Documentation

JSON:API module

Filtering

Discuss

Last updated on 25 August 2022

Collections are listings of resources. In a decoupled site, they're what you use to create things like a "New Content" list or "My content" section on the client-side.

However, when you make an unfiltered request to a collection endpoint like `/jsonapi/node/article`, you'll just get *every* article that you're allowed to see.

Without filters, you can't get only *your* articles or only articles *about llamas.*

This guide will teach you how to build filters like a pro.

Quick Start

The simplest, most common filter is a key-value filter:

```
?filter[field_name]=value&filter[field_other]=value
```

This matches all resources with "field_name" equal to "value" and "field_other" equal to "value".

For everything else, read on!

Summary

The JSON:API module has some of the most robust and feature-rich filtering features around. All of that power comes with a bit of a learning curve though.

By the end of this article, you'll be able make complex queries and reason about problems you may face, like "how do I get a list of an author's articles about llamas *or* the worlds fastest member of the animal kingdom, the peregrine falcon?"

We're going to work our way up from the very basics. After that, we'll show you a few shortcuts that make writing filters a little faster and less verbose. Finally, we'll see a number of filter examples taken from the real world.

If you're not a newcomer to Drupal, you've probably already used the Views module for these kinds of things. **Unlike the REST module that comes with Drupal Core, JSON:API does not export Views results.** Collections are JSON:APIs API-First replacement for exported "REST displays" in Views.

Building Filters

The fundamental building blocks of JSON:API's filters are *conditions* and *groups*. Conditions assert that something is true and groups let you compose those assertions into logical sets to make bigger condition groups. Those sets can be nested to make super fine queries. You can think of those nested sets like a tree:

```
Conventional representation:

a( b() && c( d() || e() ) )

Tree representation:

   a
  / \
 b & c
    / \
   d | e

In both representations:

"d" and "e" are members of "c" in an OR group.
"b" and "c" are members of "a" in an AND group.
```

So, what's inside a condition?

Let's get logical 🖐. Remember, a condition tells you a TRUE or FALSE thing about a resource and some assertion you make about it, like "was this entity created by a particular user?" When the condition is FALSE for a resource, that resource won't be included in the collection.

A condition has 3 primary parts: a **path**, an **operator** and a **value**.

- A 'path' identifies a field on a resource
- An 'operator' is a method of comparison
- A 'value' is the thing you compare against

In pseudo-code, a condition is a thing that looks like this:

```
($field !== 'space')
```

Where:

1. `$field` is the field of the resource identified by its 'path'
2. the 'operator' is `!==`
3. the 'value' is the string `'space'`

In the JSON:API module, we can't make it look as pretty as that because we need to make it work inside a URL query string. To do that, **we represent each condition with key/value pairs.**

If we were filtering on a user's first name, a condition might look something like this:

```
?filter[a-label][condition][path]=field_first_name
&filter[a-label][condition][operator]=%3D  <- encoded "=" symbol
&filter[a-label][condition][value]=Janis
```

Notice that we put a label inside the first set of square brackets. We could easily have made it `b-label` or `this_is_my_super_awesome_label` or even an integer like `666` 🤘😅. The point is that **every condition or group should have an identifier.**

But what if we have *lots* of Janises in the system?

Let's add another filter so we only get Janises with a last name that starts with "J":

```
?filter[first-name-filter][condition][path]=field_first_name
&filter[first-name-filter][condition][operator]=%3D  <- encoded "="
&filter[first-name-filter][condition][value]=Janis

&filter[last-name-filter][condition][path]=field_last_name
&filter[last-name-filter][condition][operator]=STARTS_WITH
&filter[last-name-filter][condition][value]=J
```

Maybe the plural of Janis is "Janii" 🤔 ...

There are many more filter operators than just `=` and `STARTS_WITH`. Here's the complete list that's taken right from the JSON:API codebase:

```
\Drupal\jsonapi\Query\EntityCondition::$allowedOperators = [
  '=', '<>',
  '>', '>=', '<', '<=',
  'STARTS_WITH', 'CONTAINS', 'ENDS_WITH',
  'IN', 'NOT IN',
  'BETWEEN', 'NOT BETWEEN',
  'IS NULL', 'IS NOT NULL',
];
```

Symbol operators need to be url-encoded. You can get the correct encoding with PHP's `urlencode()` function.

Condition Groups

Now we know how to build conditions, but we don't yet know how to build *groups of conditions*. How do we build a tree like we saw above?

In order to do that, we need to have a "group". A group is a set of conditions joined by a "conjunction." **All groups have conjunctions and a conjunction is either AND or OR.**

Now our filter is a bit *too* specific! Let's say we want to find all users with a last name that starts with "J" and either have the first name "Janis" *or* the first name "Joan".

To do that, we add a group:

```
?filter[rock-group][group][conjunction]=OR
```

Then, we need assign our filters to that new group.

To do that, we add a `memberOf` key. **Every condition and group can have a `memberOf` key.**

Tip: Groups can have a `memberOf` key just like conditions, which means we can have groups of groups 🤯!

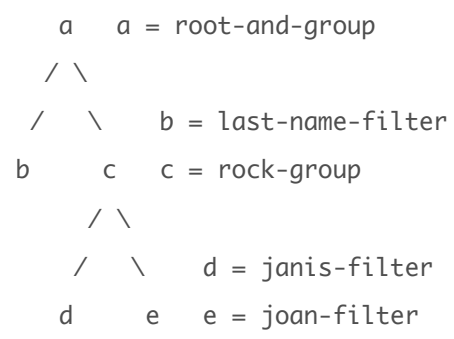
Note: Every filter without a `memberOf` key is assumed to be part of a "root" group with a conjunction of AND.

All together now:

```
?filter[rock-group][group][conjunction]=OR

&filter[janis-filter][condition][path]=field_first_name
&filter[janis-filter][condition][operator]=%3D
&filter[janis-filter][condition][value]=Janis
&filter[janis-filter][condition][memberOf]=rock-group

&filter[joan-filter][condition][path]=field_first_name
&filter[joan-filter][condition][operator]=%3D
&filter[joan-filter][condition][value]=Joan
&filter[joan-filter][condition][memberOf]=rock-group

&filter[last-name-filter][condition][path]=field_last_name
&filter[last-name-filter][condition][operator]=STARTS_WITH
&filter[last-name-filter][condition][value]=J
```

Does that look familiar?

It should, we saw it above as a tree:



You can nest these groups as deeply as your heart desires.

Paths

Conditions have one last feature: 'paths'

Paths provide a way to filter based on relationship values.

Up to this point, we've just been filtering by the hypothetical `field_first_name` and `field_last_name` on the user resource.

Let's imagine that we want to filter by the name of a user's career, where career types are stored as a separate resource. We could add a filter like this:

```
?filter[career][condition][path]=field_career.name
&filter[career][condition][operator]=%3D
&filter[career][condition][value]=Rockstar
```

Paths use a "dot notation" to traverse relationships.

If a resource has a relationship, you can add a filter against it by concatenating the relationship field name and the *relationship's* field name with a `.` (dot).

You can even filter by relationships of relationships (and so on) just by adding more field names and dots.

Tip: You can filter on a specific index of a relationship by putting a non-negative integer in the path. So the path `some_relationship.1.some_attribute` would only filter by the 2nd related resource.

Tip: You can filter by sub-properties of a field. For example, a path like `field_phone.country_code` will work even though `field_phone` isn't a relationship.

Tip: When filtering against configuration properties, you can use an asterisk (*) to stand-in for any portion of a path. For example, `/jsonapi/field_config/field_config?filter[dependencies.config.*]=comment.type.comment` would match all field configs in which

`["attributes"]["dependencies"]["config"]` (an indexed array) contains the value "comment.type.comment".

Shortcuts

That's a lot of characters to type. Most of the time, you don't need such complicated filters and for those cases, the JSON:API module has a few "shortcuts" to help you write filters faster.

When the operator is `=`, you don't have to include it. It's just assumed. Thus:

```
?filter[a-label][condition][path]=field_first_name
&filter[a-label][condition][operator]=%3D  <- encoded "=" symbol
&filter[a-label][condition][value]=Janis

becomes

?filter[janis-filter][condition][path]=field_first_name
&filter[janis-filter][condition][value]=Janis
```

It's also rare that you'll need to filter by the same field twice (although it's possible). So, when the operator is `=` and you don't need to filter by the same field twice, the path can *be* the identifier. Thus:

```
?filter[janis-filter][condition][path]=field_first_name
&filter[janis-filter][condition][value]=Janis

becomes

?filter[field_first_name][value]=Janis
```

That extra `value` is pesky. That's why you can reduce the simplest equality checks down to a key-value form:

```
?filter[field_first_name]=Janis
```

Filters and Access Control

First, a warning: don't make the mistake of confusing filters for access control. Just because you've written a filter to remove something that a user shouldn't be able to see, doesn't mean it's not accessible. **Always perform access checks on the backend.**

With that big caveat, let's talk about using filters to complement access control. To improve performance, you should filter out what your users *will not* be able to see. The most frequent support request in the JSON:API issue queues can be solved by this one simple trick!

If you know your users cannot see unpublished content, add the following filter:

```
?filter[status][value]=1
```

Using this method, you'll lower the number of unnecessary requests that you need to make. That's because **JSON:API doesn't return data for resources to which a user doesn't have access.** You can see which resources may have been affected by inspecting the `meta.errors` section of JSON:API document.

So, **do your best to filter out inaccessible resources ahead of time**.

Filter Examples

1. Only get published nodes

A very common scenario is to only load the nodes that are published. This is a very easy filter to add.

```
SHORT
filter[status][value]=1

NORMAL
filter[status-filter][condition][path]=status
filter[status-filter][condition][value]=1
```

2. Get nodes by a value of a entity reference

A common strategy is to filter content by a entity reference.

```
SHORT
filter[uid.id][value]=BB09E2CD-9487-44BC-B219-3DC03D6820CD

NORMAL
filter[author-filter][condition][path]=uid.id
filter[author-filter][condition][value]=BB09E2CD-9487-44BC-B219-3DC03D6820CD
```

To fully comply with the JSON API specification, while Drupal internally uses the `uuid` property, JSON API uses `id` instead.

Since Drupal 9.3 it is possible to filter on `target_id` also instead of only filtering by `uuid` property.

```
SHORT
filter[field_tags.meta.drupal_internal__target_id]=1

NORMAL
filter[name-filter][condition][path]=field_tags.meta.drupal_internal__target_id
filter[name-filter][condition][value]=1
```

3. Nested Filters: Get nodes created by user admin

It's possible to filter on fields from referenced entities like the user, taxonomy fields or any entity reference field. You can do this easily but just using the the following notation. reference_field.nested_field. In this example the reference field is uid for the user and name which is a field of the user entity.

```
SHORT
filter[uid.name][value]=admin

NORMAL
filter[name-filter][condition][path]=uid.name
filter[name-filter][condition][value]=admin
```

4. Filtering with arrays: Get nodes created by users [admin, john].

You can give multiple values to a filter for it to search in. Next to the field and value keys you can add an operator to your condition. Usually it's "=" but you can also use "IN", "NOT IN", ">", "<", "<>", BETWEEN".

For this example we're going to use the IN operator. Note that I added two square brackets behind the value to make it into an array.

```
NORMAL
filter[name-filter][condition][path]=uid.name
filter[name-filter][condition][operator]=IN
filter[name-filter][condition][value][1]=admin
filter[name-filter][condition][value][2]=john
```

Tip: when utilizing square brackets for multiple values filters, do not just use empty square brackets for a new value.
While these work when typed into the URL, Guzzle and other http clients will only create one value as the array key will be seen to be same and override the previous value. It is better to use an index to create unique array elements.

5. Grouping filters: Get nodes that are published and create by admin.

Now let's combine some of the examples above and create the following scenario.
WHERE user.name = admin AND node.status = 1;

```
filter[and-group][group][conjunction]=AND
filter[name-filter][condition][path]=uid.name
filter[name-filter][condition][value]=admin
filter[name-filter][condition][memberOf]=and-group
filter[status-filter][condition][path]=status
filter[status-filter][condition][value]=1
filter[status-filter][condition][memberOf]=and-group
```

You don't really have to add the and-group but I find that a bit easier usually.

6. Grouping grouped filters: Get nodes that are promoted or sticky and created by admin

Like mentioned in the grouping section, you can put groups into other groups.
WHERE (user.name = admin) AND (node.sticky = 1 OR node.promoted = 1)

To do this we put sticky and promoted into a group with conjunction OR. Create a group with conjunction AND and put
the admin filter, and the promoted/sticky OR group into that.

```
# Create an AND and an OR GROUP
filter[and-group][group][conjunction]=AND
filter[or-group][group][conjunction]=OR

# Put the OR group into the AND GROUP
filter[or-group][group][memberOf]=and-group

# Create the admin filter and put it in the AND GROUP
filter[admin-filter][condition][path]=uid.name
filter[admin-filter][condition][value]=admin
filter[admin-filter][condition][memberOf]=and-group

# Create the sticky filter and put it in the OR GROUP
filter[sticky-filter][condition][path]=sticky
filter[sticky-filter][condition][value]=1
filter[sticky-filter][condition][memberOf]=or-group

# Create the promoted filter and put it in the OR GROUP
filter[promote-filter][condition][path]=promote
filter[promote-filter][condition][value]=1
filter[promote-filter][condition][memberOf]=or-group
```

7. Filter for nodes where 'title' CONTAINS "Foo"

```
SHORT
filter[title][operator]=CONTAINS&filter[title][value]=Foo
```

```
NORMAL
filter[title-filter][condition][path]=title
filter[title-filter][condition][operator]=CONTAINS
filter[title-filter][condition][value]=Foo
```

8. Filter by non-standard complex fields (e.g. addressfield)

```
FILTER BY LOCALITY
filter[field_address][condition][path]=field_address.locality
filter[field_address][condition][value]=Mordor

FILTER BY ADDRESS LINE
filter[address][condition][path]=field_address.address_line1
filter[address][condition][value]=Rings Street
```

9. Filtering on Taxonomy term values (e.g. tags)

For filtering you'll need to use the machine name of the vocabulary and the field which is present on your node.

```
filter[taxonomy_term--tags][condition][path]=field_tags.name
filter[taxonomy_term--tags][condition][operator]=IN
filter[taxonomy_term--tags][condition][value][]=tagname
```

10. Filtering on Date (Date only, no time)

Dates are filterable. Pass a time string that adheres to the ISO-8601 format.

This example is for a Date field that is set to be date only (no time).

```
filter[datefilter][condition][path]=field_test_date
filter[datefilter][condition][operator]=%3D
filter[datefilter][condition][value]=2019-06-27
```

This example is for a Date field that supports date and time.

```
filter[datefilter][condition][path]=field_test_date
filter[datefilter][condition][operator]=%3D
filter[datefilter][condition][value]=2019-06-27T16%3A00%3A00
```

Note that timestamp fields (like created or changed) currently must use a timestamp for filtering:

```
filter[recent][condition][path]=created
filter[recent][condition][operator]=%3D
filter[recent][condition][value]=1591627496
```

11. Filtering on empty array fields

This example is for a Checkboxes/Radio buttons field with no value selected. Consider you have a field that is a checkbox. You would like to get all nodes that do not have that value checked. When checked, the JSON API returns an array:

```
"my_field":["checked"]
```

When unchecked, the JSON API returns an empty array:

```
"my_field": []
```

If you would like to get all fields that are unchecked, you must use the IS NULL on the array as follows (without a value):

```
filter[my-filter][condition][path]=my_field
filter[my-filter][condition][operator]=IS NULL
```

Follow

Help improve this page

Page status: No known problems

You can:
Log in, click Edit, and edit this page
Log in, click Discuss, update the Page status value, and suggest an improvement
Log in and create a Documentation issue with your suggestion