Django Class Notes

Clarusway 

DRF - Pagination, Filter, Search

Nice to have VSCode Extentions:

- Djaneiro - Django Snippets (Be carefull about other conflicting extentions!)

Needs

- Python, add the path environment variable
- pip
- virtualenv

Summary

- Spin up the project
- Pagination
  - PageNumberPagination
  - LimitOffsetPagination
  - CursorPagination
- Filtering
  - overriding the .get_queryset() method
  - Generic Filtering
- SearchFilter
- OrderingFilter

Spin up the project

- Create a working directory, name it as you wish, cd to new directory
- Create virtual environment as a best practice:

```
python3 -m venv env # for Windows or
python -m venv env # for Windows
virtualenv env # for Mac/Linux or;
virtualenv yourenv -p python3 # for Mac/Linux
```

- Activate scripts:

```
.\env\Scripts\activate  # for Windows
source env/bin/activate  # for MAC/Linux
```

- See the (env) sign before your command prompt.
- Install requirements:

```
pip install -r requirements.txt
```

Secure your project

.gitignore

Add standard .gitignore file to the project root directory.

Do that before adding your files to staging area, else you will need extra work to unstage files to be able to ignore them.

python-decouple

- To use python decouple in this project, first install it:

```
pip install python-decouple
```

- For more information about python-decouple
- Import the config object on `settings.py` file:

```
from decouple import config
```

- Create .env file on root directory. We will collect our variables in this file.

```
SECRET_KEY=o5o9...
```

- Retrieve the configuration parameters in `settings.py`:

```
SECRET_KEY = config('SECRET_KEY')
```

- Now you can send you project to the github, but be sure you added a .gitignore file which has .env on it.
- Manage migrations.

```
python manage.py migrate
```

- In order to login admin site, we need to create a user who can login to the admin site. Run the following command:

```
python manage.py createsuperuser  # or with the parameters
python manage.py createsuperuser --username admin --email admin@mail.com
```

- Enter your desired username, email adress, and password twice.
- Run server.
- Go to http://127.0.0.1:8000/admin/ You should see the admin's login screen.

Pagination

REST framework includes support for customizable pagination styles. This allows you to modify how large result sets are split into individual pages of data.

The pagination style may be set globally, using the DEFAULT_PAGINATION_CLASS and PAGE_SIZE setting keys.

```
REST_FRAMEWORK = {
    'DEFAULT_PAGINATION_CLASS': ...
    'PAGE_SIZE': ...
}
```

- This will be effective for all of our views.

PageNumberPagination

This pagination style accepts a single page number in the request query parameters.

```
GET https://api.example.org/accounts/?page=4
GET http://127.0.0.1:8000/?page=2
```

- To enable the PageNumberPagination style globally, use the following configuration, and set the PAGE_SIZE as desired:

```
REST_FRAMEWORK = {
    'DEFAULT_PAGINATION_CLASS': 'rest_framework.pagination.PageNumberPagination',
    'PAGE_SIZE': 2
}
```

- On GenericAPIView subclasses you may also set the `pagination_class` attribute to select PageNumberPagination on a per-view basis.

The PageNumberPagination class includes a number of attributes that may be overridden to modify the pagination style.

```
from .pagination import SmallPageNumberPagination

class StudentListCreateAPIView(generics.ListCreateAPIView):

    queryset = Student.objects.all()
    serializer_class = StudentSerializer
    pagination_class = SmallPageNumberPagination
```

If you create classes on the pagination.py:

```
from rest_framework.pagination import PageNumberPagination


class SmallPageNumberPagination(PageNumberPagination):
    page_size = 2
    # page_size_query_param="sayfa"

class LargePageNumberPagination(PageNumberPagination):
    page_size = 5
```

LimitOffsetPagination

This pagination style mirrors the syntax used when looking up multiple database records. The client includes both a "limit" and an "offset" query parameter.

- The limit indicates the maximum number of items to return, and is equivalent to the page_size in other styles.

- The offset indicates the starting position of the query in relation to the complete set of unpaginated items.

```
GET https://api.example.org/accounts/?limit=20&offset=50
GET http://127.0.0.1:8000/?limit=1&offset=2
```

- To enable the LimitOffsetPagination style globally, use the following configuration:

```
REST_FRAMEWORK = {
    'DEFAULT_PAGINATION_CLASS': 'rest_framework.pagination.LimitOffsetPagination',
    # 'PAGE_SIZE': 1
}
```

- Optionally, you may also set a PAGE_SIZE key. If the PAGE_SIZE parameter is also used then the limit query parameter will be optional, and may be omitted by the client.
- On GenericAPIView subclasses you may also set the `pagination_class` attribute to select LimitOffsetPagination on a per-view basis.

The LimitOffsetPagination class includes a number of attributes that may be overridden to modify the pagination style.

```
from .pagination import MyLimitOffsetPagination

class StudentListCreateAPIView(generics.ListCreateAPIView):

    queryset = Student.objects.all()
    serializer_class = StudentSerializer
    pagination_class = MyLimitOffsetPagination
```

If you create classes on the pagination.py:

```
from rest_framework.pagination import LimitOffsetPagination


class MyLimitOffsetPagination(LimitOffsetPagination):
    # default_limit = 2
    limit_query_param = 'how_many'  # Defaults to 'limit'.
```

CursorPagination

The cursor-based pagination presents an opaque "cursor" indicator that the client may use to page through the result set. This pagination style only presents forward and reverse controls, and does not allow the client to navigate to arbitrary positions.

Cursor based pagination requires that there is a unique, unchanging ordering of items in the result set.

Cursor based pagination is more complex than other schemes. However it does provide the following benefits:

- Provides a consistent pagination view. When used properly CursorPagination ensures that the client will never see the same item twice when paging through records, even when new items are being inserted by other clients during the pagination process.
- Supports usage with very large datasets. With extremely large datasets pagination using offset-based pagination styles may become inefficient or unusable. Cursor based pagination schemes instead have fixed-time properties, and do not slow down as the dataset size increases.

To enable the CursorPagination style globally, use the following configuration, modifying the PAGE_SIZE as desired:

```
REST_FRAMEWORK = {
    'DEFAULT_PAGINATION_CLASS': 'rest_framework.pagination.CursorPagination',
    'PAGE_SIZE': 3
}
```

Proper use of cursor based pagination requires a little attention to detail. You'll need to think about what ordering you want the scheme to be applied against. The default is to order by "-created". This assumes that there must be a 'created' timestamp field on the model instances, and will present a "timeline" style paginated view, with the most recently added items first.

You can modify the ordering by overriding the 'ordering' attribute on the pagination class, or by using the OrderingFilter filter class together with CursorPagination. When used with OrderingFilter you should strongly consider restricting the fields that the user may order by.

On GenericAPIView subclasses you may also set the pagination_class attribute to select CursorPagination on a per-view basis.

```
from .pagination import MycursorPagination

class StudentListCreateAPIView(generics.ListCreateAPIView):

    queryset = Student.objects.all()
    serializer_class = StudentSerializer
    pagination_class = MycursorPagination
```

If you create classes on the pagination.py:

```
from rest_framework.pagination import CursorPagination


class MycursorPagination(CursorPagination):
```

```
    page_size=1
    ordering = "number"
```

We can use our pagination classes on `pagination.py` globally:

```
REST_FRAMEWORK = {
    'DEFAULT_PAGINATION_CLASS': 'fscohort.pagination.SmallPageNumberPagination',
}
```

- You can get `UnorderedObjectListWarning` on your console when using SmallPageNumberPagination. To remove this warning you can add order_by() function to your list view:

```
class StudentListCreateAPIView(generics.ListCreateAPIView):

    queryset = Student.objects.all().order_by('id')
    serializer_class = StudentSerializer
```

Filtering

The default behavior of REST framework's generic list views is to return the entire queryset for a model manager. Often you will want your API to restrict the items that are returned by the queryset.

The simplest way to filter the queryset of any view that subclasses GenericAPIView is to override the .get_queryset() method.

Overriding this method allows you to customize the queryset returned by the view in a number of different ways:

- Filtering against the current user
- Filtering against the URL
- Filtering against query parameters

overriding the .get_queryset() method

We can override .get_queryset() of our generic API view to deal with URLs such as:

```
http://localhost:8000/?course=FS
```

and filter the queryset only if the username parameter is included in the URL:

```
def get_queryset(self):
    queryset = Student.objects.all()
    course = self.request.query_params.get('course')
    if course is not None:
```

```
        mycourse = Course.objects.get(title=course)
        queryset = queryset.filter(course=mycourse.id)
    return queryset
```

Or filter by name:

```
def get_queryset(self):
    queryset = Student.objects.all()
    first_name = self.request.query_params.get('first_name')
    if first_name is not None:
        # This only brings exact names:
        # queryset = queryset.filter(first_name=first_name)
        # This is more flexible:
        queryset = queryset.filter(first_name__contains=first_name)
    return queryset
```

Generic Filtering

REST framework also includes support for generic filtering backends that allow you to easily construct complex searches and filters.

Generic filters can also present themselves as HTML controls in the browsable API and admin API.

- To use DjangoFilterBackend, install django-filter.

```
pip install django-filter
```

- Then add 'django_filters' to Django's INSTALLED_APPS:

```
INSTALLED_APPS = [
    ...
    'django_filters',
    ...
]
```

- The default filter backends may be set globally, using the DEFAULT_FILTER_BACKENDS setting.

```
REST_FRAMEWORK = {
    'DEFAULT_FILTER_BACKENDS':
['django_filters.rest_framework.DjangoFilterBackend']
}
```

- If all you need is simple equality-based filtering, you can set a filterset_fields attribute on the view, or viewset, listing the set of fields you wish to filter against.

```
class StudentListCreateAPIView(generics.ListCreateAPIView):

    queryset = Student.objects.all().order_by('-id')
    serializer_class = StudentSerializer
    # Add filterset_fields:
    filterset_fields = ['first_name',]
```

- This will automatically create a FilterSet class for the given fields, and will allow you to make requests. Now you can see filter button on your API browser.
- You can use "overriding the .get_queryset() method" and "Generic Filtering" at the same time.
- The default filter backends may be set locally inside views:

```
from django_filters.rest_framework import DjangoFilterBackend

class StudentListCreateAPIView(generics.ListCreateAPIView):

    queryset = Student.objects.all().order_by('-id')
    serializer_class = StudentSerializer

    filter_backends = [DjangoFilterBackend]
    filterset_fields = ['first_name',]
```

SearchFilter

The SearchFilter class supports simple single query parameter based searching, and is based on the Django admin's search functionality.

When in use, the browsable API will include a SearchFilter control.

The SearchFilter class will only be applied if the view has a `search_fields` attribute set. The `search_fields` attribute should be a list of names of text type fields on the model, such as CharField or TextField.

```
from rest_framework.filters import SearchFilter

class StudentListCreateAPIView(generics.ListCreateAPIView):

    queryset = Student.objects.all().order_by('-id')
    serializer_class = StudentSerializer

    # Adds for filter:
    filter_backends = [SearchFilter]
    search_fields = ['first_name','last_name' ]
```

- Filter and search can be used at the same time.

OrderingFilter

The OrderingFilter class supports simple query parameter controlled ordering of results.

By default, the query parameter is named 'ordering', but this may by overridden with the ORDERING_PARAM setting.

```
from rest_framework.filters import OrderingFilter

class StudentListCreateAPIView(generics.ListCreateAPIView):

    queryset = Student.objects.all().order_by('-id')
    serializer_class = StudentSerializer

    filter_backends = [OrderingFilter]
    ordering_fields = ['first_name', ]
    # ordering_fields = '__all__'
```

☺ **Happy Coding!** ✍

Clarusway