

# SOAP API Design Principles

This page should be guideline for everyone, who creates a new service / API function. All patterns used in Atollon SOAP API should be mentioned here so we do not solve one goal by different implementations.

## Fair usage policy

Every atollon integration, which behaves like a logged user (uses atollon session) should not send more than 2 requests in parallel. This helps the atollon to provide quality of service to all users even in peak usage and extensive operations (like copying big sets of data and so on).

## Updating Atollon objects

There are some principles to understand when working with (or building) atollon API. Understanding of those principles may prevent accidental data loss, so you better bare them in mind.

1. Before updating atollon object, you need to have valid copy of whole object before updating. Update function typically reads all editable parameters and replace them. That is why you need to send all of them, because otherwise the empty ones would be considered to be deleted.
2. Some Get methods do not return pure atollon object, but extended one. Typically extended by selected or derived fields from another object. The reason behind is the reduction the of request amount. Good example is GetFolder, which returns primary contact details (name, surname, primary mobile, primary email etc.). Those extended parameters should not be a part of the update request, because server ignores derived fields in update functions (if you send them, they will be ignored anyway).
3. There are two ways how the relation between two atollon objects is maintained. The selected solution usually depends on the bussiness importance of the relation or the workflow and can be tracked in the Atollon Lagoon, which is etalon of API usage.
  1. Edit ID of related object in update function.
  2. Separate API function call, which creates the relation.

## FILTER in listing functions

Unified filter desing adds high value to listing function with low amount of time needed to implement. This paragraph focuses on those functions, which support filtering by FILTER.

# Example

```
<FILTER>
  <COL>
    <name>draftType</name>
    <type>equals</type>
    <value>1</value>
    <negation>true</negation>
  </COL>
  <COL>
    <name>providerName</name>
    <type>contains</type>
    <value>Ato1</value>
  </COL>
</FILTER>
```

Filter contains list of conditions, which defines values we want to have in the result. Each condition consists of 4 elements.

**name** - Defines which field we want to filter on. It is Enum of possible parameter names. Since we can not use Enum type (mooring request translation does not support this type) atm, write possible names into comment!

**type** - Defines the type of comparison used in condition. It is an Enum of possible values. All functions does not have to implement all types of comparison, in case they do not, please mention the supported ones in documentation.

Possible values are:

1. equals - selected rows have to find pure match (column = value)
2. contains - defines usage of ILIKE %value% comparison
3. regular - request contains regular expression (*not implemented yet*)
4. in - set of values (eg. usage of IDs and so) (*currently implemented for comma-separated integer values only*) ( column=ANY(E'{value}') )
5. any - usable for testing if value is included in the array type column ( value=ANY(column) )
6. range - range of values (1-100, 2016-08-23T00:00:00 - 2016-09-23T00:00:00) (*not implemented yet*)
7. mask - integer bitmap mask (column & mask <> 0)
8. less - numerical comparison (column < value)
9. lessOrEquals - numerical comparison (column <= value)
10. greater - numerical comparison (column > value)
11. greaterOrEquals - numerical comparison (column >= value)

**value** - string representation of value (yes, it is kinda sad that we can not verify type on the WSDL level, even if it is just simple int, but that is the price for the variability in minimum amount of attributes).

**negation** - allows add NOT before the any type condition

# ORDER in listing functions

Ordering is pretty important feature in terms of end user usage comfort. That is why every function should contain this option. I will personally find and punish every single guy, who does not use ordering enabled o client side on collections, which support the ordering!

## example

```
<ORDER>
  <COL>
    [name>created</name>
    [desc>true</desc>
  </COL>
  <COL>
    [name>total</name>
    [desc>true</desc>
  </COL>
</ORDER>
```

**name** - Defines which field are we going to order on. It is Enum of possible field names.

**desc** - pretty self-explanatory, but for sure - Boolean which defines if we want to use ascending (false) or descending (true) sorting.

---

Revision #3

Created 30 October 2024 18:29:27 by Jan Safka

Updated 30 October 2024 18:31:22 by Jan Safka