Searching and Filtering

This document explains advanced searching and filtering over the API.

Why is search implemented as PUT

REST APIs usually implement search as a GET request and Fulfil's basic search also workes over a GET and you can use URL parameters to set filters. However, fuflil uses PUT calls formore advanced search

```
<ul>
    <li>(nested OR/AND clauses, LIKE, NOT LIKE) cannot be easily specified as
    URL parameters. They ae better encoded in the body of a request</li>
    <li>URLs are logged across many parts of the infrastructure and
    search parameters may include PII. Sending them in body avoids PII
    being logged in load balancers and such infra components</li>
</ul>
</p>
```

Features of advanced search

`filters`

A filter expression is a JSON Array that is the equivalent of a SQL where clause. Each clause in a where would be a JSON list where the

- first element is the field
- second element is the operator
- third element is the value

`limit` & `offset`

Similar to LIMIT/FETCH clause in SQL and the OFFSET/SKIP in SQL.

`order`

Controls the order in which the records should be returned. This is also a list of arrays

```
[
    ["name", "ASC"],
    ["age", "DESC"]
]
```

Example of an advanced search

The below example sends a PUT request to the
product resource

to filter.

```
PUT https://apc.fulfil.io/api/v2/model/product.product/search

{
    "filters": [
        ["code", "ilike", "APL-%"]
    ]
}
```

The response would be an array of all the ids of records that match the given filter.

```
[
    12, 13, 14
]
```

Pagination

While the simple GET request accepts `page` and `per_page`, the advanced search endpoints do not support it. Instead, the advanced endpoints follow a sql style `OFFSET` and `LIMIT` (also called `FETCH` in some sql flavors).

- The `OFFSET` clause specifies the number of records to skip before starting to return record from the search result. Default: 0
- The `LIMIT` clause specifies the number of records to return. The default is `null` which returns all records, but could be extremely slow for large models.

Example, where we skip the first 10 records and returns the next 20.

```
PUT https://apc.fulfil.io/api/v2/model/product.product/search

{
    "filters": [
        ["code", "ilike", "APL-%"]
    ],
    "offset": 10,
    "limit": 20
}
```

Ordering

The advanced searching and filtering API also offers a powerful way to sort records. Similar to SQL, the sort argument accepts a field name and the direction of sort.

In this example, the results are sorted by created date in the ascending order and the sorted in the reverse order on product code.

```
{
    "filters": [
        ["code", "ilike", "APL-%"]
    ],
    "offset": 10,
    "limit": 20
    "order": [
        ["create_date", "ASC"],
        ["code", "DESC"]
    ]
}
```

Counting

Sometime, we don't need the exact records or their ids, but all what you need is a count of the records that satisfy the advanced search/filter.

For this fulfil provides another endpoint on every model to perform an advanced search, but only return the count.

```
PUT https://apc.fulfil.io/api/v2/model/product.product/search_count

{
    "filters": [
        ["code", "ilike", "APL-%"]
    ]
}
```

Search & Read in a single API call

While the above API call is useful, in most cases you want to read attributes of the records returned by the endpoint. This creates an additional API call and slows down the user experience.

This can be avoided by using the `**search_read**` endpoint which filters exactly the way search does, but instead of just returning ids, you can request specific fields to be returned (including the id).

For example, the below request combines the search and read into a single API call to Fulfil.

```
PUT https://apc.fulfil.io/api/v2/model/product.product/search_read

{
    "filters": [
        ['code', 'ilike', 'APL-%']
    ],
    "fields": [
        'variant_name',
        'code',
        'tempalte_name',
        'default_uom.symbol'
    ]
}
```

Field Filter Expressions

Field filter expressions are used when you are searching for records using the find or search APIs.

Example:

```
[
    ['first_name', '=', 'Steve']
]
```

The above expression has 1 filter condition, while the below one has 2.

```
[
    ['first_name', '=', 'Steve'],
    ['last_name', '=', 'Jobs'],
]
```

The most common pattern of a filter condition has a `**field_name**`, an `**operand**` and a `**value**`.

This section assumes basic knowledge of SQL and tries to explain the use of the filter expressions by the use of WHERE clause of equivalent SQL queries.

The above expression would then convert to:

```
SELECT ..
WHERE
    first_name = 'Steve'
    AND
    last_name = 'Jobs'
```

The filter conditions can be combined with `**AND**` or `**OR**` operators. When nothing is specified, it implies `**AND**`.

```
[
    'OR',
    [
        ['first_name', '=', 'Steve'],
    ],
    [
        ['first_name', '=', 'Tim'],
    ],
]
```

Roughly translates to this SQL:

```
SELECT ..
WHERE
first_name='Steve'
OR
first_name='Tim';
```

You can also combine `**AND**` and `**OR**`.

```
[
    'OR',
    [
        ['first_name', '=', 'Steve'],
        ['company', '=' 'Apple'],
    ],
    [
        ['first_name', '=', 'Larry'],
        ['company', '=' 'Google'],
    ],
]
```

Roughly translates to this SQL:

```
SELECT ..
WHERE
(first_name='Steve' AND company='Apple')
OR
(first_name='Larry' AND company='Google')
```

Operators

Exact match (`=`)

An “exact” match. For example:

```
['id', '=', 1]
```

Roughly translates to this SQL:

```
SELECT .. WHERE id=1;
```

Case Insensitive match (`ilike`)

A case-insensitive match. So, the filter:

```
['first_name', 'ilike', 'Steve']
```

Roughly translates to this SQL:

```
SELECT .. WHERE first_name ILIKE 'Steve';
```

You can also use wildcards in the search. So, the filter:

```
['first_name', 'ilike', '%Steve%']
```

Roughly translates to this SQL:

```
SELECT .. WHERE first_name ILIKE '%Steve%';
```

Only String (Char) fields support the `ilike` operand. You can also use `not ilike` to exclude results that match a pattern.

Case Sensitive match (`like`)

A case-insensitive match. So, the filter:

```
['first_name', 'like', 'Steve']
```

Roughly translates to this SQL:

```
SELECT .. WHERE first_name LIKE 'Steve';
```

You can also use wildcards in the search.

So, the filter:

```
['first_name', 'like', '%Steve%']
```

Roughly translates to this SQL:

```
SELECT .. WHERE first_name LIKE '%Steve%';
```

Only String (Char) fields support the `ilike` operand. You can also use `not like` to exclude results that match a pattern.

Comparison Operators (`>`, `>=`, `<`, `<=`, `!=`)

So, the filter:

```
['amount', '>=', 100]
```

Roughly translates to this SQL:

```
SELECT .. WHERE amount >= 100;
```

The operand is supported by:

- Numeric/Currency fields
- Integer fields
- Float fields
- Date fields
- Datetime fields

IN and NOT IN

```
['state', 'in', ['done', 'processing']]
```

Roughly translates to this SQL:

```
SELECT ..
WHERE state IN ('done', 'processing');
```

```
['state', 'not in', ['done', 'processing']]
```

Roughly translates to this SQL:

```
SELECT ..
WHERE state NOT IN ('done', 'processing');
```

Lookups that span relationships

Fulfil offers a powerful and intuitive way to “lookup” nested data in related models. In SQL terms this would be like a join. For example, if you had the ID of a sale order and you wanted to pull all lines with that order, this would look like:

```
['sale.number', '=', 'SO1234']
```

Roughly translates to this SQL:

```
SELECT ..
FROM sale_line
JOIN sale_order ON ..
WHERE
    sale_order.number = 'SO1234';
```

All of the above operands are supported in a nested lookup.