frf Documentation

Release 0.1

Adam Olsen

Aug 29, 2017

Contents

Falcon Rest Framework is a REST framework inspired by Django Rest Framework, and written using the Falcon web framework. Falcon's claim to fame is speed, and it achieves this in part by avoiding object instantiation of objects during the request as much as possible. If you think about it, a Django Rest Framwork viewset that looks like this:

```
from rest_framework.viewsets import ViewSet
from package import filters, serializers, permissions, models, authentication

class SomeViewSet(ViewSet):
    queryset = models.Foo.objects.all()
    serializer_class = serializers.FooSerializer
    permission_classes = [permissions.IsStaffPermission]
    authentication_classes = [authentication.OAuth2Authentication]
    filter_backends = [filters.SearchFilter, filters.OrderingFilter]

    def update(self, request, *args, **kwargs):
        # update...
```

Every one of **serializers.FooSerializer**, **permissions.IsStaffPermission**, **authentication.OAuth2Authentication**, **filters.SearchFilter**, and **filters.OrderingFilter** are instantiated and destroyed as a part of the request process. This is in addition to any objects that Django itself creates. Falcon tries to defer this to the boot process, and only instantiates a very minimal set of classes during the request. Falcon Rest Framework follows this philosophy, deferring everything from filters, middleware, authentication, parsers, renders, serializers are all created during boot, not during the request.

A similar viewset in frf would look like this:

```
from frf.viewsets import ViewSet
from package import filters, serializers, permissions, models, authentication

class SomeViewSet(viewsets.ViewSet):
    model = models.Foo
    serializer = serializers.FooSerializer()
    permissions = [permissions.IsStaffPermission()]
    authentication = [authentication.OAuth2Authentication()]
    filters = [
        filters.SearchFilter(models.Foo.name),
        filters.OrderingFilter(models.Foo.name),
    ]

    def update(self, req, resp, **kwargs):
        # ... update
```

Read the documentation to find out more!

Contents:

CHAPTER 1

Getting Started

Installation

Requirements

Main requirements:

- pytz
- SQLAlchemy
- falcon
- jinja2
- pycrypto
- tabulate
- pytest
- python-datutil
- gunicorn

Optional requirements:

- redis (for redis cache support)
- Sphinx (for document generation)
- colorama (for pretty output)
- pyfiglet (for pretty output)

PyPy

PyPy is the fastest way to run your FRF app.

```
$ pip install frf
```

From Source

```
$ git clone https://github.com/enderlabs/frf
$ cd frf
$ python setup.py install
```

Tutorial

In this tutorial, we will be making a simple API to retrieve and store blog articles. We will call the project `blogapi`.

We assume that you already have Python 3 installed, which is required for FRF. One you have followed the following steps, you should have a working blog api!

Create a Virtualenv

To keep the packages that you install that are required for FRF separate from system packages, I recommend using virtualenv. You can install virtualenv like so:

```
$ pip install virtualenv virtualenvrapper
```

Now, create your virtualenv for our `blogapi` application:

```
$ mkvirtualenv --python=/usr/bin/python3 blogapi
```

The location of your python3 executable may be in a different location, and you may need to change the `--python` flag accordingly.

Now, create your project directory, something like the following:

```
$ mkdir ~/blogapi
$ cd ~/blogapi
$ add2virtualenv .
```

Install Falcon Rest Framework

Make sure you are in the project directory, and that the virtualenv is active:

```
$ cd ~/blogapi
$ workon blogapi
```

Now, install FRF:

```
$ pip install frf
```

If you want stdout output to be a bit prettier, install `colorama` and `pyfiglet`:

```
$ pip install colorama pyfiglet
```

And, now, you should be able to initialize your project directory:

```
$ frf-setupproject -o .
```

Create the Blog Application

FRF applications are arranged by apps. For instance, our `blogapi` repository might have a `blog` app for serving blog entries, but we might also have a `users` app for user authentication or a `gallery` app for serving up pictures.

Let's create our `blog` app:

```
$ ./manage.py startapp blog
```

This will create a directory structure like the following:

```
blogapi
|-- blog
|   |-- __init__.py
|   |-- models.py
|   |-- tests.py
|   |-- urls.py
|   |-- serializers.py
|   +-- viewsets.py
+-- blogapi
    |-- __init__.py
    +-- settings.py
```

Create Models

Lets create a model for our blog articles. Open up `blog/models.py`, and edit it to look like this:

Listing 1.1: models.py

```
import uuid

from frf import models


class Article(models.Model):
    __tablename__ = 'blog_article'
    uuid = models.Column(models.GUID, primary_key=True, default=uuid.uuid4)
    author = models.Column(models.String(40), index=True)
    post_date = models.Column(
        models.DateTime(timezone=True),
        default=models.func.current_timestamp())
    title = models.Column(models.String(255))
    text = models.Column(models.Text)
```

Let's tell FRF that we are ready to use our new model. Open up `blogapi/settings.py` and find the line that says `INSTALLED_APPS` and change it to look like this:

Listing 1.2: settings.py

```
#: Database configuration
INSTALLED_APPS = ['blog']
```

Now we can tell FRF to actually create the table in the database for us:

```
$ ./manage.py syncdb
Creating table blog_article... Done.
```

Create a Serializer

We need a serializer to convert our blog posts to and from json. Open up `blog/serializers.py` and edit it to look like this:

Listing 1.3: serializers.py

```
from frf import serializers

from blog import models


class ArticleSerializer(serializers.ModelSerializer):
    uuid = serializers.UUIDField(read_only=True)

    class Meta:
        fields = ('uuid', 'author', 'post_date', 'title', 'text')
        required = ('author', 'title', 'text')
        model = models.Article
```

Most primitive type fields are detected from the model automatically. Here we are overriding the `uuid` field so we can make it read-only. The other fields are detected and added automatically.

Create a ViewSet

Let's create a view to serve up our blog entries. Open up `blog/viewsets.py` and edit it to look like this:

Listing 1.4: viewsets.py

```
from frf import viewsets

from blog import models, serializers


class ArticleViewSet(viewsets.ModelViewSet):
    serializer = serializers.ArticleSerializer()

    def get_qs(self, req, **kwargs):
        return models.Article.query.order_by(
            models.Article.post_date.desc())
```

Add a URL Route

We need to tell FRF how to map what url to this new ViewSet. Open `blog/urls.py` and edit it to look like this:

Listing 1.5: urls.py

```
from blog import viewsets

article_viewset = viewsets.ArticleViewSet()

urlpatterns = [
    ('/blog/articles/', article_viewset),
    ('/blog/articles/{uuid}/', article_viewset),
    ]
```

Now we need to tell our app to use the blog url routes. Open `blogapi/urls.py` and edit it to look like this:

Listing 1.6: urls.py

```
from frf.urls import include, route  # noqa

urlpatterns = [
    ('/api/', include('blog.urls')),
]
```

Start the Server

Start up the webserver:

```
$ ./managed.py runserver
Oh hai, starting gunicorn...
[2016-09-22 17:11:41 -0600] [11875] [INFO] Starting gunicorn 19.6.0
[2016-09-22 17:11:41 -0600] [11875] [INFO] Listening at: http://0.0.0.0:8080 (11875)
[2016-09-22 17:11:41 -0600] [11875] [INFO] Using worker: sync
[2016-09-22 17:11:41 -0600] [11878] [INFO] Booting worker with pid: 11878
[2016-09-22 17:11:41 -0600] [11879] [INFO] Booting worker with pid: 11879
```

Congratulations!!! You now have a blog api ready for requests. Let's give it a try...

Try it Out

The following examples use curl, please make sure it's installed before beginning!

Create a Post

```
$ curl -H 'Content-Type: application/json' -X POST -d \
    '{"author": "adam", "title": "This is an article", "text":
    "How do you like this most amazing article?"}' \
    http://0.0.0.0:8080/api/blog/articles/
```

Now you can use the shell to see if it worked:

```
$ ./manage.py shell
Python 3.5.1 (v3.5.1:37a07cee5969, Dec  5 2015, 21:12:44)
[GCC 4.2.1 (Apple Inc. build 5666) (dot 3)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
(InteractiveConsole)
>>> from blog import models
```

```
>>> a = models.Article.query.first()
>>> a.author
'adam'
>>> a.title
'This is an article'
>>> a.text
'How do you like this most amazing article?'
>>> a.post_date
datetime.datetime(2016, 9, 22, 23, 20, 3, tzinfo=<UTC>)
>>>
>>> a.uuid
UUID('1e1cbab1-c8b0-4cd7-a8e1-a471aead09e2')
>>>
```

Do this a few more times, so that we have a few articles in the system.

List Posts

```
$ curl -H 'Content-Type: application/json' -X GET \
    http://0.0.0.0:8080/api/blog/articles/ | python -m json.tool
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
100   715  100   715    0     0   224k      0 --:--:-- --:--:-- --:--:--  349k
[
    {
        "text": "...",
        "author": "adam",
        "post_date": "2016-09-22T23:40:27+00:00",
        "title": "Fantastic article",
        "uuid": "14cb654e-e85d-4016-9196-28551eb52d6e"
    },
    {
        "text": "Test Article",
        "author": "adam",
        "post_date": "2016-09-22T23:39:03+00:00",
        "title": "This is an another article",
        "uuid": "8f44fafd-9190-425e-ab5a-08392f661912"
    },
    {
        "text": "How do you like this most amazing article?",
        "author": "adam",
        "post_date": "2016-09-22T23:38:28+00:00",
        "title": "This is an article",
        "uuid": "40dacbad-2374-4042-91b1-5d5d80b1701f"
    },
    {
        "text": "How do you like this most amazing article?",
        "author": "adam",
        "post_date": "2016-09-22T23:22:17+00:00",
        "title": "This is an article",
        "uuid": "1e1cbab1-c8b0-4cd7-a8e1-a471aead09e2"
    }
]
```

Retrieve a Post

Note a UUID from the last step, in my case, one of them was `1e1cbab1-c8b0-4cd7-a8e1-a471aead09e2`. Let's retrieve our article using the api:

```
$ curl -H 'Content-Type: application/json' -X GET \
    http://0.0.0.0:8080/api/blog/articles/1e1cbab1-c8b0-4cd7-a8e1-a471aead09e2 |␣
↪python -m json.tool
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
100   195  100   195    0     0  14773      0 --:--:-- --:--:-- --:--:-- 16250
[
    {
        "text": "How do you like this most amazing article?",
        "author": "adam",
        "uuid": "1e1cbab1-c8b0-4cd7-a8e1-a471aead09e2",
        "post_date": "2016-09-22T23:20:03+00:00",
        "title": "This is an article"
    }
]
```

Update a Post

Let's update the author of the post to `unknown`:

```
$ curl -H 'Content-Type: application/json' -X PUT -d \
    '{"author": "unknown"}' \
    http://0.0.0.0:8080/api/blog/articles/1e1cbab1-c8b0-4cd7-a8e1-a471aead09e2
```

Use the API to verify:

```
$ curl -H 'Content-Type: application/json' -X GET \
    http://0.0.0.0:8080/api/blog/articles/1e1cbab1-c8b0-4cd7-a8e1-a471aead09e2 |␣
↪python -m json.tool
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
100   195  100   195    0     0  14773      0 --:--:-- --:--:-- --:--:-- 16250
[
    {
        "text": "How do you like this most amazing article?",
        "author": "unknown",
        "uuid": "1e1cbab1-c8b0-4cd7-a8e1-a471aead09e2",
        "post_date": "2016-09-22T23:20:03+00:00",
        "title": "This is an article"
    }
]
```

Delete a Post

Finally, let's delete our post:

```
$ curl -H 'Content-Type: application/json' -X DELETE \
    http://0.0.0.0:8080/api/blog/articles/1e1cbab1-c8b0-4cd7-a8e1-a471aead09e2
```

And, verify using the shell:

```
$ ./manage.py shell
Python 3.5.1 (v3.5.1:37a07cee5969, Dec  5 2015, 21:12:44)
[GCC 4.2.1 (Apple Inc. build 5666) (dot 3)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
(InteractiveConsole)
>>> from blog import models
>>> a = models.Article.query.filter_by(
...     uuid='1e1cbab1-c8b0-4cd7-a8e1-a471aead09e2').first()
>>> a is None
True
```

Authentication

Well, we have an api. However, there are no restrictions, anyone can post any blog post at any time. We need to make an admin version of the api, and protect it with a password.

Create Admin API

Let's create an admin version of the api, that will live at `/api/admin/blog`...

Open up `blog/viewsets.py` and edit it to look like this:

Listing 1.7: viewsets.py

```
from frf import viewsets
from frf.authentication import BasicAuthentication

from blog import models, serializers

from blogapi import conf


def authorize(username, password):
    check_password = conf.get('PASSWORDS', {}).get(username)

    if not password or check_password != password:
        return None

    return username


class ArticleViewSet(viewsets.ModelViewSet):
    serializer = serializers.ArticleSerializer()
    allowed_actions = ('list', 'retrieve')

    def get_qs(self, req, **kwargs):
        return models.Article.query.order_by(
            models.Article.post_date.desc())


class AdminArticleViewSet(ArticleViewSet):
    allowed_actions = ('list', 'retrieve', 'update', 'create', 'delete')
    authentication = (BasicAuthentication(authorize), )

    def create_pre_save(self, req, obj, **kwargs):
        obj.author = req.context.get('user', 'unknown')
```

Let's go over the changes we made. First, we made a duplicate API that just inherits from the first api. It only changes a few things:

1. We are changing what can happen on both apis with `allowed_actions`. For the non-admin api, it can only list and retrieve events, while the admin api can do everything.
2. We are adding `frf.authentication.BasicAuthentication` the admin api. This will decode the basic authentication header, and pass it to our newly defined `authenticate` function, which just checks the username and password against a new setting (that we will add soon) - `PASSWORDS`.
3. Added the `create_pre_save` that just assigns the currently logged in user to the `author` field.

Update the Serializer

Now that we are determining the author automatically, we don't need it to be required in the serializer. Open up `blog/serializers.py` and remove the `required=True` parameter from `author`:

Listing 1.8: serializers.py

```
from frf import serializers

from blog import models


class ArticleSerializer(serializers.ModelSerializer):
    uuid = serializers.UUIDField(read_only=True)
    author = serializers.StringField()  # change this field
    post_date = serializers.ISODateTimeField()
    title = serializers.StringField(required=True)
    text = serializers.StringField(required=True)

    class Meta:
        model = models.Article
```

Add the PASSWORDS Setting

Let's add our `PASSWORDS` setting to the settings file. Open up `blogapi/settings.py` and add the following:

Listing 1.9: settings.py

```
#: Admin passwords
PASSWORDS = {
    'adam': 'onetwo34',
}
```

Update URLS

Now we just need to tell the system about our new setting, so open up `blog/urls.py` and add the new api to `urlpatterns`:

Listing 1.10: urls.py

```
from blog import viewsets

article_viewset = viewsets.ArticleViewSet()
admin_article_viewset = viewsets.AdminArticleViewSet()

urlpatterns = [
    ('/blog/articles/', article_viewset),
    ('/blog/articles/{uuid}/', article_viewset),
    ('/admin/blog/articles/', admin_article_viewset),
    ('/admin/blog/articles/{uuid}/', admin_article_viewset),
    ]
```

Try it Out!

Let's try posting to our old api and see what happens:

```
$ curl -v -H 'Content-Type: application/json' \
   -X POST -d '{"title": "Fantastic article", "text": "..."}' \
   http://0.0.0.0:8080/api/blog/articles/
*   Trying 0.0.0.0...
* Connected to 0.0.0.0 (127.0.0.1) port 8080 (#0)
> POST /api/blog/articles/ HTTP/1.1
> Host: 0.0.0.0:8080
> User-Agent: curl/7.43.0
> Accept: */*
> Content-Type: application/json
> Content-Length: 61
>
* upload completely sent off: 61 out of 61 bytes
< HTTP/1.1 405 Method Not Allowed
< Server: gunicorn/19.6.0
< Date: Fri, 23 Sep 2016 16:46:48 GMT
< Connection: close
< content-type: application/json; charset=UTF-8
< allow: GET
< content-length: 0
<
* Closing connection 0fd
```

As you can see, we got a "Method Not Allowed" response, because we can no longer post to that api. Let's post to the new API and see what happens:

```
$ curl -v -H 'Content-Type: application/json' \
   -X POST -d '{"title": "Fantastic article", "text": "..."}' \
   http://0.0.0.0:8080/api/blog/articles/
*   Trying 0.0.0.0...
* Connected to 0.0.0.0 (127.0.0.1) port 8080 (#0)
> POST /api/admin/blog/articles/ HTTP/1.1
> Host: 0.0.0.0:8080
> User-Agent: curl/7.43.0
> Accept: */*
> Content-Type: application/json
> Content-Length: 61
>
```

```
* upload completely sent off: 61 out of 61 bytes
< HTTP/1.1 401 Unauthorized
< Server: gunicorn/19.6.0
< Date: Fri, 23 Sep 2016 16:49:30 GMT
< Connection: close
< content-length: 698
< content-type: application/json; charset=UTF-8
< www-authenticate: T, o, k, e, n
<
* Closing connection 0
{"title": "Not Authorized", "description": "Not Authorized", "traceback": "Traceback␣
↪(most recent call last):\n  File \"/Users/synic/.virtualenvs/blogapi/lib/python3.5/
↪site-packages/falcon-1.0.0-py3.5.egg/falcon/api.py\", line 189, in __call__\n    ␣
↪responder(req, resp, **params)\n  File \"/Users/synic/Projects/skedup/lib/frf/frf/
↪views.py\", line 68, in on_post\n    self.dispatch('post', req, resp, **kwargs)\n ␣
↪File \"/Users/synic/Projects/skedup/lib/frf/frf/viewsets.py\", line 59, in␣
↪dispatch\n    self.authenticate(method, req, resp, **kwargs)\n  File \"/Users/synic/
↪Projects/skedup/lib/frf/frf/views.py\", line 21, in authenticate\n    challenges=
↪'Token')\nfalcon.errors.HTTPUnauthorized\n"}
```

And we got a "Not Authorized" error message, because we are not supplying a username or password. Let's try doing that:

```
$ curl -v -H 'Content-Type: application/json' \
   -X POST -d '{"title": "Fantastic article", "text": "..."}' \
   --user "adam:onetwo34" \
   http://0.0.0.0:8080/api/blog/articles/
* Connected to 0.0.0.0 (127.0.0.1) port 8080 (#0)
* Server auth using Basic with user 'adam'
> POST /api/admin/blog/articles/ HTTP/1.1
> Host: 0.0.0.0:8080
> Authorization: Basic YWRhbTpvbmV0d28zNA==
> User-Agent: curl/7.43.0
> Accept: */*
> Content-Type: application/json
> Content-Length: 60
>
* upload completely sent off: 60 out of 60 bytes
< HTTP/1.1 201 Created
< Server: gunicorn/19.6.0
< Date: Fri, 23 Sep 2016 16:51:40 GMT
< Connection: close
< content-type: application/json; charset=UTF-8
< content-length: 152
<
* Closing connection 0
[{"text": "...", "post_date": "2016-09-23T16:51:40+00:00", "uuid": "4a7485f6-91cd-
↪407d-8daa-1322b4f909d6", "title": "Fantastic article", "author": "adam"}]
```

And our post is created. You can see that `author` is automatically sent to your username. It's as easy as that!

Permissions

Fred keeps going through and editing your articles, putting a blurb at the bottom stating that tabs are better than spaces.

Fred is wrong.

Let's add the idea of a "superuser" and make it so that non-superusers can create articles, but cannot edit or delete existing articles that don't belong to them.

To do this, we can use FRF's permissions system.

Create a Permission Class

Let's create a `HasEditPermission` class. Open up `blog/permissions.py` and add the following code:

Listing 1.11: permissions.py

```
from frf import permissions

from blogapi import conf


class HasEditPermission(permissions.BasePermission):
    def has_permission(self, req, view, **kwargs):
        user = req.context.get('user', None)
        if not user:
            return False

        if req.method in ('PUT', 'PATCH', 'DELETE'):
            obj = view.get_obj(req, **kwargs)
            return user == obj.author or user in conf.get('SUPERUSERS', [])

        return True
```

Update Viewsets

Let's add this new permission to the admin viewset. Open up `blog/viewsets.py` and make the following change:

Listing 1.12: viewsets.py

```
from frf import viewsets
from frf.authentication import BasicAuthentication

from blog import models, serializers, permissions

from blogapi import conf


def authorize(username, password):
    check_password = conf.get('PASSWORDS', {}).get(username)

    if not password or check_password != password:
        return None

    return username


class ArticleViewSet(viewsets.ModelViewSet):
    serializer = serializers.ArticleSerializer()
    allowed_actions = ('list', 'retrieve')

    def get_qs(self, req, **kwargs):
        return models.Article.query.order_by(
```

```
            models.Article.post_date.desc())


class AdminArticleViewSet(ArticleViewSet):
    allowed_actions = ('list', 'retrieve', 'update', 'create', 'delete')
    authentication = (BasicAuthentication(authorize), )
    permissions = (permissions.HasEditPermission(), )

    def create_pre_save(self, req, obj, **kwargs):
        obj.author = req.context.get('user', 'unknown')
```

Add the SUPERUSERS Setting

And, finally, we need to add the list of superusers to the settings file.

Open up `blogapi/settings.py` and add the following:

Listing 1.13: settings.py

```
#: Superusers
SUPERUSERS = ['synic', 'yourusernamehere', ]
```

Filtering

Let's add a filter to our api, so we can view posts from specific authors by passing a query string argument, like: `http://0.0.0.0:8080/api/blog/articles/?author=adam`.

Open up `blog/viewsets.py` and edit it to look like this:

Listing 1.14: viewsets.py

```
from frf import viewsets, filters
from frf.authentication import BasicAuthentication

from blog import models, serializers

from blogapi import conf


def authorize(username, password):
    check_password = conf.get('PASSWORDS', {}).get(username)

    if not password or check_password != password:
        return None

    return username


class ArticleViewSet(viewsets.ModelViewSet):
    serializer = serializers.ArticleSerializer()
    allowed_actions = ('list', 'retrieve')
    filters = [filters.FieldMatchFilter(models.Article.author)]

    def get_qs(self, req, **kwargs):
        return models.Article.query.order_by(
            models.Article.post_date.desc())
```

```
class AdminArticleViewSet(ArticleViewSet):
    allowed_actions = ('list', 'retrieve', 'update', 'create', 'delete')
    authentication = [BasicAuthentication(authorize)]

    def create_pre_save(self, req, obj, **kwargs):
        obj.author = req.context.get('user', 'unknown')
```

Bam! Now we can filter by author.

Custom Filters

Let's make a custom filter that limits the number of blog articles displayed by listing articles to the 5 most recent, unless a flag (`?filter=all`) is in the query string. Again, open up `blog/viewsets.py` and edit it to look like this:

Listing 1.15: viewsets.py

```
from frf import viewsets, filters
from frf.authentication import BasicAuthentication

from blog import models, serializers

from blogapi import conf


class RecentFlagFilter(filters.FlagFilter):
    def __init__(self, *args, **kwargs):
        super().__init__(flag='all')

    def filter_default(self, req, qs):
        return qs.limit(conf.get('DEFAULT_ARTICLE_COUNT', 5))


def authorize(username, password):
    check_password = conf.get('PASSWORDS', {}).get(username)

    if not password or check_password != password:
        return None

    return username


class ArticleViewSet(viewsets.ModelViewSet):
    serializer = serializers.ArticleSerializer()
    allowed_actions = ('list', 'retrieve')
    filters = [
        filters.FieldMatchFilter(models.Article.author),
        RecentFlagFilter(),
        ]

    def get_qs(self, req, **kwargs):
        return models.Article.query.order_by(
            models.Article.post_date.desc())
```

```
class AdminArticleViewSet(ArticleViewSet):
    allowed_actions = ('list', 'retrieve', 'update', 'create', 'delete')
    authentication = [BasicAuthentication(authorize)]

    def create_pre_save(self, req, obj, **kwargs):
        obj.author = req.context.get('user', 'unknown')
```

Here, we are using a specific type of filter called a *Flag Filter*. Flag filters apply a filter when no *flag* is present (flags are denoted by `?filter=[flag]`, and a different filter when the flag is present. Here, we limit our queryset by the setting `DEFAULT_ARTICLE_COUNT` (which doesn't exist, but defaults to 5) when the flag is NOT present, and we do nothing when the flag is present. Thus, when we put `?filter=all` on the query string, the queryset is NOT limited, and we get all the results back, otherwise, we only see the latest 5.

By using the setting `DEFAULT_ARTICLE_COUNT` here, we can change the default number of articles returned by changing that setting in our `settings.py`.

Pagination

What if, over the years, we end up having thousands of articles - it would be too much data to return to the api in one go. Lets add basic pagination support to our viewset.

Open up `blog/viewsets.py` and add the `paginate` line from below:

Listing 1.16: viewsets.py

```
# ... snip

class ArticleViewSet(viewsets.ModelViewSet):
    serializer = serializers.ArticleSerializer()
    paginate = (10, 100)  # add this line!
    allowed_actions = ('list', 'retrieve')

# ... snip
```

And, that's it! Your api will now page by 10, and your users can request a certain page by passing `?page=[num]` on the query string. Note that pagination starts at page 1, not page 0.

Your users can also change the number of results they receive per page, by passing the query string parameter `?per_page=[num]`. In our line above, we have `paginate = (10, 100)`. In that scenario, the `10` is the default number of items per page, and `100` is the *maximum* number that can be requested by passing `per_page`. Easy!

Renderers

Let's give our users some hints about what page they are on, by changing the output of a list request. We want the output of `/api/blog/articles/` to look like this:

```
{
  "results": [
    {
      "post"_date: "2016-09-23T16:53:34+00:00",
      "author": "adam",
      "text": "...",
      "title": "Fantastic article",
      "uuid": "6068530e-b10c-4cc4-ba4f-b6cf1b041a85"
    },
    {
```

```
      "post"_date: "2016-09-23T16:56:44+00:00",
      "author": "adam",
      "text": "another article",
      "title": "Another Article",
      "uuid": "8068530a-a10c-fcc4-ba4f-c6cf1b041a8f"
    },
  ],
  "meta": {
    "total": 2,
    "page": 1,
    "per_page": 10,
    "page_limit": 100,
  }
}
```

We call objects that change the output like this *renderers*. Open up `blog/viewsets.py` and add the "list meta" renderer, like this:

Listing 1.17: viewsets.py

```
from frf import viewsets, filters, renderers
from frf.authentication import BasicAuthentication

from blog import models, serializers

# ... snip

class ArticleViewSet(viewsets.ModelViewSet):
    serializer = serializers.ArticleSerializer()
    paginate = (10, 100)
    allowed_actions = ('list', 'retrieve')
    renderers = [renderers.ListMetaRenderer()]

# ... snip
```

And, now our output should have paging information:

```
$ curl -H 'Content-Type: application/json' -X GET \
    http://0.0.0.0:8080/api/blog/articles/ | python -m json.tool
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
100   830  100   830    0     0  29363      0 --:--:-- --:--:-- --:--:-- 28620
{
  "results": [
    {
      "uuid": "6068530e-b10c-4cc4-ba4f-b6cf1b041a85",
      "title": "Fantastic article",
      "post_date": "2016-09-23T16:53:34+00:00",
      "author": "adam",
      "text": "..."
    },
    {
      "uuid": "4a7485f6-91cd-407d-8daa-1322b4f909d6",
      "title": "Fantastic article",
      "post_date": "2016-09-23T16:51:40+00:00",
      "author": "adam",
      "text": "..."
    },
```

```
    {
      "uuid": "9ba4dffc-b1b7-426a-81b3-a25a67c55666",
      "title": "title",
      "post_date": "2016-09-23T01:27:37+00:00",
      "author": "fred",
      "text": "text"
    },
    {
      "uuid": "ed5a8f18-51c7-492e-b778-474bf087e23f",
      "title": "title",
      "post_date": "2016-09-23T01:27:06+00:00",
      "author": "adam",
      "text": "text"
    },
    {
      "uuid": "aa72fe4d-0df0-471a-9519-8f3fbbd6615d",
      "title": "another title",
      "post_date": "2016-09-23T01:26:08+00:00",
      "author": "guy",
      "text": "guy's article"
    }
  ],
  "meta": {
    "per_page": 10,
    "page": 1,
    "page_limit": 100,
    "total": 5
  }
}
```

CHAPTER 2

API Reference

Serializers

Serializer Fields

ViewSets

Application Setup

Database Access

Configuration

Cache

Utilities

FRF comes with various utilities that can make your life a bit easier.

Cli Utilities

Date/Time Utilities

Encryption

Importing

Filters

Filters allow you to filter your queryset based on criteria such as query string parameters, headers, etc.

Renderers

Renders allow you to change the output of a view/viewset.

Parsers

Parsers allow you to modify input data before your view/viewset processes it.

Decorators

CHAPTER 3

Indices and tables

- genindex
- modindex
- search