Enhancing RDAP filtering capabilities

Mario Loffredo, Maurizio Martinelli

IIT-CNR/Registro.it

Contents

- REST services
- RDAP status
- .it proposals to IETF RegExt WG
- Reasons for filtering
- “filter” parameter
- Short demo
- Considerations
- Future developments
- Q & A

REST services

- REST services SHOULD offer capabilities for efficient management of result sets:
  - filtering
  - sorting
  - paging
  - subsetting

- Reasons:
  - minimizing the bandwidth usage
  - speeding up the response time
  - improving the precision of the queries and, consequently, obtain more reliable results
  - decreasing CPU time and memory spent on both server and client

RDAP status

- RDAP provides limited search capabilities (RFC 7482)
  - the search condition consists of a single predicate
- A search query can potentially generate a large result set
- The result set:
  - must be scrolled when looking for the desired data (best case scenario)
  - can be truncated according to the server limits (worst case scenario)
- RDAP lacks of result filtering, sorting, paging, and subsetting capabilities:
  - you cannot restrict the result set by adding search conditions
  - you cannot specify possible sort criteria to have the most relevant objects at the beginning of the result set
  - you cannot scroll the result set by subsequent queries when the result set is truncated
  - you cannot request for a partial response

.it proposals to IETF RegExt WG

- Two I-Ds about managing large RDAP responses:
  - *I-D.loffredo-regext-rdap-sorting-and-paging*
    Loffredo, M., Martinelli, M., and S. Hollenbeck, "Registration Data Access Protocol (RDAP) Query Parameters for Result Sorting and Paging", draft-loffredo-regext-rdap-sorting-and-paging-03, March 2018
  - *I-D.loffredo-regext-rdap-partial-response*
    Loffredo, M. and M. Martinelli, "Registration Data Access Protocol (RDAP) Partial Response", draft-loffredo-regext-rdap-partial-response-01, March 2018
- One I-D about reverse search:
  - *I-D.loffredo-regext-rdap-reverse-search*
    Loffredo, M. and M. Martinelli, "Registration Data Access Protocol (RDAP) Reverse Search", draft-loffredo-regext-rdap-reverse-search-01, March 2018

Reasons for filtering

- The extraction of the desired information from a RDAP response could be time and resource consuming
  - even if sorting, paging and subsetting would be implemented
- Users can obtain exactly what they are searching for
- If pagination is not implemented, filtering can avoid the loss of relevant results due to truncation
- Users might be interested in performing searches that are currently unsupported:
  - a registrar might search its own domains for a certain status or for a specific event in a range of dates
  - a law authority might search all the contacts for a specific email

“filter” parameter - principles

- Parameter:
  - Name: **filter**
  - Value: a search condition
- How to represent the value?
  - traditionally, a search condition includes a set of predicates combined by logical operators AND, OR and NOT
  - a predicate contains three components:
    - a property name;
    - an allowed operator for the property;
    - a value (or a list of values) whose type is allowed for the property
- The value can be represented as a JSON expression
  - JSON can represent search conditions whose complexity ranges from very simple to extremely complicated
  - JSON is both human-readable and machine-processable

“filter” parameter - Predicate properties

- The properties already defined in *I-D.loffredo-regext-rdap-sorting-and-paging* can be used in a predicate:
  - *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
- “status” and “roles” should be also considered
- “name” vs. “ldhName“, “unicodeName”?

“filter” parameter - Predicate values

- Basic type:
  - string
  - number
  - boolean
  - datetime
    - RFC3339 full-date and date-time formats are considered
- Array of a basic type

“filter” parameter - Predicate operators

- Operators for properties whose type is a basic type:
  - **no values:** isnull, isnotnull
  - **one value:** eq, ne, le, ge, lt, gt
  - **array of two values:** between
  - **array of N values:** in
  - Specific operators on strings (e.g. “contains”, “starts with”) can be implemented using **eq**/**ne** operators and the wildcard
- Operators for properties (such as status) whose type is an array:
  - ***any***: the property must contain at least one of the values in the array
  - ***all***: the property must contain all the values in the array, but it could also contain additional values
  - ***exactly***: the property must contain all the values in the array and cannot contain additional values
- Operators for predicates:
  - **one predicate:** not
  - **N predicates:** and, or

“filter” parameter - The value in JSON

- A simple predicate consists of a JSON array:
  - the number of items ranges from 2 (operators without value) to 3 (operators with value):
    - [“lastChangedDate”, “isnull”]
    - [“registrationDate”, “gt”, “2018-01-20”]
    - [“registrationDate”, “between”, [“2018-01-20”,“2018-01-21”]]
    - [“country”, “in”, [“it”,“ch”,“de”,“fr”]]
  - deserialization of a JSON array into an object:
    - it is not a standard capability of JSON libraries
    - it can be implemented through a few lines of code
    - a JSON array is more compact than a JSON object

- A complex condition consists of a JSON object, including a single member:
  - the logical operator is the member name
  - the sub-predicates (one or more) are the member values
    - {"or":[["registrationDate","ge","2018-01-20"],["expirationDate","le","2019-01-20"]]}
    - {"not":{"or":[["registrationDate","ge","2018-01-20"],["expirationDate","le","2019-01-20"]]}}

“filter” parameter - The value in JCR

```
@{root} $expression = {
 (
  $or_expression     |
  $and_expression    |
  $not_expression    |
  $predicates_array  |
  $predicate
 )
}

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

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

$not_expression = {
"not" : $expression
}

$predicates_array = [ $predicate + ]
```

```
$predicate = [
 /^[A-Za-z]+$/,
 (
  ("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 + ]
```

“filter” parameter - To be noted

- *isnull* and *isnotnull* are used when the predicate represents, respectively, the absence or the presence of a property in the expected results
  - ["transferDate","isnull"]
- All predicates in an array are implicitly combined by "and"
  - {"and":[["registrationDate","ge","2018-01-20"],["expirationDate","le","2019-01-20"]]}
  - [["registrationDate","ge","2018-01-20"],["expirationDate","le","2019-01-20"]]
- The operator "between" is a shortcut for two predicates combined by "and" including the same property
  - {"and":[["registrationDate","ge","2018-01-20"],["registrationDate","le","2019-01-20"]]}
  - ["registrationDate","between",["2018-01-20","2019-01-20"]]
- The operator "in" is a shortcut for N predicates combined by "or" including the same property and the "eq" operator
  - {"or":[["country","eq","it"],["country","eq","ch"],["country","eq","de"], ["country","eq","fr"]]}
  - [“country”, “in”, [“it”,“ch”,“de”,“fr”]]

Short demo

- Search domains whose name starts with "w"
  - https://rdap.pubtest.nic.it/domains?name=w*.it
- How many are there ?
  - https://rdap.pubtest.nic.it/domains?name=w*.it&count=1
- Which is the oldest ?
  - https://rdap.pubtest.nic.it/domains?name=w*.it&count=1&sortby=registrationDate
- What are the domains registered since 2015 ?
  - https://rdap.pubtest.nic.it/domains?name=w*.it&count=1&sortby=registrationDate&filter=["registrationDate","gt","2015-01-01"]
- What are the inactive domains registered since 2015 ?
  - https://rdap.pubtest.nic.it/domains?name=w*.it&count=1&sortby=registrationDate&filter=[["registrationDate","gt","2015-01-01"],["status","any",["inactive"]]]
- Return only the domain names sorted by LDH name
  - https://rdap.pubtest.nic.it/domains?name=w*.it&count=1&sortby=ldhName&filter=[["registrationDate","gt","2015-01-01"],["status","any",["inactive"]]]&fieldSet=id

Implementation considerations

- The implementation of the filter parameter is technically feasible
  - operators for filtering results are supported by DBMSs
  - the impact on RDAP is limited to the search query format

- Additional technical considerations:
  - almost all properties in RDAP are optional
    - if a predicate includes an unimplemented property, an error should be returned
  - the filter parameter adds further conditions to the search pattern, to increase flexibility:
    - the pattern could be wildcard and search conditions could be described entirely by the filter value
    - otherwise, the filter parameter might be taken as a new segment path
      - domains?filter={"or":[["ldhName","eq","wha*"],["ldhName","eq","whi*"]]}
  - most suitable properties of the topmost objects have been reported in predicates
    - they can be extended with other properties that have not been considered yet
  - servers could implicitly filter results according to user access levels:
    - the implicit filter can be represented in the same way as the explicit filter
    - final filter = {"and":[<implicit filter>,<explicit filter>]}
  - some characters in predicate values must be encoded to have URL-safe queries
    - blank encoded as '%20', '+' encoded as '%2B'

Security considerations

- Search queries typically require more server resources than lookup queries
- This increases the risk of server resource exhaustion and subsequent denial of service due to abuse
- Risks can be mitigated by:
  - limiting the rate of search requests
  - truncating and paging results
  - requesting a partial response
  - enhancing filtering capabilities

Future developments

- RDAP servers can provide different capabilities:
  - some query paths cannot be available
  - bootstrapping is not implemented
  - queries can be extended with additional parameters
  - authentication and access levels can be implemented
  - responses can contain proprietary extensions

- How could RDAP clients face with such a diversity?

- Proposal:
  - servers could provide their own policies via a REST API specification format
    - **OpenAPI**, RAML, API Blueprint, JSON API, JSON Schema
      - https://rdap.pubtest.nic.it/specification
    - Bootstrapping can help find the desired specification (**e.g.** draft-ietf-regext-rdap-object-tag-02)
      - https://rdap.pubtest.nic.it/specification/VRSN
      - https://rdap.pubtest.nic.it/specification/BRNIC
      - https://rdap.pubtest.nic.it/specification/GOOGLE
  - clients could automatically configure themselves
    - http://petstore.swagger.io/

Registro.it

Thanks for your attention!

Q & A

Consiglio Nazionale delle Ricerche

Istituto di Informatica e Telematica