****

Zentity (Version 1.0)

Search User Guide

May 2009

Contents

[Introduction 3](#_Toc229296638)

[Overview 3](#_Toc229296639)

[Scenarios/Input Syntax 3](#_Toc229296640)

[Search by Value 3](#_Toc229296641)

[Property Based Search 4](#_Toc229296642)

[Search Using Special Tokens 5](#_Toc229296643)

[Search Using Predicates 6](#_Toc229296644)

[Search Operators 9](#_Toc229296645)

[Types of Literals 12](#_Toc229296646)

[Grouping 15](#_Toc229296647)

[ResourceType Token 15](#_Toc229296648)

[Resource Type Inference 16](#_Toc229296649)

[Excluded Resource Types 17](#_Toc229296650)

[Excluded Predicates 17](#_Toc229296651)

[Token Priorities 18](#_Toc229296652)

[Content search 18](#_Toc229296653)

[Application Configuration Section 19](#_Toc229296654)

[Implicit Fields 19](#_Toc229296655)

[Predicates Tokens 20](#_Toc229296656)

[Special Tokens 21](#_Toc229296657)

[Excluded Predicates 21](#_Toc229296658)

[Excluded Resource Types 22](#_Toc229296659)

[Known Issues 23](#_Toc229296660)

[Appendix 23](#_Toc229296661)

[Tokens 23](#_Toc229296662)

# Introduction

This document explains how to use Zentity Search[[1]](#footnote-2).

## Overview

Search provides a user with capabilities to retrieve resource metadata and content from the repository using keyword search.

Users can supply basic keywords and Zentity Search API searches the metadata properties along with optional Content search.

Search enables a user to query repository in a format familiar to him through Windows Desktop Search (WDS)’s Advanced Query Syntax (AQS). Keeping the query syntax similar will help in a minimal learning curve for a user who is already familiar with AQS. There would be a learning curve for a new user, which will be minimized with the intuitiveness of the query syntax.

# Scenarios/Input Syntax

This section describes the use cases and search input syntax from an end-user perspective.

### Search by Value

When a user searches for some string without specifying a [token](#_Tokens), we term that search string as an implicit search (e.g. a simple keyword query). This means that the user doesn’t know which property contains that value and wants to search for possible matches.

A single word is considered as a string. If more than one word is supplied in the query string, the query would be interpreted as an *AND* of all the words on the implicit columns. To override this behavior and find for a group of words in sequence (multi-word strings) user will have to group those words within delimiters. Delimiters can be either single or double quotes.

Examples:

1. *Design patterns* - search for resources with “Design” AND “patterns” in the implicit fields of resource (default would be just *Title* and *Description*. The defaults are configurable through an XML file.)
2. *“Design patterns”* - search for all resources having the complete string “Design patterns” in the implicit fields.
3. *‘Design patterns’ ResourceType:Book* - search for book resources with ‘Design pattern’ in their implicit fields.

The default implicit fields and their resource types are as follows.

|  |  |  |
| --- | --- | --- |
| **Resource Namespace** | **Resource Type** | **Resource Properties** |
| Zentity.Core | Resource | 1. Title 2. Description |
| Zentity.ScholarlyWorks | Tag | 1. Name |
| Contact | 1. Email |
| ScholarlyWork | 1. Notes 2. Abstract |
| PersonalCommunication | 1. From 2. To |
| Code | 1. ProgrammingLanguage 2. Technology |
| Person | 1. LastName 2. FirstName 3. MiddleName |
| Lecture | 1. Series 2. Venue |
| Experiment | 1. Name |
| Publication | 1. BookTitle 2. Publisher |
| Email | 1. Subject |
| Video | 1. Director |
| Journal | 1. JournalName |
| Proceedings | 1. EventName |

Based on the above default configuration,

1. If the user enters a search string such as *physics*, the Search assumes the resource type as base *Resource* and searches for the word “design” in properties *Title* and *Description* only.
2. If the user enters a search string such as *physics resourcetype:tag*, the Search takes the resource type as *Tag* and searches for the word “design” in properties *Title, Description and Name.*
3. If the user enters a search string such as *physics resourcetype:lecture*, the Search takes the resource type as *Lecture* and searches for the word “design” in properties of resource type *Lecture* including the ones it derived from its base types. Namely, the properties *Title, Description, Notes, Abstract, Series* and *Venue* will be searched.

These implicit columns change per resource type and are configurable through an XML file.

The configuration XML is of the following form:

|  |
| --- |
| <?xml version="1.0" encoding="utf-8" ?>  <implicitProperties xmlns="urn:zentity">  <!-- Zentity.Core Module -->  <module namespace="Zentity.Core">  <resourceType name="Resource">  <property>Title</property>  <property>Description</property>  </resourceType>  </module>  <!-- Zentity.ScholarlyWorks Module -->  <module namespace="Zentity.ScholarlyWorks">  <resourceType name="Tag">  <property>Name</property>  </resourceType>  <resourceType name="Contact">  <property>Email</property>  </resourceType>  </module>  </implicitProperties> |

The complete configuration of implicit fields can be found at *<Installation\_path>\WebUI\Search\ImplicitProperties.config*.

|  |
| --- |
| Note: Only string columns can be a part of implicit column list. Other column types will be ignored. |

### Property Based Search

Property based search would happen through property tokens. Property tokens are all the properties of all the classes within all modules of Zentity Data Model. Through Core search determines the exact table and column to which the property tokens match to.

Examples:

1. Search for a resource with title containing a word “Design patterns” OR title containing a word “Software architecture”, the query would look like:

*Title: “Design patterns” OR Title: “Software architecture”*

The above query can also be written as:

*Title: (“Design patterns” OR “Software architecture”)*

1. Search for a resource with title containing a word “Design patterns” AND title containing a word “Software architecture”, the query would look like:

*Title: “Design patterns” AND Title: “Software architecture”*

The above query can also be written as:

*Title: (“Design patterns” AND “Software architecture”)*

OR

*Title: “Design patterns” Title: “Software architecture”*

OR

*Title: (“Design patterns” “Software architecture”)*

|  |
| --- |
| Note: If no logical operator is specified between conditions then an operator ‘AND’ is assumed between them. |

### Search Using Special Tokens

Search also provides special tokens which act like property tokens but will have more resource fields mapped to it. By default special tokens like *date* will be mapped to *‘date added’*, *‘date modified’* etc. This can again be configurable through an XML file.

Similar to [value search](#_Search_by_Value), it also searches in the specified columns of specified resource type as well as all its base types defined in the XML.

The default special tokens and their mappings to resource properties are as follows.

|  |  |  |  |
| --- | --- | --- | --- |
| Special Token | Resource Namespace | Resource Type | Resource Properties |
| date | Zentity.Core | Resource | 1. DateModified 2. DateAdded |
| Zentity.ScholarlyWorks | PersonalCommunication | 1. DateExchanged |
| Lecture | 1. DateEnd 2. DateStart |
| Publication | 1. DatePublished 2. DateSubmitted 3. DateAccepted |
| Tutorial | 1. DatePresented |
| Patent | 1. DateApproved 2. DateRejected |
| name | Zentity.ScholarlyWorks | Tag | 1. Name |
| Person | 1. FirstName 2. LastName 3. MiddleName |
| Experiment | 1. Name |
| Publication | 1. BookTitle 2. Publisher |
| Video | 1. Director |
| Journal | 1. JournalName |
| Proceedings | 1. EventName |
| email | Zentity.ScholarlyWorks | Contact | 1. Email |
| PersonalCommunication | 1. From 2. To |

Examples:

1. Search for a resources added or modified since last week.

*Date: >”last week”*

Search can be overloaded by specifying a resource type field to extract only resources of a particular type.

1. Search for all Lectures added, modified, started or ended since last week.

*Date: >”last week” ResourceType: Lecture*

*Resource* is a base type of *Lecture*; hence it would search in four properties. As in the above table, two properties *DateEnd* and *DateStart* are selected for resource type *Lecture* and similarly resource type *Resource* has two properties *DateModified* and *DateAdded*.

Special property tokens XML is of the following form:

|  |
| --- |
| <?xml version="1.0" encoding="utf-8" ?>  <specialTokens xmlns="urn:zentity">  <token name="date" dataType="DateTime">  <!-- Zentity.Core Module -->  <module namespace="Zentity.Core">  <resourceType name="Resource">  <property>DateModified</property>  <property>DateAdded</property>  </resourceType>  </module>  <!-- Zentity.ScholarlyWorks Module -->  <module namespace="Zentity.ScholarlyWorks">  <resourceType name="PersonalCommunication">  <property>DateExchanged</property>  </resourceType>  <resourceType name="Lecture">  <property>DateEnd</property>  <property>DateStart</property>  </resourceType>  </module>  </token>  </specialTokens> |

The complete configuration of special tokens can be found at *<Installation\_path>\WebUI\Search\SpecialTokens.config*.

|  |
| --- |
| Note:   1. [*ResourceType*](#_ResourceType_token) is a special token. 2. Similar to implicit properties we simply ignore the columns that are not of the data type of the special token. Also note that special tokens can be of varied data types, but implicit columns will always be of data type string (the implicit data type for implicit columns). |

### Search Using Predicates

Search for a resource providing predicate filter criteria. The predicate filter criterion is used to filter out the object or subject side of the relationship. So basically you search for subjects or objects in a predicate. The predicate names stored in the database and the predicate tokens that can be configured through an XML file make up the list of supported predicate tokens.

Given a token (assuming it is a predicate token); the Search would first look for the token in the predicate XML. If it is not found, it will query Core for the list of predicate ‘Name’s and try to find a match. This way we can query for authors using a predicate token *author* or predicate name *ScholarlyWorkIsAuthoredBy*.

The default friendly names and their actual predicates are as follows.

|  |  |  |
| --- | --- | --- |
| Friendly Name | Actual Predicate | Is Reverse Relation |
| author | ScholarlyWorkIsAuthoredBy | False |
| authoredwork | ScholarlyWorkIsAuthoredBy | True |
| tag | ScholarlyWorkItemHasTag | False |
| taggeditem | ScholarlyWorkItemHasTag | True |
| categorizeditem | CategoryNodeHasScholarlyWorkItem | False |
| category | CategoryNodeHasScholarlyWorkItem | True |
| citedby | ScholarlyWorkIsCitedBy | False |
| citation | ScholarlyWorkIsCitedBy | True |
| addedby | ScholarlyWorkItemIsAddedBy | False |
| addeditem | ScholarlyWorkItemIsAddedBy | True |
| presenter | ScholarlyWorkIsPresentedBy | False |
| presentedwork | ScholarlyWorkIsPresentedBy | true |
| version | ScholarlyWorkHasVersion | false |
| versionof | ScholarlyWorkHasVersion | true |
| contributor | ScholarlyWorkHasContributionBy | false |
| contributioninwork | ScholarlyWorkHasContributionBy | true |
| editor | ScholarlyWorkIsEditedBy | false |
| editedwork | ScholarlyWorkIsEditedBy | true |
| representation | ScholarlyWorkHasRepresentation | false |
| representationof | ScholarlyWorkHasRepresentation | true |

|  |
| --- |
| Note: If the value of reverse relation is *true*, then the meaning of corresponding friendly token is reverse as that of actual predicate. For example,  Consider the friendly names *author* and *authoredwork*. Both of these friendly names map to the same predicate *ScholarlyWorkIsAuthoredBy*.  The reverse relation attribute for *author* is *false*; hence its meaning would be same as that of the actual predicate.  That is, the queries *ScholarlyWorkIsAuthoredBy: ‘Eric Gamma’* and *Author: ‘Eric Gamma’* would return the resources authored by ‘Eric Gamma’.  However the reverse relation attribute for *authoredwork* is *true*; hence its meaning would be reverse as that of the actual predicate.  That is, the query *authoredwork: ‘Design Patterns’* would return all the authors of resources containing ‘Design Patterns’ in their implicit fields. |

Examples of predicate tokens:

1. Search for resources with title containing a word “Design patterns” AND authored by “Eric Gamma”

*‘Design patterns’ author: ‘Eric Gamma’*

In the same context a query that might return a bigger result set could be formed as:

*Design patterns author: ‘Eric Gamma’*

The query formed would look like:

*Title:Design AND Title:patterns AND author: (Title:‘Eric Gamma’)*

*The implicit field is assumed to be Title, which might not be the case always.*

OR

*Design patterns author: (Eric Gamma)*

Similar query can be written in a detailed format as:

*Title: “Design patterns” author: (Title: “Eric Gamma”)*, which is same as

*Title: “Design patterns” AND author: (Title: “Eric Gamma”)*

OR

*author: (Title: “Eric Gamma”) AND Title: “Design Patterns”*

These queries are more precise and no assumption about implicit fields has been made.

1. Search for all books authored by “Eric Gamma”.

A detailed query would look like:

*ResourceType: “Book” author: (Title: “Eric Gamma”)*

A simple form of the same query:

*author: ’Eric Gamma’ ResourceType:Book*

1. Search for all resources authored by “Eric Gamma” AND “John Vlissides”

A detailed query would look like:

*author: (Title:“Eric Gamma”) author: (Title: “John Vlissides”)*

Some simpler forms:

*author: ’Eric Gamma’ author:’John Vlissides’*

*author:(“Eric Gamma” AND “John Vlissides”)*

*author:(“Eric Gamma” “John Vlissides”)*

1. Search for all resources authored by “Eric Gamma” AND published by “Addison-Wesley”

A detailed query would look like:

*author: (Title: “Eric Gamma”) AND publisher: (Title: “Addison-Wesley”)*

A simpler form:

*author: ’Eric Gamma’ publisher: ’Addison-Wesley’*

1. Search for all resources tagged with ‘Physics’

A detailed query would look like:

*Tag: (Title: ‘Physics’)*

A simpler form:

*Tag: Physics*

|  |
| --- |
| Note: User defined Zentity predicates would be searchable without adding an entry to the predicate XML. But for a user friendly name to these predicates a user will have to add an entry in predicate XML. |

Predicate XML

|  |
| --- |
| <?xml version="1.0" encoding="utf-8" ?>  <predicateTokens xmlns="urn:zentity">  <!-- ScholarlyWorkIsAuthoredBy -->  <token name="author" predicate="ScholarlyWorkIsAuthoredBy" reverseRelation="false" />  <token name="authoredwork" predicate="ScholarlyWorkIsAuthoredBy" reverseRelation="true" />  </predicateTokens> |

The complete configuration of predicate tokens can be found at *<Installation\_path>\WebUI\Search\PredicateTokens.config*.

### Search Operators

#### Logical Operators

Following logical operators are supported. Both the logical operators satisfy the associative and commutative laws of an operator.

* AND
* OR

Example: *Title: “Design patterns” OR Title: “Compilers”*

*AND* will take higher precedence than *OR* by default unless overridden with brackets.

If no logical operator is specified between to conditions then an *AND* is assumed between them.

Example:

1. The queries like:

*Title: “Design patterns” Date: 01/01/2009*

Are interpreted as:

*Title: “Design patterns” AND Date: 01/01/2009*

1. The queries like:

*Title: “Design patterns” OR Title:”Sql Server” AND DateAdded: 01/01/2009*

Are interpreted as:

Search for resources where title contains ‘Design Patterns’ OR title contains

‘Sql Server’ and added on ‘01/01/2009’.

*(Title: “Design patterns”) OR ( Title:”Sql Server” AND DateAdded: 01/01/2009)*

|  |
| --- |
| Note:   1. In case of more than one consecutive logical operators like,   *Title: “Design patterns” AND AND OR Date: 01/01/2009*  Only last logical operator in the sequence is considered and others are discarded.  So the above query would be interpreted as:  *Title: “Design patterns” OR Date: 01/01/2009*   1. Logical operators at the start and end of the queries are discarded. And no error is thrown.   *OR AND OR Title: “Design patterns” Date: 01/01/2009 AND*  Would be interpreted as:  *Title: “Design patterns” Date: 01/01/2009*   1. Logical operators at unnecessary/illogical places like brackets containing only operators and no condition are also discarded.   Example:  *Title: “Design patterns” (AND (OR)) Date: 01/01/2009*  OR  *Title: “Design patterns” (AND) Date: 01/01/2009*  OR  *Title: “Design patterns” (OR) Date: 01/01/2009*  Would be interpreted as:  *Title: “Design patterns” Date: 01/01/2009*   1. Logical operators are case sensitive to differentiate between actual strings and operators. |

#### Conditional Operators

Search supports following conditional operators:

* = (Equal to)
* > (Greater than)
* < (Less than)
* >= (Greater than or equal to)
* <= (Less than or equal to)
* != (Not equal to)

All conditional operators are treated with equal precedence. They can only be used with property tokens and special tokens.

Examples:

1. Search for resources with title equal to ‘Design patterns’.

*Title: = “Design patterns”*

1. Search for resources with title greater than or equal to ‘Design patterns’.

*Title: >= “Design patterns”*

1. Search for resources with size equal to 10.

*size: = 10*

1. Search for files with size not equal to 10.

*ResourceType:File size: != 10*

More examples:

*size: > 10 AND size: < 20* can be written in multiple ways.

1. *size: (> 10 AND < 20)* (with spaces between operator and operand)
2. *size: (>10 AND <20)* (without spaces between operator and operand)
3. *size: (> 10 < 20)*OR*size: (>10 <20)*

|  |
| --- |
| Note:   1. In case of non string data type, if a conditional operator is not specified then an equal to (=) operator is assumed.   Example: *size: 10* would be interpreted as *size: = 10*   1. In case of string data type,   *Title: “Design patterns”* and *Title: = “Design patterns”* would have different meanings.  *Title: “Design patterns”* means title containing a string ‘Design patterns’.  *Title: = “Design patterns”* means title equals to ‘Design patterns’.   1. *size: = (10 OR 20)* is not same as *size: (=10 OR =20)*.   *size: = (10 OR 20)* would return unexpected results. Search would treat *size: =* one part and *(10 OR 20)* as second part*.* |

#### ‘NOT’ Operator

The ‘NOT’ operator is case sensitive. It can be used to negate one or more conditions in a search query.

Examples:

1. Search for a scholarly work that does not have some property value.

For example, search for persons who are neither Eric nor Tom.

*NOT Title:Eric AND NOT Title:Tom AND ResourceType:Person*

OR

*Title: NOT Eric AND Title: NOT Tom AND ResourceType:Person*

OR

*Title: (NOT Eric AND NOT Tom) AND ResourceType:Person*

OR

*NOT Title:(Eric OR Tom) AND ResourceType:Person*

1. Search on negation of a relationship.

For example, search for all items not authored by Eric.

*NOT author:Eric*

Search for all items authored by anyone but not by ‘Eric’ alone

*author:NOT Eric*

OR

*author:(NOT Eric)*

1. Search on negation of implicit value.

For example, search for all items with *Title* and *Description* not containing any word starting with ‘Eric’.

*NOT Eric*

1. Search on negation of operators like <, <=, >, >=, =.

For example, search for all items with size not equal to “1”.

*Resourcetype:File NOT Size:= 1*

OR

*Resourcetype:File Size: != 1*

Search for all items with size not less than “1” and not more than “5”.

*Resourcetype:File NOT Size:<1 AND NOT Size:>5*

OR

*Resourcetype:File NOT Size:(<1 OR >5)*

OR

*Resourcetype:File NOT (Size:<1 OR Size:>5)*

OR

*Resourcetype:File (Size: NOT <1 AND Size: NOT >5)*

OR

*Resourcetype:File Size: NOT (<1 OR >5)*

|  |
| --- |
| NOTE: If there are even numbers of consecutive NOT operators in a search query then they get cancelled out.  For example,   1. *Resourcetype:File NOT NOT Size:= 1*  be interpreted as *Resourcetype:File Size:= 1.* 2. *Resourcetype:File NOT NOT NOT Size:= 1* would be interpreted as *Resourcetype:File NOT Size:= 1.* |

### Types of Literals

Following types of literals are supported:

* **Strings**

Strings can be:

* Characters demarcated by double/single quotes.

Double/single quotes in a query string will have to be escaped if they are to be searched for. The escape character is ‘\’.

In this case, the string is interpreted as “*token containing a word/phrase equal to the characters demarcated by quotes”*.

1. Search queries having quotes, wildcards need to be escaped.

Example: *Description: “\”Design patterns\””*

**OR**

The query can be written without escape character ‘\’ as

*Description: ‘"Design patterns”’* i.e. characters demarcated by single quotes.

|  |
| --- |
| Note: Escape character ‘\’ is required only in following cases:   1. Characters are demarcated by double quotes and if it contains a double quote, then that double quote needs to be escaped using the escape character.   Example: ***“Design” patterns”*** needs to be written as *“Design\” patterns”*.  But for***‘Design” patterns’*** an escape character is not required.   1. Characters are demarcated by single quotes and if it contains a single quote, then that single quote needs to be escaped using the escape character.   Example: ***‘Design’ patterns’*** needs to be written as *‘Design\’ patterns’*.  But for ***“Design’ patterns”*** an escape character is not required. |

* A single word following a token.

In this case, the string is interpreted as *“token containing a word starting with the specified word”*.

* A single word without quotes.

|  |
| --- |
| Note:   1. The characters that form a string can be any Unicode/ASCII characters. 2. All the conditional operators are applicable for strings literals. |

* **Dates**

Date literals are expected to be in a format similar to “the current culture’s short date format”.

Quotes are optional for date values.

In case of *date*, no conditional operator means an ‘=’ operator.

1. Search for resources added on 23rd November 2008.

The following queries would be interpreted as search for resources with DateAdded = “23/11/2008”:

*DateAdded:23/11/2008*OR*DateAdded: = 23/11/2008*

OR

*DateAdded:”23/11/2008”*OR*DateAdded: = ”23/11/2008”*

OR

*DateAdded: ‘23/11/2008’* OR*DateAdded: = ‘23/11/2008’*

The query syntax allows for exotic literals for relative dates (similar to WDS)

* Relative dates: Today, tomorrow, yesterday

Example: *DateAdded: today*

* Multi-word relative dates: This week, next month, last week, past month, or next year.

Example: *DateAdded: “this week”*OR*DateAdded: = ‘this week’*

Relative dates will always be with respect to today’s time frame.

The available relative date literals are as follows:

|  |  |
| --- | --- |
| Date Literals | Meaning |
| Today | The current day. |
| Tomorrow | Next day relative to the current day. |
| Yesterday | Previous day relative to the current day. |
| This week | The current week starting from Sunday. |
| Next week | The next week starting from next Sunday. |
| Last Week | The last week. |
| This month | The current month starting from 1st. |
| Next month | The next month starting from 1st. |
| Last month | The last month starting from 1st. |
| This year | The current year starting from 1st of January. |
| Next year | The next year starting from 1st of January. |
| Past year | The previous year starting from 1st of January. |

|  |
| --- |
| Note: The date literals are not case sensitive. Multi word date literals should be enclosed within single/double quotes OR remove the space between the two words.  Search is responsible for converting such literals to a date in the custom locale format. |

1. Search for resources added during a period

A range search is possible by using < and > operators within a single query. Example, to find all Books titled “Design patterns” and printed between year 2001 and year 2007, she can have to fire following query:

*Title: =“Design patterns” AND DateAdded:(> 01/01/2001 AND < 31/12/2007)*

OR

*Title: =“Design patterns” DateAdded:(>01/01/2001 <31/12/2007)*

* **Numbers (integer and real)**

Both real and integer literals are considered as number literals. Real numbers are considered only in the following format, [-+]?[0-9]+\.?[0-9]\*

Quotes are optional for numeric values.

By default “=” operator would be considered in case of numbers, if no conditional operator is specified.

1. Search for files having size 1KB.

The following queries would be interpreted as search for files with Size = “1”:

*Size:1 ResourceType:File*OR*Size: = 1 ResourceType:File*

OR

*Size:”1” ResourceType:File*OR*Size: = ”1” ResourceType:File*

OR

*Size:’1’ ResourceType:File*OR*Size: = ‘1’ ResourceType:File*

## Grouping

Brackets can be provided in the search expression to allow a user to override the default precedence of operators. Brackets will always have higher precedence over all the operators provided by search grammar. Associativity within a bracketed sub-search query still happens to be from left to right.

Example:

Search for all resources titled “Eric Gamma” and having description containing either “author” or “editor”

A detailed query would look like:

*Title: = “Eric Gamma” AND (Description:”author” OR Description:”editor”)*

OR

*Title: = “Eric Gamma” AND (Description: (”author” OR ”editor”))*

### ResourceType Token

*ResourceType* token can be used by the user to force the resource type that she wants the query (both simple and complex) to be filtered upon. Only one *ResourceType* token is allowed within a simple/complex query. This implies that a user cannot run a single query to search for resources of type let’s say, Book and Lecture, instead she will have to manually fire two queries.

Examples:

1. Search for books with title equals to “Design Patterns”.

*Title: = “Design Patterns” AND ResourceType:Book*

OR

*ResourceType:Book AND Title: = “Design Patterns”*

OR

*Title: = “Design Patterns” ResourceType:Book*

1. In case of predicate search, the query can have 2 different resource types. One resource type for subject criteria and other resource type for object criteria.

For example, Search for *Book* resources where title contains a word ‘design’ AND authored by Eric, Eric is a person.

*Title:’design’ resourcetype:Book author:(Eric resourcetype:Person)*

In this case,

*Title: design resourcetype: Book* is the subject criteria and *author: (Eric resourcetype: Person)* is the object criteria.

1. If more than one resource types are specified then only first one is considered and other are discarded.

For example, search for books OR lectures with *Title* equals to “Design Patterns”.

*Title: = “Design Patterns” AND (ResourceType:Book OR ResourceType:Lecture)*

Then only resource type *Book* is considered and *Lecture* is discarded. The query is interpreted as:

*Title: = “Design Patterns” AND ResourceType:Book*

1. Resource type can be specified with or without single/double quotes.

For example,

*ResourceType:Book*

OR

*ResourceType:”Book”*

OR

*ResourceType:’Book’*

|  |
| --- |
| Note:  *ResourceType: (Book OR Lecture)*  Search would treat the above query as 3 independent strings ‘ResourceType:’, ‘Book’ and ‘Lecture’. It would try to search for resources with ‘ResourceType:’ and ‘Book’ OR ‘ResourceType:’ and ‘Lecture’ their implicit fields (by default *Title* and *Description*). |

## Resource Type Inference

Search will try and infer on which type of resources to return depending on the fields being searched for. This inference will happen only for fields/properties that are exclusive among all classes within the Scholarly Work hierarchy.

Examples:

1. For the following query, resource type *Resource* would be inferred, because only the resource type *Resource* has a property called *Title*.

*Title: = “Design Patterns”*

So the query would be interpreted as:

*Title: = “Design Patterns” AND ResourceType:Resource*

1. For the following query, resource type *Resource* would be inferred for subject criteria and *Person* for object criteria, because only the resource type *Resource* has a property called *Title* and *FirstName* is a property of resource type *Person*.

*Title:’design’ author:(FirstName:Eric)*

So the query would be interpreted as:

*Title:’design’ resourcetype:Resource AND author:(Eric resourcetype:Person)*

1. *ISBN:’99921-58-10-7’ author:(FirstName:Eric)*

Would be interpreted as:

*ISBN:’99921-58-10-7’ resourcetype:Book AND author:(Eric resourcetype:Person)*

Note, here the resource type *Book* is inferred for the subject criteria because only *Book* has the property *ISBN*.

1. For the following queries, resource type *Book* would be inferred. Resource type *Resource* has *Title* and resource type *Book* has *ISBN* but *Book* is derived from *Resource*, hence *Book* is considered.

*Title:’design’ ISBN:’99921-58-10-7’* OR*ISBN:’99921-58-10-7’ Title:’design’*

So the queries would be interpreted as:

*(Title:’design’ AND ISBN:’99921-58-10-7’) resourcetype:Book*

1. For the following query, resource type *Book* would be inferred. Resource type *Book* has *ISBN* and resource type *File* has *Size* but neither *Book* is derived from *File* nor *File* is derived from *Book*, hence in this case, the resource type of first property token is considered.

*ISBN:=’99921-58-10-7’ size:5*

Would be treated as:

*(ISBN:=’99921-58-10-7’ size:5) resourcetype:Book*

And it would be interpreted as, search for a book which has *ISBN* equals to ’99921-58-10-7’ and its one of its implicit fields contain a word ‘size:5’.

In the above case, ‘size’ would not be treated as a property token, because resource type *Book* doesn’t have a property called ‘size’. Hence ‘size:5’ would be treated as a normal value search string.

|  |
| --- |
| Note:   1. If more than one resource types have the same property then it would throw an exception saying *‘Ambiguous resource type’*. 2. If no *property token* found in the query, then resource type *Resource* is assumed. |

## Excluded Resource Types

Resource types specified in *ExcludedResourceTypes.config* are excluded from the search results.

User can not search for those types of resources. By default the security related resource types *Identity* and *Group* are excluded.

Based on the above default configuration,

1. If the user enters a search string such as *John resourceType:Identity*, the Search would return zero results giving an error message *‘Invalid resource type’*.
2. If the user enters a search string such as *resourceType:Resource*, the Search would return all the resources except for of type *Identity* and *Group*.
3. If the user enters a search string such as *Title:’Eric’*, the Search would return all the resources where *Title* contains a word ‘Eric’ but it would subtract the resources of type *Identity* and *Group* from result set.

These excluded resource types are configurable through an XML file.

The configuration XML is of the following form:

|  |
| --- |
| <?xml version="1.0" encoding="utf-8"?>  <excludedResourceTypes xmlns="urn:zentity">  <module namespace="Zentity.Security.Authorization">  <resourceType name="Identity" />  </module>  </excludedResourceTypes> |

The complete configuration of special tokens can be found at *<Installation\_path>\WebUI\Search\ExcludedResourceTypes.config*.

## Excluded Predicates

Predicates specified in *ExcludedPredicates.config* are excluded from the search and are treated as normal search words.

User cannot use those predicates for predicate query. By default the following security related predicates are excluded.

* IdentityBelongsToGroups
* DenyCreateAccess
* DenyDeleteAccess
* DenyOwnershipOf
* DenyReadAccess
* DenyUpdateAccess
* HasCreateAccess
* HasDeleteAccess
* HasReadAccess
* HasUpdateAccess
* IsOwnerOf

Based on the above default configuration,

1. If the user enters a search string such as *IdentityBelongsToGroups: group1*, the Search would not treat ‘IdentityBelongsToGroups’ as a predicate. Instead it would interpret the query as two separate words ‘IdentityBelongsToGroups:’ and ‘group1’ and treat as a value search.
2. If the user enters a search string such as *IdentityBelongsToGroups:group1*, the Search would not treat ‘IdentityBelongsToGroups’ as a predicate. Instead it would interpret the query as a single word ‘IdentityBelongsToGroups:group1’ and treat as a value search.

These excluded predicates are configurable through an XML file.

The configuration XML is of the following form:

|  |
| --- |
| <?xml version="1.0" encoding="utf-8"?>  <excludedPredicates xmlns="urn:zentity">  <predicate name="IdentityBelongsToGroups"/>  </excludedPredicates> |

The complete configuration of special tokens can be found at *<Installation\_path>\WebUI\Search\ExcludedPredicates.config*.

## Token Priorities

Mainly there are three types of [tokens](#_Tokens).

1. Special Tokens
2. Property Tokens
3. Predicate Tokens

Special tokens have highest priority while predicate tokens have lowest priority.

For example,

If a user searches with a token *Email* and *Email* is a special token as well as a property token, then Search would treat it as a special token.

When a token is found, Search first checks whether it’s a special token. It that token is not a special token then it checks for property token and finally checks for predicate tokens.

## Content search

Search can be used to search for files based on a list of words in the uploaded content. Results returned would be of type ‘Zentity.Core.File’.

Content search is possible only on FTS enabled database.

It takes only a list of words to be searched for. *Tokens* and *operators* cannot be used for content search.

Search creates a FREETEXT query for words supplied by a user. When FREETEXT is used, the full-text query engine internally performs the following actions on the *search query*:

* Separates the string into individual words based on word boundaries (word-breaking).
* Generates inflectional forms of the words (stemming).
* Identifies a list of expansions or replacements for the terms based on matches in the thesaurus.

Examples of content search:

1. Search for files containing a word “Design”.

*Design*

The above query would return all the files containing the word “Design” in their *content*.

1. Search for files containing word “Design” OR “Algorithm”.

*Design Algorithm*

Note: There is no need to specify an operator.

1. Search using inflectional forms of the words (stemming)

The following search queries would return the same set of results:

* 1. *Play*
  2. *Playing*
  3. *Played*

|  |
| --- |
| Note: During index creation, the Full-Text Engine omits stop words from the full-text index. A stop word is a *token* that does not have linguistic meaning.  Hence, stop words such as “a”, “and”, “is”, “the” etc. are left out of the full-text index since they are known to be useless to a search.  For example, a user who wants to search for all files containing all 3 words “computer”, **“and”,** and “software”,  *computer and software*  But the SQL FTS engine recognizes “**AND**” as a stop word, and discards it. It considers only the words “computer” and “software”. It then matches all files in which “computer” OR “software” is present.  Similarly, the queries which contain only stop words like *and* and *or and the* would always return zero results. |

## Application Configuration Section

To configure Search, user needs to follow the following steps.

### Implicit Fields

1. Implicit properties for Search can be configured using an XML file. The XML file must conform to following XML schema definition.

|  |
| --- |
| <?xml version="1.0" encoding="utf-8"?>  <xs:schema attributeFormDefault="unqualified" elementFormDefault="qualified" targetNamespace="urn:zentity" xmlns:xs="http://www.w3.org/2001/XMLSchema">  <xs:element name="implicitProperties">  <xs:complexType>  <xs:sequence>  <xs:element minOccurs="0" maxOccurs="unbounded" name="module">  <xs:complexType>  <xs:sequence>  <xs:element minOccurs="0" maxOccurs="unbounded" name="resourceType">  <xs:complexType>  <xs:sequence>  <xs:element minOccurs="0" maxOccurs="unbounded" name="property" type="xs:string" />  </xs:sequence>  <xs:attribute name="name" type="xs:string" use="required" />  </xs:complexType>  </xs:element>  </xs:sequence>  <xs:attribute name="namespace" type="xs:string" use="required" />  </xs:complexType>  </xs:element>  </xs:sequence>  </xs:complexType>  </xs:element>  </xs:schema> |

1. Specify path of the XML file in application configuration file using *ImplicitPropertiesFileName* key as:

|  |
| --- |
| <?xml version="1.0" encoding="utf-8"?>  <configuration>  <appSettings>  <add key="ImplicitPropertiesFileName" value="ImplicitProperties.config" />  </appSettings>  </configuration> |

The file path can be absolute as relative to the base directory that the assembly resolver uses to probe for assemblies.

If key *ImplicitPropertiesFileName* is not specified in application configuration file, then Search would ignore the implicit query parts from search query.

### Predicates Tokens

1. Friendly names for predicate names stored in the database can be configured through an XML file. The XML file must conform to following XML schema definition.

|  |
| --- |
| <?xml version="1.0" encoding="utf-8"?>  <xs:schema attributeFormDefault="unqualified" elementFormDefault="qualified" targetNamespace="urn:zentity" xmlns:xs="http://www.w3.org/2001/XMLSchema">  <xs:element name="predicateTokens">  <xs:complexType>  <xs:sequence>  <xs:element minOccurs="0" maxOccurs="unbounded" name="token">  <xs:complexType>  <xs:attribute name="name" type="xs:string" use="required" />  <xs:attribute name="predicate" type="xs:string" use="required" />  <xs:attribute name="reverseRelation" type="xs:boolean" use="required" />  </xs:complexType>  </xs:element>  </xs:sequence>  </xs:complexType>  </xs:element>  </xs:schema> |

1. Specify path of the XML file in application configuration file using *PredicateTokensFileName* key as:

|  |
| --- |
| <?xml version="1.0" encoding="utf-8"?>  <configuration>  <appSettings>  <add key="PredicateTokensFileName" value="PredicateTokens.config" />  </appSettings>  </configuration> |

The file path can be absolute as relative to the base directory that the assembly resolver uses to probe for assemblies.

Use of this file is optional. If the key *PredicateTokensFileName* is not found in application configuration file, then Search would consider only the predicate names stored in the database as predicate tokens.

### Special Tokens

1. Special tokens can be configured through an XML file. The XML file must conform to following XML schema definition.

|  |
| --- |
| <?xml version="1.0" encoding="utf-8"?>  <xs:schema attributeFormDefault="unqualified" elementFormDefault="qualified" targetNamespace="urn:zentity" xmlns:xs="http://www.w3.org/2001/XMLSchema">  <xs:element name="specialTokens">  <xs:complexType>  <xs:sequence>  <xs:element minOccurs="0" maxOccurs="unbounded" name="token">  <xs:complexType>  <xs:sequence>  <xs:element minOccurs="0" maxOccurs="unbounded" name="module">  <xs:complexType>  <xs:sequence>  <xs:element minOccurs="0" maxOccurs="unbounded" name="resourceType">  <xs:complexType>  <xs:sequence>  <xs:element minOccurs="0" maxOccurs="unbounded" name="property" type="xs:string" />  </xs:sequence>  <xs:attribute name="name" type="xs:string" use="required" />  </xs:complexType>  </xs:element>  </xs:sequence>  <xs:attribute name="namespace" type="xs:string" use="required" />  </xs:complexType>  </xs:element>  </xs:sequence>  <xs:attribute name="name" type="xs:string" use="required" />  <xs:attribute name="dataType" type="xs:string" use="required" />  </xs:complexType>  </xs:element>  </xs:sequence>  </xs:complexType>  </xs:element>  </xs:schema> |

1. Specify path of the XML file in application Configuration file using *SpecialTokensFileName* key as:

|  |
| --- |
| <?xml version="1.0" encoding="utf-8"?>  <configuration>  <appSettings>  <add key="SpecialTokensFileName" value="SpecialTokens.config" />  </appSettings>  </configuration> |

The file path can be absolute as relative to the base directory that the assembly resolver uses to probe for assemblies.

Use of this file is optional. If the key *SpecialTokensFileName* is not found in application configuration file, then Search would not consider any token as a special token.

### Excluded Predicates

1. Predicates which should be excluded from Search can be configured through an XML file. The XML file must conform to following XML schema definition.

|  |
| --- |
| <?xml version="1.0" encoding="utf-8"?>  <xs:schema attributeFormDefault="unqualified" elementFormDefault="qualified" targetNamespace="urn:zentity" xmlns:xs="http://www.w3.org/2001/XMLSchema">  <xs:element name="excludedPredicates">  <xs:complexType>  <xs:sequence>  <xs:element minOccurs="0" maxOccurs="unbounded" name="predicate">  <xs:complexType>  <xs:attribute name="name" type="xs:string" use="required" />  </xs:complexType>  </xs:element>  </xs:sequence>  </xs:complexType>  </xs:element>  </xs:schema> |

1. Specify path of the XML file in application configuration file using *ExcludedPredicatesFileName* key as:

|  |
| --- |
| <?xml version="1.0" encoding="utf-8"?>  <configuration>  <appSettings>  <add key="ExcludedPredicatesFileName" value="ExcludedPredicates.config" />  </appSettings>  </configuration> |

The file path can be absolute as relative to the base directory that the assembly resolver uses to probe for assemblies.

Use of this file is optional. If the key *ExcludedPredicatesFileName* is not found in application configuration file, then Search would not exclude any predicate token.

### Excluded Resource Types

1. Resource types which should be excluded from Search can be configured through an XML file. The XML file must conform to following XML schema definition.

|  |
| --- |
| <?xml version="1.0" encoding="utf-8"?>  <xs:schema attributeFormDefault="unqualified" elementFormDefault="qualified" targetNamespace="urn:zentity" xmlns:xs="http://www.w3.org/2001/XMLSchema">  <xs:element name="excludedResourceTypes">  <xs:complexType>  <xs:sequence>  <xs:element minOccurs="0" maxOccurs="unbounded" name="module">  <xs:complexType>  <xs:sequence>  <xs:element minOccurs="0" maxOccurs="unbounded" name="resourceType">  <xs:complexType>  <xs:attribute name="name" type="xs:string" use="required" />  </xs:complexType>  </xs:element>  </xs:sequence>  <xs:attribute name="namespace" type="xs:string" use="required" />  </xs:complexType>  </xs:element>  </xs:sequence>  </xs:complexType>  </xs:element>  </xs:schema> |

1. Specify path of the XML file in application configuration file using *ExcludedResourceTypesFileName* key as:

|  |
| --- |
| <?xml version="1.0" encoding="utf-8"?>  <configuration>  <appSettings>  <add key="ExcludedResourceTypesFileName" value="ExcludedResourceTypes.config" />  </appSettings>  </configuration> |

The file path can be absolute as relative to the base directory that the assembly resolver uses to probe for assemblies.

Use of this file is optional. If the key *ExcludedResourceTypesFileName* is not found in application configuration file, then Search would not exclude any resource type.

# Known Issues

There are some known issues while searching for a resource based on its property values (metadata).

1. If full text search (FTS) is not enabled on database, then following queries would return the same set of results.
   1. *Title: comp*
   2. *Title: “comp”*

The above queries would be interpreted as search for resources with title containing a ***substring*** ‘comp’.

While in case of FTS enabled database,

1. *Title: comp* would search for resources with title containing a ***word starting with*** ‘comp’
2. *Title: “comp”* would search for resources with title containing a ***word equal*** to ‘comp’

So in case of Non-FTS database, search may return a larger set of results.

1. In case of FTS enabled database, stop words such as “a”, “and”, “is”, “the” etc. are left out of the full-text index since they are known to be useless to a search. Hence this can have unexpected results when the stop words are used by the user in a search query.

For example, a user who wants to search for all resources with *Title* containing a sequence “computer **and** software”,

*Title: “computer and software”*

But the SQL FTS engine recognizes “**AND**” as a stop word, and discards it.

It then matches all documents in which “computer” and “software” are separated by other noise words. Search would return resources with T*itle* containing “computer ***or*** software”, and “computer ***but*** software”.

Although it ignores the stop words, the full-text index does take into account their position.

Hence, resources that contained simply “computer software” or “computer ***and the*** software” would not be returned.

Similarly, the queries like

*Title:”and”* and *Title: “Computer” AND Title:”and”*

Would always return zero results as *Title:”and”* contains only a noise word and it would be discarded.

# Appendix

## Tokens

On broad level there can be three types of tokens in a search string. All the tokens are case insensitive.

A token should be followed by a colon ‘:’ and there should not be any empty space between them.

* **Property tokens**

These are nothing but fields of classes within Scholarly Work hierarchy. All the fields within the hierarchy can be used. Token names that appear multiple times across the hierarchy won’t be used to infer about the resource type, instead an explicit mention of resource type would be required.

For examples on property tokens please check section on [Property based search](#_Property_based_search).

* **Predicate tokens**

Resources within Zentity repository are related to each other through predicates. Thus an author can be related to a book that he has authored through an authored-by predicate. The book becomes the subject and author the object. To make Search aware of the relationships and help user search for subjects depending upon the relationships they have with objects and further filter down on the objects through an object criteria we introduce the concept of predicate tokens. Predicate tokens are a set of

* + all the predicate ‘Names’ that are possible within Zentity system and
  + the predicate tokens defined in the configurable predicate XML

Predicate tokens in Zentity Search make it very powerful to search for resources depending on the relationships they have with object resources and vice-versa (Check [Search Using Predicates](#_Search_Using_Predicates) for more detailed examples).

* **Special tokens**

Special tokens don’t actually map to something physical in the Core database, but are a virtual and easy representation of a group of fields within the Core database. We have introduced three special tokens viz: name, date and email. An administrator of Zentity repository can configure this and add more special tokens that suit her and her users’ needs. No inference of resource type would be made based on special tokens. Instead the fields that they apply to would be inferred based on the resource type that the query applies to, this would be done in a lazy fashion, until resource type inferences happens.

Special tokens can be used as a hack to rename ‘property tokens’, thus a property name ‘File.Extension’ can actually be mapped to a special token ‘ext’. Check [Search Using Special Tokens](#_Search_using_special) for more examples.

* **ResourceType token**

*ResourceType* token can be used by the user to force the resource type that she wants the query (both simple and complex) to be filtered upon. Only one resource type will be allowed within a simple/complex query. This implies that a user cannot run a single query to search for resources of type let’s say, *Book* and *Lecture*, instead she will have to manually fire two queries. Check [ResourceType token](#_ResourceType_token) for more examples.

1. We would refer to Zentity Search as Search (note the capital S) elsewhere in the document. [↑](#footnote-ref-2)