Learning Objectives

- **Explain the differences between the `PageNumberPagination`, `LimitOffsetPagination`, and `CursorPagination` classes**
- **Identify differences in the pagination response from the different pagination classes**
- **Implement pagination on custom action methods**
- **Install and set up the third-party library `django-filter`**
- **Identify the advantage of using the `FilterSet` class**
- **Customize the order of results from filtering**

Clone Blango Repo

Clone Blango Repo

Before we continue, you need to clone the `blango` repo so you have all of your code. You will need the SSH information for your repo.

In the Terminal

- Clone the repo. Your command should look something like this:

```
git clone git@github.com:<your_github_username>/blango.git
```

- You should see a `blango` directory appear in the file tree.

You are now ready for the next assignment.

Pagination Part 1

Django Rest Framework Third-Party Libraries

Just as Django has a rich collection of libraries and plugins to add enhancements, so does Django Rest Framework. In this module, we're going to look at three of them: django-filter, which adds extensible filtering support; SimpleJWT, for authentication with JSON web tokens; and django-versatileimagefield, which adds some enhancements to image management.

Since we finished the last module with filtering, the first library we'll look at is *django-filter*. However, before we do, we're going to talk about paginating of list responses.

DRF Pagination

We're about to make some changes to allow clients to filter the response using query parameters, for example filtering posts to a particular author by appending `?author=<author_id>`. Since we're in this area, we'll look at paginating the response, which also means changing the returned data using query parameters.

DRF provides three built-in pagination classes:

`PageNumberPagination`

Using this class treats the list results as a page, the client can move through the results by specifying a `page`. For example: `/api/v1/posts/?page=2`. By default the size of the page is fixed, and defined in the DRF settings dictionary. However, you can subclass `PageNumberPagination` and set attributes to customize the behavior. The attributes you're most likely to set are:

- `page_size`: An integer indicating the page size, which will override the `PAGE_SIZE` setting.
- `page_query_param`: A string indicating the page query parameter. Defaults to `page`.
- `page_size_query_param`: If set, will allow the client to set the size of the page through a query parameter. An example would be something like `page_size`. If unset, then the page size can't be changed by the client and is fixed at `page_size`.
- `max_page_size`: The maximum page size that the client can request, only valid if `page_size_query_param` is also set.

- `last_page_strings`: A list/tuple of strings that the client can use as an alias to fetch the last page. Defaults to the tuple `("last",)`. For example, the client could do `?page=last`.

There are a couple of other options, you can see the full list at the `PageNumberPagination` documentation.

The `PageNumberPagination` class (or other pagination classes) can be applied globally with the `DEFAULT_PAGINATION_CLASS` item in the `REST_FRAMEWORK` setting:

```
REST_FRAMEWORK = {
    "DEFAULT_PAGINATION_CLASS":
        "rest_framework.pagination.PageNumberPagination",
    "PAGE_SIZE": 100
}
```

Or, a pagination class can be applied to a specific view or viewset:

```
from rest_framework.pagination import PageNumberPagination

class PostViewSet(viewsets.ModelViewSet):
    pagination_class = PageNumberPagination
    # existing attributes/methods omitted
```

`LimitOffsetPagination`

This pagination class works like paginating a SQL query. You would get the first 100 results like this: `/api/v1/posts/?offset=0&limit=100`. Then, fetch the next 100 like this: `/api/v1/posts/?offset=100&limit=100`, and so on.

There are options for customizing behavior too, if you subclass `LimitOffsetPagination` these are some of the common attributes you might want to set.

- `default_limit`: This defaults to the `PAGE_SIZE` setting from the `REST_FRAMEWORK` settings.
- `limit_query_param`: A string indicating the query parameter to use for the limit, defaults to `limit`.
- `offset_query_param`: A string indicating the query parameter to use for the offset, defaults to `offset`.
- `max_limit`: If set (to an integer) determines the maximum limit the client can set.

The full list of parameters that can be set can be found at the `LimitOffsetPagination` documentation.

Like `PageNumberPagination`, `LimitOffsetPagination` can be set globally with the `DEFAULT_PAGINATION_CLASS` setting of `rest_framework.pagination.LimitOffsetPagination`. Or, it can be manually set on a view or viewset class with the `pagination_class` attribute.

`CursorPagination`

This class uses a special cursor query parameter to page through the results. The parameter is opaque, in that the client doesn't control it. Instead, on each request, DRF generates URLs containing the cursor variable. The client can then use these URLs to fetch the next set of results.

Here's an example of a response body that's paginated by `CursorPagination`:

```
{
    "next": "http://127.0.0.1:8000/api/v1/posts/?
        cursor=cD1hZHZhbmNlZC1kamFuZ28tYS1yZXZpZXc%3D",
    "previous": "http://127.0.0.1:8000/api/v1/posts/?
        cursor=cj0xJnA9YWR2YW5jZWQtZGphbmdvLWEtcmV2aWV3",
    "results": [
        {
           ... list of results
        }
    ]
}
```

To use `CursorPagination`, your model must have a unique field that can be ordered on. By default, DRF looks for a date and time field called `created`. Your results start at the most recently created object, and go through each field in the created order. In theory, this means you can pick up where you left off, even much later in the future. You can order on other unique fields too though, like `slug`, however you might miss out on results if an object is inserted with a slug that's ordered before your current cursor point. This could be seen as an advantage though, as you'll never see the same object twice.

`CursorPagination` should be considered when working with very large datasets. Paging through data can become slow with using offsets, whereas cursor based paging does not. The disadvantage though, is that a client can't jump to an arbitrary page or offset.

The most common attributes for customization (once you've created a subclass) are:

- `page_size`: An integer indicating the page size, which will override the `PAGE_SIZE` setting.
- `cursor_query_param`: The query parameter string to use in the URL for the cursor, defaults to `cursor`.

- `ordering`: A string, or list of strings, to order the results by. This defaults to `-ordering`.

Since `CursorPagination` is more complex than the other classes, it's recommended to read the official documentation to see all the requirements of its use.

Like other pagination classes, `CursorPagination` can be set globally with the `DEFAULT_PAGINATION_CLASS` setting of `rest_framework.pagination.CursorPagination`. Or, it can be manually set on a view or viewset class with the `pagination_class` attribute.

Pagination Part 2

Custom pagination

If none of the built-in pagination classes meet your needs, you can write your own. Check the custom pagination styles documentation for more information.

The pagination response

As you may have noticed from our `CursorPagination` response snippet, once pagination is enabled, any list results are nested inside a dictionary.

The items in the response dictionary will depend on the pagination class. For `PageNumberPagination` and `LimitOffsetPagination` you'll have the items:

- `count`: The total number of records available.
- `next`: A URL to the next set of results. For example, if you're on the first page, this will be a URL with a query parameter like `?page=2`. If there are no more pages, this will be `null`.
- `previous`: A URL to the previous set of results. If you're on the second page, it will have the parameter `?page=1`. If you're on the first page, it will be `null`.
- `results`: An array of objects on this page. This is the equivalent of the body of the response that was sent when pagination was not enabled.

`CursorPagination` responses don't contain a `count` item. Custom pagination classes could have completely different response bodies too.

Here's an example of a `Post` list response using `PageNumberPagination`, with a page size of 2:

```
{
    "count": 11,
    "next": "http://127.0.0.1:8000/api/v1/posts/?page=2",
    "previous": null,
    "results": [
        {
            "id": 15,
            "tags": [
                "django"
            ],
            "author":
        "http://127.0.0.1:8000/api/v1/users/ben@example.com",
            "created_at": "2021-08-22T20:45:43.732629Z",
            "modified_at": "2021-08-30T08:34:23.183019Z",
            "published_at": "2021-08-30T08:34:21Z",
            "title": "Hello from Postman, with new API4",
            "slug": "hello-from-postman-with-new-api4",
            "summary": "This was created from Postman with new
        API.",
            "content": "<p>This is a great way to send data to
        an API (AGAIN), by using Postman.</p>"
        },
        {
            "id": 6,
            "tags": [
                "django",
                "test"
            ],
            "author":
        "http://127.0.0.1:8000/api/v1/users/ben@example.com",
            "created_at": "2021-07-20T04:21:42.688000Z",
            "modified_at": "2021-08-21T07:37:22.891390Z",
            "published_at": "2021-07-21T04:08:45Z",
            "title": "Django vs Python",
            "slug": "django-vs-python",
            "summary": "Which is better, Django or Python? We
        investigate.",
            "content": "<p>Many people don't know, but Django is
        created using Python. So it's not a matter of picking
        one or the other, Python is always there when Django is
        involved!</p>\r\n<p>It might be like saying, cars vs
        roads, which is better?</p>"
        }
    ]
}
```


It's best to enable pagination early on in a project, as it can break clients that don't expect a paginated response.

Pagination of custom methods

Once we've turned on pagination, either globally or for a class, it's automatically applied to generic APIView endpoints and viewsets. To make pagination work with custom `action` methods or on normal `APIViews`, they must be manually applied.

For non-generic `APIViews`, the process is:

- Instantiate a paginator class.
- Paginate the queryset by calling the `paginate_queryset()` method with the queryset and request. This will evaluate the queryset and return a list with a subset of objects.
- Pass the page of objects as the data to the serializer you're using.
- Call the `get_paginated_response()` method on the paginator, and pass in the serializer's data. This returns a `Response`.
- Return the response from `get_paginated_response()`.

If using a viewset or generic view, it's a bit easier, as helper methods are provided.

- Call `self.paginate_queryset()`, passing in the queryset (e.g. from `self.get_queryset()`). If no paginator is set up (on the class or in the settings), then this method will return `None`.
- If `paginate_queryset()` returns a list, then pass the page into a serializer.
- Pass the serializer's `data` to `self.get_paginated_response()`, which returns a `Response` object.
- Return the `Response` from your action method.
- If `paginate_queryset()` returns `None`, then fall back to your original method of generating a response, by passing the entire queryset to the serializer.

Since we're not using any base `APIViews` we can't demonstrate the process, but we'll show how to update an action method. Here is the `mine` method on `PostViewSet`:

```
    def mine(self, request):
        if request.user.is_anonymous:
            raise PermissionDenied("You must be logged in to see
        which Posts are yours")
        posts = self.get_queryset().filter(author=request.user)


        page = self.paginate_queryset(posts)


        if page is not None:
            serializer = PostSerializer(page, many=True,
        context={"request": request})
            return self.get_paginated_response(serializer.data)


        serializer = PostSerializer(posts, many=True, context=
        {"request": request})
        return Response(serializer.data)
```

Note that we only make the paginated response if `self.paginate_queryset(posts)` returns non-`None` (it might sometimes return an empty list). Otherwise, we fall back to the normal response. In this way, we can toggle pagination on or off through settings or the `pagination_class` attribute and have all our list response be in the same format.

Try It Out - Pagination

Try It Out:

Let's set up pagination for your Blango project. We'll use the `PageNumberPagination` class. There are a few changes to make, we'll start simple: updating the settings. Open `settings.py` and add the settings for pagination to the `REST_FRAMEWORK` dictionary:

```
REST_FRAMEWORK = {
    # existing settings omitted
    "DEFAULT_PAGINATION_CLASS":
    "rest_framework.pagination.PageNumberPagination",
    "PAGE_SIZE": 100,
}
```

This is technically all you need to do to get pagination working, however if you try out the `Tag` list API now, you'll get a warning in your console:

```
Pagination may yield inconsistent results with an unordered
object_list: <class 'blog.models.Tag'> QuerySet.
```

This is because we don't have any ordering on our `Tag` model, which means that the ordering could change between pages. Let's add one now: we'll just set the default ordering to `value` so the tags are retrieved in alphabetical order.

Add this `class Meta` attribute to your `Tag` model:

Open models.py

```
class Meta:
    ordering = ["value"]
```

Now you can load up the `Post` or `Tag` list views in the browser, and you should see them paginated. If you really want to test it properly, you can temporarily adjust the `PAGE_SIZE` down to `1` to see how it works with multiple pages. Remember to change it back to a reasonable number like `100` when you're finished testing.

View Blog

Next we need to fix our action methods on `PostViewSet` (`mine()`) and `TagViewSet` (`posts()`). Unless we manually paginate the querysets, they'll be returned in the prior, non-paginated format.

Open `blog/api/views.py` and update the `mine()` method on `PostViewSet` to this:

Open api/views.py

```
    def mine(self, request):
        if request.user.is_anonymous:
            raise PermissionDenied("You must be logged in to see
        which Posts are yours")
        posts = self.get_queryset().filter(author=request.user)

        page = self.paginate_queryset(posts)

        if page is not None:
            serializer = PostSerializer(page, many=True,
        context={"request": request})
            return self.get_paginated_response(serializer.data)

        serializer = PostSerializer(posts, many=True, context=
        {"request": request})
        return Response(serializer.data)
```

Then we'll perform a similar fix to the `posts()` method on `TagViewSet`. Update the method to this:

```
    def posts(self, request, pk=None):
        tag = self.get_object()
        page = self.paginate_queryset(tag.posts)
        if page is not None:
            post_serializer = PostSerializer(
                page, many=True, context={"request": request}
            )
            return
        self.get_paginated_response(post_serializer.data)
        post_serializer = PostSerializer(
            tag.posts, many=True, context={"request": request}
        )
        return Response(post_serializer.data)
```

Now you can try out these methods through the DRF GUI or Postman and verify that they are returned as paginated responses too. If you are testing with DRF GUI, you need to look at the blog in a separate tab in your browser. Click the blue arrow icon to open blango in its own tab.

View Blog

The final thing we should do is fix up our tests. They've been written to assume the results are in the root of the response, but now they're under the `results` key of the returned dictionary. We need to update the tests to read the results from here instead. Luckily, it's a simple fix.

Open test_post_api.py

Open the `test_post_api.py` file and find the line in the `test_post_list()` method that reads:

```
data = resp.json()
```

It should be about the second line of the test. Update it to extract the `results` from the JSON:

```
data = resp.json()["results"]
```

You need to do a similar fix to `test_tag_api.py`. In the `test_tag_list()` method find the line:

Open test_tag_api.py

```
data = resp.json()
```

It should be the third line. Update it to:

```
data = resp.json()["results"]
```

Run your tests with `python manage.py test` and you should see that all six of them still pass.

Now that we've covered pagination, let's look at the *django-filter* module.

Django-Filter Introduction

Django-Filter Introduction

django-filter provides methods of filtering querysets that can be used in either standard Django views or with DRF. Let's look at a really simple way of getting some filtering in our API.

First, `django-filter` needs to be installed with `pip`:

```
pip3 install django-filter
```

Then the filter backends need to be set. As with other DRF options, the filter backends can be set globally in the `REST_FRAMEWORK` settings:

```
    REST_FRAMEWORK = {
        # other settings omitted
        "DEFAULT_FILTER_BACKENDS": [
            "django_filters.rest_framework.DjangoFilterBackend"
        ],
    }
```

Or, they can be applied to individual views and viewsets with the `filter_backends` attribute:

```
import django_filters.rest_framework

class PostViewSet(viewsets.ModelViewSet):
    filter_backends =
        [django_filters.rest_framework.DjangoFilterBackend]

    # other attributes and methods omitted
```

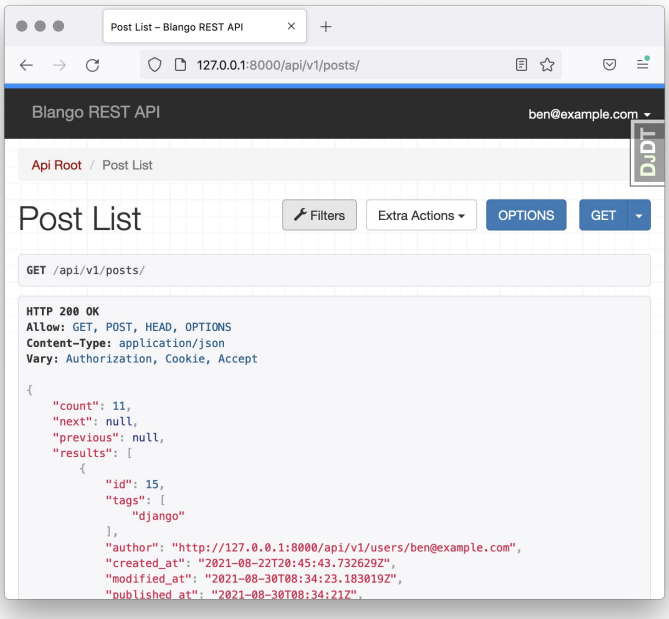
The `django_filters.rest_framework.DjangoFilterBackend` backend is the backend designed to work with `django-filter`.

Whichever method you choose, `django_filters` needs to also be added to `INSTALLED_APPS` in the settings file.

Then to get simple equality-based filtering, we just need to define a list of fields that we want to be able to filter on. These are set as the `filterset_fields` attribute. We'll look at how to do more complex searches soon, but for now, something like this will provide basic filtering of `Posts`:

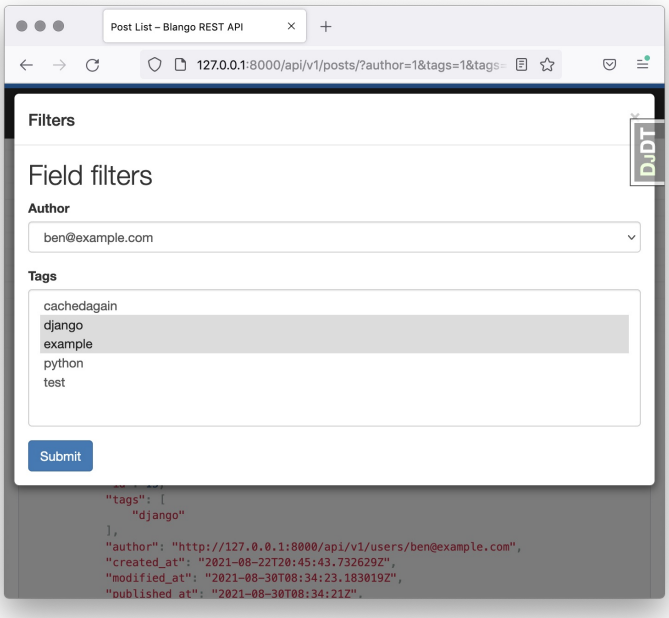
```
class PostViewSet(viewsets.ModelViewSet):
    filterset_fields = ["author", "tags"]
    # existing attributes and methods omitted
```

Filter backends work with the DRF GUI to add a filtering form. The posts list page now has a **Filters** button at the top.



filters button

Clicking on it brings up a modal and form where the filtering options can be selected.



filters modal

After clicking **Submit** the form is submitted which takes the browser to a new URL. The URL of the page can be examined to see how the filters are applied:

`http://127.0.0.1:8000/api/v1/posts/?author=1&tags=1&tags=2`

The `author` and `tags` are both set by ID. Having the same parameter multiple times act as an *or* operator, so posts containing either tag `1` (*django*) or tag `2` (*example*) are found.

Try It Out

You can now set up this same filtering on your Blango project. Start by installing `django-filter` with `pip`:

```
pip3 install django-filter
```

Next, add `DEFAULT_FILTER_BACKENDS` to the `REST_FRAMEWORK` setting in `settings.py`:

Open settings.py

```
    REST_FRAMEWORK = {
        # existing settings omitted
        "DEFAULT_FILTER_BACKENDS": [
            "django_filters.rest_framework.DjangoFilterBackend"
        ],
    }
```

And, add `django_filters` to your `INSTALLED_APPS` setting.

Next, head over to `blog/api/views.py`. Add the `filterset_fields` attribute to `PostViewSet`:

Open api/views.py

```
class PostViewSet(viewsets.ModelViewSet):
    filterset_fields = ["author", "tags"]
    # existing attributes and methods omitted
```

Now visit the post list page in a browser and try out filtering. You should also notice that the existing queryset rules are also applied, for example, if you're logged out, you should see only published posts that match your filter criteria.

View Blog

Now let's take a look in more detail at what's happening under the hood when we add filterset fields.

The FilterSet Class

The `FilterSet` Class

Behind the scenes, `DjangoFilterBackend` is actually using the `filterset_fields` attribute to build a `FilterSet` (`django_filters.rest_framework.FilterSet`) class. This is a class that defines the filters that can be used on a view or viewset. It's kind of like a `ModelSerializer`: it can automatically pick up fields from a model and allow simple equality-based filtering (like we've already seen), or we can add custom fields with more complex rules.

A manually defined `FilterSet` that has the same functionality as what was built using just the `filterset_fields` attribute would look like this:

```
from django_filters import rest_framework as filters
from blog.models import Post


class PostFilterSet(filters.FilterSet):
    class Meta:
        model = Post
        fields = ["author", "tags"]
```

It's set on the view/viewset using the `filterset_class` attribute:

```
class PostViewSet(viewsets.ModelViewSet):
    filterset_class = PostFilterSet
    # filterset_fields can be removed
    # other existing attributes and methods omitted
```

The real advantage of using `FilterSet` class is by defining extra fields which can have validation and use custom filter arguments. In this way we can come up with "meta" fields that will be applied to the queryset filters in custom ways. This is best introduced with an example.

Let's say we want to allow clients to look for `Posts` between certain dates. This can't be done just by adding `published_at` to the fields list, because that would only allow searching by the exact date and time, down to the millisecond.

Instead two fields should be created, one that accepts the date being searched from, and another that accepts the date being searched to. The client can set one or both to create a date range. This is done with the `django_filters.rest_framework.DateFilter` class as a field. They're instantiated with `field_name` and (usually) the `lookup_expr` keyword arguments:

```
class PostFilterSet(filters.FilterSet):
    published_from = filters.DateFilter(
        field_name="published_at", lookup_expr="gte",
        label="Published Date From"
    )
    published_to = filters.DateFilter(
        field_name="published_at", lookup_expr="lte",
        label="Published Date To"
    )

    class Meta:
        model = Post
        fields = ["author", "tags"]
```

The `label` argument is optional and we'll see its purpose soon.

This will provide two new arguments that can be used in the URL query parameters, `published_from` and `published_to`. To look at just `published_from`, it would be applied to the queryset by joining the `field_name` to the `lookup_expr`.

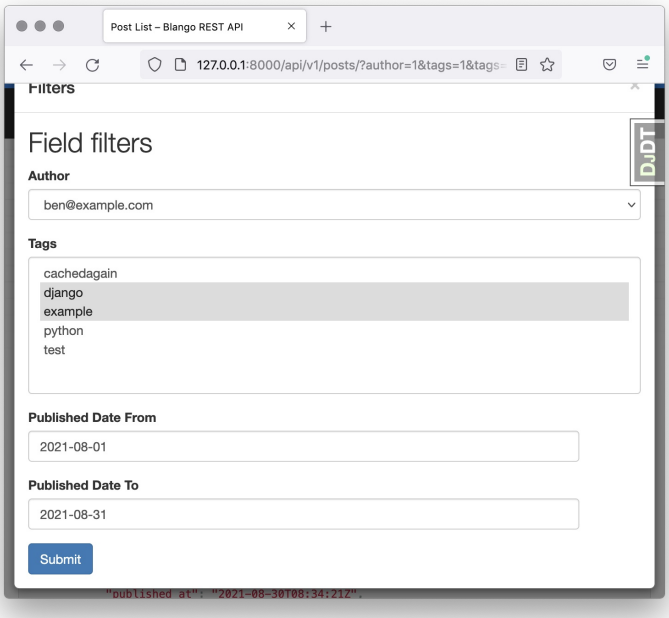
For example, to get all `Posts` published in August, the query parameters would be `?published_from=2021-08-01&published_to=2021-08-31`.

Which would be applied to the queryset like so:

```
Post.objects.filter(
    published_at__gte="2021-08-01"
).filter(
    published_at__lte="2021-08-31"
)
```

Although note this is for illustration purposes, the filter will actually be applied to the result of the view/viewset's `get_queryset()` method.

In the DRF GUI, these fields will now be shown in the filters form, and will be labeled with the optional `label` argument that was used.



published date filters

The `Filter` also checks that the value being entered is valid. For example, trying to filter by an invalid date (with a query parameter of `?published_from=not+valid`) will generate a response with a `400` status code and this body:

```
{
    "published_from": [
        "Enter a valid date."
    ]
}
```

Now that we've seen an example, let's look at some of the other fields that are available. These classes are all defined in the `django_filters.rest_framework` module, which we've imported as the alias `filters`.

- `CharFilter`: for strings.
- `BooleanFilter`: for filtering by boolean, valid values are `True`, `true` and `1` for true, or `False`, `false` and `0` for false.
- `ChoiceFilter`: filter based on a list of available choices. Choices are set in the same manner as the `ChoiceField` for a model: with a list of tuples containing the value and a label, passed in as the `choices` keyword argument.
- `UUIDFilter`: accepts a UUID as a string.
- `DateFilter`: accepts a date.
- `DateTimeFilter`: accepts a date and time.
- `TimeFilter`: accepts a time.

Let's look at a couple more ways of using filters. Related fields can be searched on too, by specifying the `field_name` in the same way it would be passed to the `filter()` method: with a double underscore before the related field. For example, to allow searching by the author's exact email, specify a `CharField` like this:

```
author_email = filters.CharFilter(field_name="author__email",
        label="Author Email")
```

Or we could take this further and allow searching inside a user email rather than exact match, using the `icontains` (case insensitive contains) lookup expression:

```
author_email = filters.CharFilter(
    field_name="author__email",
    lookup_expr="icontains",
    label="Author Email Contains",
)
```

Similarly, searching inside of `content` and `summary` can be added with a couple of extra fields:

```
summary = filters.CharFilter(
    field_name="summary",
    lookup_expr="icontains",
    label="Summary Contains",
)
content = filters.CharFilter(
    field_name="content",
    lookup_expr="icontains",
    label="Content Contains",
)
```

Using *django-filter* can get much more complex, if your application requires it. The official DRF integration documentation is a good starting point if you need to make a deeper dive.

Try It Out - FilterSet

Try It Out

You'll now add these advanced filters to your Blango app. We'll keep the filtersets in their own file. Create the file `blog/api/filters.py`.

Start by importing the filters module and our `Post` model:

```
from django_filters import rest_framework as filters

from blog.models import Post
```

Then define the `PostFilterSet` class like so:

```
class PostFilterSet(filters.FilterSet):
    published_from = filters.DateFilter(
        field_name="published_at", lookup_expr="gte",
        label="Published Date From"
    )
    published_to = filters.DateFilter(
        field_name="published_at", lookup_expr="lte",
        label="Published Date To"
    )
    author_email = filters.CharFilter(
        field_name="author__email",
        lookup_expr="icontains",
        label="Author Email Contains",
    )
    summary = filters.CharFilter(
        field_name="summary",
        lookup_expr="icontains",
        label="Summary Contains",
    )
    content = filters.CharFilter(
        field_name="content",
        lookup_expr="icontains",
        label="Content Contains",
    )

    class Meta:
        model = Post
        fields = ["author", "tags"]
```

Save this file and switch over to `blog/api/views.py`.

Open api/views.py

Start by importing `PostFilterSet` at the start of the file:

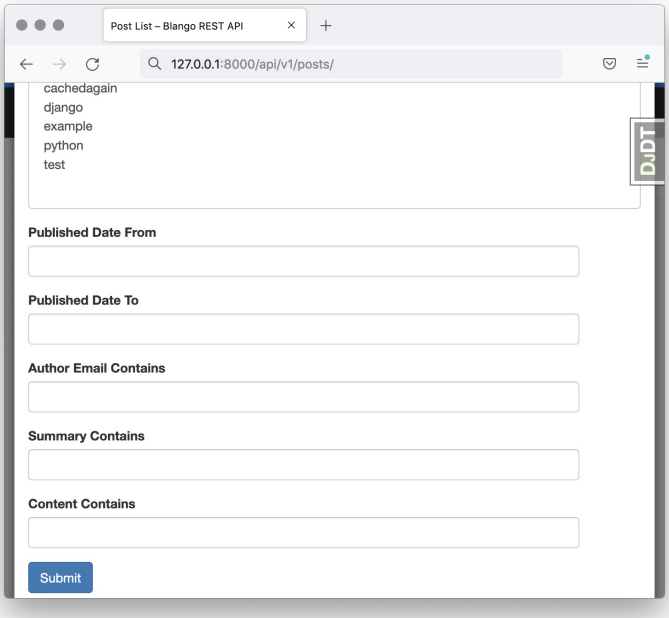
```
from blog.api.filters import PostFilterSet
```

Then update `PostViewSet`. Remove the `filterset_fields` attribute, and replace it with a `filterset_class` attribute:

```
class PostViewSet(viewsets.ModelViewSet):
    filterset_class = PostFilterSet
    # filterset_fields can be removed
    # other existing attributes and methods omitted
```

Now load up the `Posts` list view in the DRF GUI and open the filters modal. You should see all the extra filters.

View Blog



all extra filters

Try out some searches, and confirm you're getting back the results you expect.

The last thing we're going to look at which is kind of related to filtering, is ordering of results.

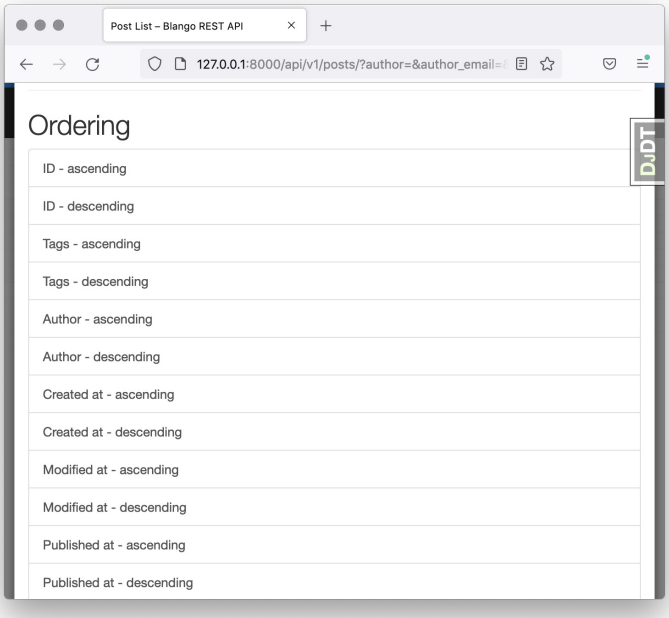
Ordering

Ordering

Custom ordering of results returned in DRF can be easily set up with the use of the `rest_framework.filters.OrderingFilter` class. It can just be added to the list of `DEFAULT_FILTER_BACKENDS` and then it's available to all views and viewsets automatically:

```
REST_FRAMEWORK = {
    # existing settings omitted
    "DEFAULT_FILTER_BACKENDS": [
        "django_filters.rest_framework.DjangoFilterBackend",
        "rest_framework.filters.OrderingFilter"
    ],
}
```

By default, all readable serialized fields are available for ordering:

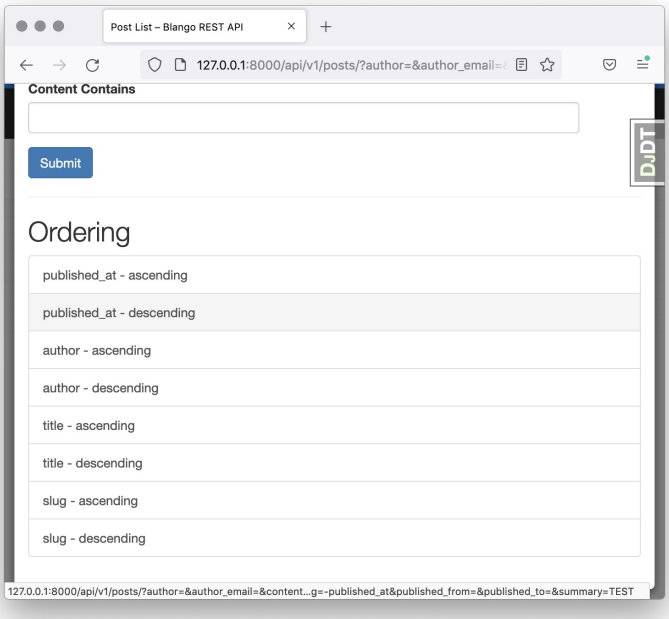


all ordering options

The fields that can be ordered on can be customized by using the `ordering_fields` attribute on a view/viewset.

For example, `Post` list can be made orderable just by published date, author, title and slug with this setting:

```
class PostViewSet(viewsets.ModelViewSet):
    ordering_fields = ["published_at", "author", "title",
        "slug"]
    # existing attributes and methods omitted
```



specified ordering options

In terms of how the fields are applied in the query parameters, to sort descending the field name is prepended by a `-`, just like when ordering in a Django queryset. Multiple fields can be applied by joining with a comma. To sort by author ascending, and then published date descending, the query parameter would be like this: `?ordering=author,-published_at`.

Try It Out

Let's add some ordering to DRF in your Blango project. Start in `settings.py`, add `rest_framework.filters.OrderingFilter` to the list of `DEFAULT_FILTER_BACKENDS`:

```
REST_FRAMEWORK = {
    # existing settings omitted
    "DEFAULT_FILTER_BACKENDS": [
        "django_filters.rest_framework.DjangoFilterBackend",
        "rest_framework.filters.OrderingFilter"
    ],
}
```

Ordering will work just fine with just this setting change, but let's also specify the orderable fields on `PostViewSet`. Open `blog/api/views.py` and add the list of fields to the `ordering_fields` attribute:

Open api/views.py

```
class PostViewSet(viewsets.ModelViewSet):
    ordering_fields = ["published_at", "author", "title",
        "slug"]
    # other existing attributes and methods omitted
```

Load up a DRF GUI for the `Posts` list and you'll see the ordering fields are now available in the filters modal.

View Blog

That's all we're going to cover for our first DRF third-party library. Next up we're going to look at authenticating with *SimpleJWT*.

Pushing to GitHub

Pushing to GitHub

Before continuing, you must push your work to GitHub. In the terminal:

- Commit your changes:

```
git add .
git commit -m "Finish pagination and django filter"
```

- Push to GitHub:

```
git push
```