCHResults* with results or *false* if nothing is found. Finally, the search subsystem merges the results from individual modules and returns the merged *SEARCHResults* object or false if no module provided results.

The *SEARCHResults* object able to store the results of two different types (simultaneously):

- Standard results: each result item is described by the associative array. The following members are defined
 - ◆ *title* - short title describing the result item
 - ◆ *description* - the longer description of the result item, an HTML content is allowed
 - ◆ *rating* - the value between 0 and 1 indicating the result quality, the higher value the better match is
 - ◆ *props* - the associative array with standard ADEI properties describing the item. For example, for channel searches the props array will contain *db_server*, *db_name*, *db_group*, and *db_mask* properties. For interval searches, it would be just property *window*
 - ◆ *certain* - this option indicates what the search module is completely certain what it is this record what the user is actually looking for
 - ◆ Arbitrary number of other properties which are used by the search engine internally (for record matching, for example)

Just an example, of associative array describing a found time interval:

```
array(  
  'title' => 'January 2005',  
  'props' => array(  

```

```

        'window' => "1104537600-1107216000"
    ),
    'description' => false,
    'certain' => true
)

```

- Custom results: The search engine is mainly used to provide results in ADEI web display. In some cases the results are provided by 3rd party applications which doesn't respect the ADEI way of structuring results, but provide just an HTML page with all results in ready to display form. In order to support such third party application, ADEI search module may return a *SEARCHResults* module with custom results. In this case, instead of per-item associative array, the *SEARCHResults* will store the XHTML content representing all results provided by the search module.

The merged results of multiple search module may contain both per-item associative arrays for results of some modules and XHTML results for others.

Default Implementation

The *Search* function of search engine may implement the search in arbitrary way. However, there is a standard procedure defined for search of textual data. It intended to standardize handling of search modifiers and simplify coding of new search engines for many standard cases. The default implementation of engine *Search* function is using following procedure:

- *GetList* function is called to get complete associative list of available elements. In this list the key is an element identifier and value contains the presented above associative array describing the item. Besides the described standard properties, this array should contain the properties used for matching. The default implementation expects *name* and optionally *uid* properties to be matched against search string. And of course, the *rating* and *certain* fields are not set yet. So, we have the following list of properties:
 - ◆ *title* - title
 - ◆ *description* - description
 - ◆ *props* - ADEI properties
 - ◆ *uid* - record unique identifier if any (used for matching)
 - ◆ *name* - record short name (used for matching)
 - ◆ arbitrary number of other properties used by custom matching functions
- Each item of the received string is matched against search terms using *CheckString* function. This function returns the match rating (*0* means there is no match)
- The items for which the non-zero rating is returned are checked against filters and added to the search results
- Finally, the duplicating results are filtered using *Accept* function of *SEARCHResults* object. The comparisons between results are carried out using compare function returned by *GetCmpFunction* function of search engine. The default function just compares *title* members of the associative arrays describing compared items.

The default *CheckString* function is working in following way:

- The search string is split in phrases and for each phrase *CheckPhrase* function is called (see [adeiSEARCH/String](#) for splitting algorithm)
- Depending on the used module, the *CheckPhrase* function is expected to construct from the associative array a string value to match against search terms. The default behavior is to use *name* member of associative array. If the *name* member is non-existing, the match succeeds.

- The constructed string value is passed to the *CheckTitlePhrase* function.
- *CheckTitlePhrase* function checks if passed string is fitting to the current search phrase and returns the rating.
- Finally the rating computed for all search phrases are reconciled in overall rating using rules described in the [adeiSEARCH/String](#)

The matching is performed in one of 4 supported modes depending on the match modifiers and global options specified in the search string (see [adeiSEARCH/String](#))

- *standard match* - The beginning of any word should match search phrase. The **word sinus cosinusfff** matches the phrase **sinus cosinus**, but **xsinus cosinus** - not.
- *word match* - The words should match completely. The **word sinus cosinus fff** matches, and **sinus cosinusfff** - not.
- *fuzzy match* - The words boundaries are not important and even **xsinus cosinusx** matches the **sinus cosinus** search phrase.
- *regex match* - In this mode the search phrase considered regular expression and this regular expression is matched against passed string

Search Engine

So, the search modules are provided by classes implementing the [SEARCHEngine?](#) interface. Each class could provide one or more search module. To implement a new search engine the following actions should be taken:

- The class implementing *!SEARCHEngine* interface and extending *!SEARCHEngine* base class should be implemented
- This class should provide a list of supported modules in the *modules* member of class. It is associative array where the key is module id and the value is module title.
- It should define either custom *Search* function or provide *GetList* function to be used in conjunction with the default approach described above.
- The implemented class should be stored under *classes/search* directory (the file name should be lower-cased class name with *.php* extension).
- The search engine should be enabled in the configuration. A new element should be appended into the `$SEARCH_ENGINE` associative array. The key is a class name and the value is default initialization parameters.

Standard Engine

GetList function should return an array containing a descriptions of all all available items (through which we would search). This description is an associative array described in sections above. It should contain the following members:

- *title* - the title used to describe record in the search results
- *description* - the longer description of the record, HTML content is allowed
- *props* - the associative array with standard ADEI properties describing the record
- *name* or arbitrary number of other properties used by the search engine for record matching

For example, the following array could be returned by the *GetList* function:

```

array(
  array(
    'title' => 'January 2005',
    'props' => array(
      'window' => "1104537600-1107216000"
    ),
    'description' => false,
    'certain' => true
  ),
  ...
)

```

Of course, all other functions described in the *Default Implementation* section could be overridden as well. For example, if the engine is intended to support more than one search module, it needs to override *CheckPhrase* function. The default implementation constructs the match string just by taking *name* member of associative array. This string is, then, matched against search phrases. However, if multiple modules are used, for each module the algorithm for construction of match string should be specified. The *CheckPhrase* function in this case should construct a match string depending on the search module specified and pass it to the default *CheckTitlePhrase* function (or perform string matching by itself).

Custom Engine

The custom search engine needs to provide *!Search* function. It should create the *!SEARCHResults* object and fill it with results using *!Append* function call. Then return the object or *false* if nothing is found. Just a simple example:

```

function Search($search_string, $module, SEARCHFilter $filter = NULL, $opts = false) {
  $res = new SEARCHResults($filter, $this, $module);
  $res->Append(array(
    'title' => 'January 2005',
    'props' => array(
      'window' => "1104537600-1107216000"
    ),
    'description' => false
  )
  );
  if ($res->HaveResults()) return $res;
  return false
}

```

The special search engines intended to return custom XHTML content should use following approach instead (the XHTML strings are passed to the *!Append* function instead of the associative arrays describing found items):

```

$result = new SEARCHResults(NULL, $this, $module, "");
$result->Append("<XHTML content>");
return $result;

```

The `<?xml?>` should not be included into the content.

Search Filters

The filters are used to reject part of the search results as well as to add/modify information associated with found items. The filters are specified at the search string as follows:

```

interval:June 2005

```

If such filter is found, the *!INTERVALSearchFilter* object (defined in the *classes/search/intervalfilter.php*) is constructed. This object will get the filter value (*June 2005*) as a single parameter to its constructor. And it should implement a single function: *FilterResult* which should return *true* if the current record should be filtered out or *false* otherwise. The *FilterResult* receives two parameters:

- associative array with associative array describing the current item
- current rating of match

Both these parameters can be altered by *FilterResult* function.

Example. Lets consider standard *item* search used in conjunction with *interval* filter. The search will provide multiple records describing found item (i.e. the associative array with information will contain standard properties: *db_server*, *db_name*, *db_group*, and *db_mask*). The *interval* filter is intended to limit the display interval. Therefore, when the *FilterResult* function is called, it will add the *window* property to the associative array limiting display window to *June 2005*.

If multiple filters are specified they are executed sequentially until any filter will not reject the current item by returning *true* from *FilterResult* function. To implement a new filter it is necessary:

- Choose a not used name, for example: *site*
- Implement *SITESearchFilter* class extending *BASESearchFilter* (*site* is capitalized for class name)
- Implement *FilterResult(&\$info, &\$rating)* function which returns *true* if the value should be filtered out or *false* otherwise. The filter value is accessible using *\$this->value*.
- Place the implemented class in the *classes/search/sitefilter.php* (the lowercase filter name is used for the file name)

Format of search string

The search string consists of four components:

- The first component defines type of the search. Examples are *item search*, *channel value search*, *datetime search*.
- Second component provides some options. For example, demands exact or fuzzy match
- Third and fourth components are type-dependent and containing search string and additional limits

I: The format is as follows:

```
[type/module specification] [global flags] <search string>
[limits]
```

Everything besides search string is optional. By default if the type is not specified, the search string is analyzed. Analysis routine guesses the type of search and executes a default set of modules for this type. The default behavior is to search for channel and group names. See [String Analysis](#) section for details.

The search type is specified in the curly brackets in the beginning of the search string. The search module available in the *classes/search* should be indicated (name of the class should be specified). Optional options for the class constructor could be indicated as well. If multiple modules are specified, the multiple searches are performed sequentially. The following format is expected:

```
{module_name(opt1=value1,opt=value2), another_module(...) }
```

II: The global options are going next to the search type and specified in the square brackets. This options then passed to the search modules with the search string and handled by the module code. The following options are supported:

- = - Exact match, this means what the search string is matched completely without splitting into the phrases
- *w* - Word match, if not overridden by match modifiers, see below
- ~ - Fuzzy match, if not overridden by match modifiers, see below

III: Then the search string is follows. If the *Exact match* flag is not specified, it consists of the phrases. The phrase is

- words consisting of alphanumeric symbols, dash and underscore symbols (-,_)
- multiple words enclosed in singular or double quotes("")
- regular expressions enclosed in / from both ends

This is an example of a search string consisting of 4 components: two words, one phrase, and a regular expression:
{ { {word1 word2 "phrase 3" /regexp/ } } }

Before each phrase, a match modifier could be specified. The following match modifiers are supported

- if no modifier is specified, the phrases starting from search term will be matched
- = - full match, the whole words are matched
- ~ - fuzzy match, any part of a word could be matched

Please consider following example to understand the meaning of match modifiers. By default if a search for **sin** is performed the words **sin** and **sinus** will be matched, but *cosinus* - not. However, if a fuzzy search is given (~**sin**), the **cosinus** will be matched as well. On other hand if a full match is required (=sin), only **sin** will be matched. Both **sinus** and **cosinus** will be rejected.

The match of each phrase against data records produces ratings from 0 to 1 indicating match quality. The value 0 means what the record is not matched and value 1 indicates a full match. If several phrases are listed in search string, the ratings of each phrase match are multiplied to produce overall rating. For example, if phrase1 matched with rating 0.70, phrase2 matched with rating 0.30 and word3 is fully matched, the overall rating would be: $0.70 * 0.30 * 1$.

Rating computation could be altered using unary and binary operations. Lets assume what *[word]* is a rating of *word*, then the ratings of these operations are computed as follows:

- ! *word* - The resulting rating would be $1 - [word]$
- + *word* - The rating below 1 will be cut to 0
- - *word* - All non-zero ratings will be cut to zero, and zero rating will be replaced with 1
- (*word1|word2*) - The maximal rating amongst $[word1]$ and $[word2]$

Few examples of complex search strings:

```
=sinus | cos1
```

```
!"a b c" ~d -e +('f g' !(i (k))) "m n"
```

IV: On-or-more limits can be set in the last part of the search string. The following format is expected

Format of search string

```
limit_name:limit_value another_limit:another_limit_value
```

The limits handling is completely module specific. Example:

```
+sinus | cos1 interval:2006
```

Implemented Engines

The following engines are implemented in the current version.

INTERVALSearch Engine

Provided Modules:

- *interval* - Tries to parse the time interval from textual representation given in search string. The only property *window* is returned with interval of UNIX timestmaps.

Supported Filters:

- *interval* - allows to find intersection of two intervals

ITEMSearch Engine

Provided Modules:

- *channel* - Searches items by uid only
- *item* - Searches items by uid and name
- *group* - Searches groups by name
- *mask* - Searches masks by name
- *control* - Searches controls by uid only
- *control_item* - Searches controls by uid and name
- *control_group* - Searches control groups by name

Supported Filters:

- *interval* - adds window property to the items specification

PROXYSearch Engine

Provided Modules:

- *proxy* - downloads XML document from the specified location and applying XSLT stylesheet to convert it into the XHTML. Accepts several parameters:
 - ◆ *xml* - the service to obtain XML document from (mandatory)
 - ◆ *xslt* - the stylesheet to apply to XML, could be omitted if the service returns XHTML directly
 - ◆ *noprops* - instructs ADEI to not add current properties when calling the service, otherwise the passed *db_server*, *db_name*, and other properties would be added in the end of service request.

Supported Filters:

Implemented Engines

- *interval* - adds *window* property to the XML service request

Example usage:

```
proxy(xml=katrin.php?target=runs;xslt=katrinsearch;noprops) } interval:1218431322-1253472677
```

String Analysis

If the search modules are not specified, the auto-detection is used to select appropriate modules. The search string is sequentially given to all search engines as they are listed in the configuration. The engines are supposed to analyze the string. If the format of string fits the expectations of the engine (for example, the interval search checks if there is month names, short or full, within the string) it claims the string by returning the set of modules (along with parameters) which will be used for search. Otherwise *false* is returned. If the search string is not claimed by any engine, the channel search will be performed.

If the string is claimed by a search engine no further detection is performed and string is handled by modules specified by the engines. Therefore, the engines listed first in the configuration had more chances to claim a new string. This should be considered for the ordering of the engines in the configuration.

The string analysis is performed by the *DetectModules* function. It accepts a search string as an only argument. The result should be an associative array where the keys are module names and the values are module parameters. The base implementation of SEARCHEngine claims no strings.