7th ROW – Registration Operations Workshop

May 17th, 2018

Vancouver, Canada

Enhancing RDAP filtering capabilities

Mario Loffredo, Maurizio Martinelli

IIT-CNR/Registro.it

Together with sorting and paging, filtering is considered one of the most effective means to improve the management of REST APIs results. The main reasons for its presence are:

- Minimizing the bandwidth usage;
- Speeding up the response time;
- Improving the precision of the queries and, consequently, obtaining more reliable results;
- Decreasing CPU time and memory spent on both server and client.

At present, RDAP (RFC 7482) provides limited search capabilities because the condition of a search query consists of a single pattern. Therefore, a search query can potentially generate a large result set that, in the best case, must be scrolled looking for the desired data but, in the worst case, can be truncated according to the server limits. Even if a server would provide clients with sorting and paging capabilities, the extraction of the desired information, within a result set, could be very time consuming. Furthermore, users might be interested in performing searches that currently RDAP does not allow. For example, a registrar user might search his own domains for a certain status or for a specific event in a range of dates.

Therefore, in authors opinion, the best solution to fix such an inefficiency is to implement sorting, paging and filtering at server side. In this manner, clients could obtain always and only the desired results with the minimum response. The authors have already submitted two Internet Drafts, both facing the issues connected with subsetting of large RDAP responses. The former, written in collaboration with Scott Hollenbeck, was about results set sorting and paging and the latter was about partial responses. The current ongoing proposal about filtering definitively seems to complete RDAP extensions dedicated to the efficient management of large result sets.

The Registro.it RDAP public test server provides a new query parameter to enhance search filtering. The parameter is named ***filter*** and its value is a JSON object representing a condition expression. Describing the parameter value in JSON can enable clients and servers to deal with search conditions whose complexity ranges from very simple to extremely complicated.

Traditionally, a search condition includes a set of predicates combined by the logical operators AND, OR, and NOT. A predicate contains three components:

- a property name;
- an allowed operator for the property;
- a filter value (or a list of values) whose type is allowed for the property.

The authors have already defined in the I-D about sorting and paging a set of property references involved in the specification of sort criteria that can be adopted in the specification of a filter as well. The properties are:

- Object common properties:
    - registrationDate
    - reregistrationDate
    - lastChangedDate
    - expirationDate
    - deletionDate
    - reinstantiationDate
    - transferDate
    - lockedDate
    - unlockedDate

- Object specific properties:
    - Domain: ldhName
    - Nameserver: ldhName, ipV4, ipV6.
    - Entity: fn, handle, org, email, tel, country, countryName, locality

In addition to the above object properties, status and roles, should be also defined.

Then, in addition to the logical operators, it is reasonable to expect the presence of commonly used comparison operators like “equal”, “not equal”, “less than” and so on. Specific operators about strings like “contains”, “starts with”, “ends with” can be implemented using the “equal” operator and the wildcard. Appropriate operators on arrays should be considered too.

Finally, value types can be “string”, “number”, “boolean”, “datetime” or “array” (of primitive type). As regards to datetime values, RFC3339 full-date and date-time formats should be supported.

As a consequence of what stated above, the simplest JSON data structure describing a predicate consists of an array of three items. A complex predicate can be represented through a one member JSON object where the logical operator is the member name and the sub-predicates (one or more) are the member values. Even if, the deserialization of a JSON array in a data structure is not a standard capability of JSON libraries, it can be accomplished by a customization requiring a few lines of code but, on the other hand, the proposed representation for a predicate is much more compact than using a JSON object.

In the following, a JCR-based representation modelling the value of the filter parameter is presented:

```
@{root} $expression = {
       (
         $or_expression   |
         $and_expression  |
         $not_expression  |
         [ $predicate + ] |
         $predicate
        )
}

$or_expression = {
      "or" : [ $expression, $expression + ]
}
```

```
$and_expression = {
      "and" : [ $expression, $expression + ]
}

$not_expression = {
      "not" : $expression
}

$predicate = [
      /^[A-Za-z]+$/,
      (
        ("isnull"|"isnotnull")                      |
        ("eq"|"ne"), $basic_value                   |
        ("le"|"lt"|"gt"|"ge"), $not_pattern_value             |
        "between", [ $not_pattern_value, $not_pattern_value ] |
        ("in"|"any"|"all"|"exactly"), $array_value
      )
]

$basic_value = @{not} (
                       { // : any * } |
                       [ any * ]      |
                       null
                      )

$not_pattern_value = @{not} (
                             { // : any * } |
                             [ any * ]      |
                             null           |
                              $pattern_value
                            )

$pattern_value = /^[^\*]*\*[^\*]*$/

$array_value = [ $not_pattern_value + ]
```

Here, in the following, some examples:

```
https://rdap.pubtest.nic.it/domains?name=we*.it&filter=["registrationDate
","ge","2018-01-20"]

https://rdap.pubtest.nic.it/domains?name=we*.it&filter={"or":[["registrat
ionDate","ge","2018-01-20"],["expirationDate","le","2019-01-20"]]}

https://rdap.pubtest.nic.it/domains?name=we*.it&filter={"not":{"or":[["re
gistrationDate","ge","2018-01-20"],["expirationDate","le","2019-01-
20"]]}}
```

In addition to the JCR constraints, some further constraints must be applied:

- all values in an array must have the same type;
- *any*, *all* or *exactly* operators must be associated only to an RDAP property whose type is an array of primitive types (like status, roles).

All predicates in an array of predicates are implicitly combined by AND. Therefore, the two expressions below have the same meaning:

```
{"and":[["registrationDate","ge","2018-01-
20"],["expirationDate","le","2019-01-20"]]}
```

```
[["registrationDate","ge","2018-01-20"],["expirationDate","le","2019-01-
20"]]
```

In the same way as the array of predicates is a shortcut for the definition of two or more predicates combined by AND, the *in* operator is a shortcut for the definition of a list of predicates, each one referencing the same property, combined by OR.

The *isnull* and *isnotnull* operator are used in those cases where the predicate would express, respectively, the absence or the presence of a property in the expected results. For these operators, the last item of the predicate array can be missing and, if present, must be ignored. For example, to receive only domains that have never been transferred, a predicate could be:

```
["transferDate","isnull"]
```

The *any*, *all* and *exactly* operators are used with the following meaning:

- *any* means that the property must contain at least one of the values in the array;
- *all* means that the property must contain all the values in the array, but it could contain also additional values;
- *exactly* means that the property must contain all the values in the array and cannot contain any additional value.

If an array contains only one value, *any* and *all* have the same meaning.

The implementation of the new parameter is technically feasible, as operators for filtering rows according to a search condition are currently supported by Relational as well as NoSQL DBMSs. The impact on the current state of the RDAP standard is limited to the search query format.

Finally, some considerations must be made:

- RDAP does not specify any mandatory object property. Except for *objectClassName* and the property identifying uniquely an object (e.g. *ldhName* for domain), all other properties are optional. Therefore, a client could require a filter including a reference to a property that is not implemented. In this case, the server is recommended to provide the client with an error response.
- The filter parameter allows to restrict the result set returned by a search query. Therefore, basically, it adds additional predicates on the initial condition expressed by the search pattern. To achieve the maximum flexibility, patterns could be wildcards and the search conditions could be expressed entirely by the filter parameter value. Otherwise, filter might be taken as a new segment path and the current search patterns might be included among the property references, like in the following:

  ```
  domains?filter={"or":[["name","eq","wha*"],["name","eq","whi*"]]}
  ```

- The present document reports only some suitable properties of the topmost objects in the search results array. Obviously, they can be extended with other properties that have not yet been considered. For example, the search patterns not yet appearing among the property references could be taken into account (e.g. *nsLdhName* and *nsIp* for domains).
- Servers could implicitly filter results according to user access levels. For example, registrar users could search only their own domains.
- Some characters, which cannot be part of a URL (for example, the space or the character `‘+’` in datetime values expressed as local date plus offset), must be encoded (for example, respectively, `‘%20’` and `‘%2B’`).