

DESENVOLVIMENTO DE APIs COM

django

REST

framework



PYTHON
ACADEMY



EBOOK

DESENVOLVIMENTO DE APIS COM DJANGO REST FRAMEWORK

ANTES DE COMEÇARMOS...

Quer dominar a Linguagem Python e fazer parte da parcela de programadores **mais bem paga** do mercado de TI?

Se **sim**, conheça a Jornada Python e torne-se um especialista em Python e domine as principais tecnologias do mercado - como o poderoso framework web **Django** - através de projetos práticos e reais, mesmo que esteja começando do absoluto zero.

+28h
de conteúdo
(e crescendo)

17 Módulos
de Python
5 Módulos
de Django

- + Suporte à dúvidas
- + Certificado de Conclusão
- + Ebooks exclusivos
- + Atualizações futuras sem custo adicional

Domine as **principais tecnologias** web do mercado...



PYTHON

django

DJANGO



HTML



CSS



JAVASCRIPT



BOOTSTRAP

E seja requisitado por recrutadores **nacionais** e **internacionais**!

[CLIQUE AQUI E CONHEÇA A JORNADA PYTHON](#)

 **JORNADA
PYTHON**

SUMÁRIO

INTRODUÇÃO	3
PROJETO QUE VAMOS DESENVOLVER	4
CONFIGURAÇÃO DO PROJETO	6
MODELAGEM	10
SERIALIZERS	14
VIEWSETS	16
ROUTERS	21
INTERFACE NAVEGÁVEL	25
FILTROS, BUSCA TEXTUAL E ORDENAÇÃO	28
FILTROS DE BUSCA	28
BUSCA TEXTUAL	30
ORDENAÇÃO DE RESULTADOS	31
CONFIGURAÇÕES ESPECÍFICAS	34

INTRODUÇÃO

Meu caro Padawan do caminho Python, seja bem vindo a mais um material exclusivo da Python Academy!

Nesse Ebook, vamos tratar de uma biblioteca muito poderosa para construções de APIs: o *Django Rest Framework*, que aqui vamos chamar de **DRF**.

Vamos aprender que é possível combinar Python e Django para desenvolver APIs web de uma forma **simples, rápida e flexível**.

Para começar, vamos listar abaixo alguns motivos para usar o DRF:

- Provê uma interface navegável para debugar sua API;
- Possui diversas estratégias de autenticação, incluindo pacotes para *OAuth1* e *OAuth2*;
- Realiza serialização de objetos de fontes ORM (bancos de dados) e não-ORM (classes próprias);

- Tem extensa documentação disponível e grande comunidade de usuários;
- É utilizado por grandes corporações, como Heroku, EventBrite, Mozilla e Red Hat;
- **E O MELHOR DE TUDO:** utiliza o nosso querido Django como base!

Para utilizar o DRF é interessante que você já possua um conhecimento prévio no desenvolvimento de aplicações Web utilizando o Django.

AINDA NÃO É CRAQUE??

Não tem problema! Aqui na Python Academy você conta com o melhor material sobre Django em nosso **Blog**! Acesse alguns conteúdos já disponíveis:

- [Django: introdução ao framework](#)
- [Django: A Camada Model](#)
- [Django: A Camada View](#)
- [Django: A Camada Template](#)

Após essa breve introdução sobre o Django Rest Framework, vamos aprofundar nosso estudo, e a melhor forma de aprender uma nova ferramenta é colocando a mão no código. E vamos fazer isso desenvolvendo um mini projeto, ou seja, vamos verificar na prática a usabilidade do **DRF**.

PROJETO QUE VAMOS DESENVOLVER

Nesse Ebook vamos desenvolver uma API para consulta de um tipo de investimento financeiro: os **Fundos de Investimentos Imobiliários** (ou FIIs).

Caso você não saiba o que são Fundos Imobiliários, segue uma pequena explicação:

“Fundos Imobiliários (FIs) são fundos de investimento destinados à aplicação em empreendimentos imobiliários. Isso inclui, além da aquisição de direitos reais sobre bens imóveis, o investimento em títulos relacionados ao mercado imobiliário, como letras de crédito imobiliário (LCI), letras hipotecárias (LH), cotas de outros FI, certificados de potencial adicional de construção (CEPAC), certificados de recebíveis imobiliários (CRI), entre outros previstos na regulamentação.” (Retirado de: investidor.gov.br).

Neste projeto, vamos prover formas de cadastrar, buscar, atualizar e remover Fundos Imobiliários de um Banco de Dados através de Requisições HTTP, em uma API Web!

Mas primeiro, vamos iniciar criando a estrutura do projeto e configurando o DRF!

CAPÍTULO 2

CONFIGURAÇÃO DO PROJETO

Para começar, vamos nomear nosso projeto. Ele será chamado de **Fundsfinder** (“buscador” de fundos).

Agora vamos à mão na massa!

```
# Cria a pasta e a acessa  
mkdir fundsfinder && cd fundsfinder  
  
# Cria o ambiente virtual  
virtualenv venv  
  
# Ativa o ambiente virtual  
source venv/bin/activate  
  
# Instala Django e DRF  
pip install django djangorestframework
```


Feito isso, nós: criamos a pasta do projeto, criamos e ativamos um ambiente virtual utilizando o Virtualenv (**caso não saiba o que é, temos um post completo em nosso Blog, [acesse aqui!](#)**), e instalamos as dependências (Django e DRF).

E agora, vamos criar um novo *app* Django para separar as responsabilidades da API que vamos desenvolver.

Vamos chamá-lo de `api`.

Para isso usaremos o comando `startapp` do `django-admin` na raiz do projeto (onde se encontra o arquivo `manage.py`), dessa forma:

```
python3 manage.py startapp api
```

Vamos aproveitar e criar a estrutura inicial do banco de dados com:

```
python3 manage.py migrate
```

E agora, teremos a seguinte estrutura:

```
$ tree -L 3 -I venv
.
├── api
│   ├── admin.py
│   ├── apps.py
│   ├── __init__.py
│   ├── migrations
│   │   └── __init__.py
│   ├── models.py
│   ├── tests.py
│   └── views.py
├── db.sqlite3
├── fundsfinder
│   ├── asgi.py
│   ├── __init__.py
│   ├── settings.py
│   ├── urls.py
│   └── wsgi.py
└── manage.py

3 directories, 14 files
```

Vamos executar o servidor local para verificar se está tudo correto com o comando:

```
python manage.py runserver
```

Ao acessar <http://localhost:8000> no browser, a seguinte tela deve ser mostrada:

django

View [release notes](#) for Django 3.2



The install worked successfully! Congratulations!

You are seeing this page because `DEBUG=True` is in your settings file and you have not configured any URLs.



Django Documentation
Topics, references, & how-to's



Tutorial: A Polling App
Get started with Django



Django Community
Connect, get help, or contribute

Agora, vamos adicionar um super usuário com o comando `createsuperuser` (lembre-se da senha que você escolher):

```
Python manage.py createsuperuser --email admin@fundsfinder.com  
--username admin
```

Só falta uma coisa para terminarmos as configurações iniciais do nosso projeto: adicionar tudo ao `settings.py`.

E para isso, precisamos abrir o arquivo `fundsfinder/settings.py` e adicionar os *apps* `api`, `fundsfinder` e `rest_framework` (necessário para funcionamento do DRF) à configuração `INSTALLED_APPS`, ficando assim:

```
INSTALLED_APPS = [  
    'django.contrib.admin',  
    'django.contrib.auth',  
    'django.contrib.contenttypes',  
    'django.contrib.sessions',  
    'django.contrib.messages',  
    'django.contrib.staticfiles',  
    'rest_framework',  
    'fundsfinder',  
    'api'  
]
```

Pronto!

Com isso temos a estrutura inicial para finalmente começarmos nosso projeto.

O processo de desenvolvimento de aplicações que utilizam o Django Rest Framework geralmente seguem a seguinte linha de implementação:

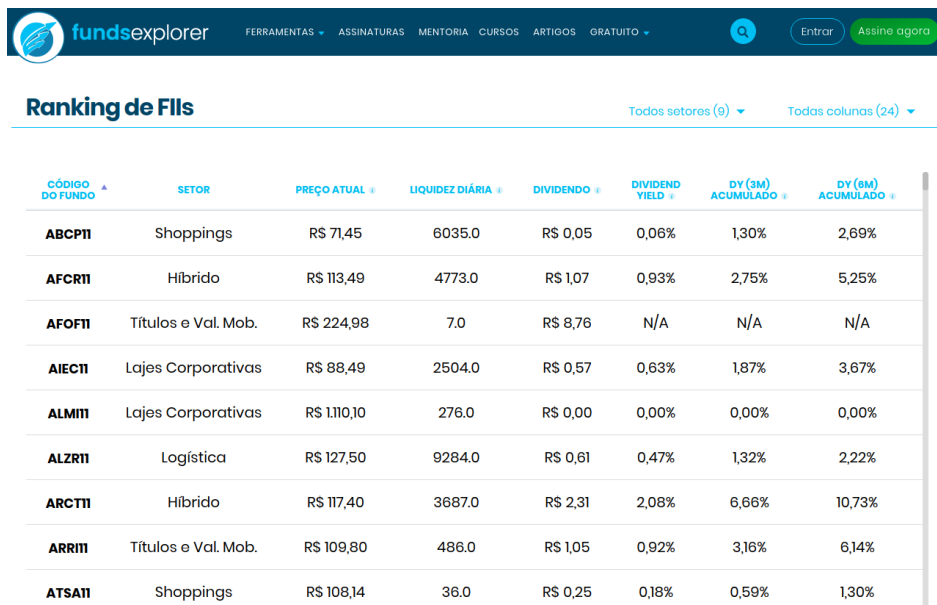
1. Modelagem;
2. *Serializers*;
3. *ViewStes*;
4. *Routers*.

Nesse Ebook vamos tratar de cada um destes passos e agora vamos começar com a parte da **Modelagem**, que consiste em definir os campos, comportamentos e Relacionamentos **de** e **entre** cada Entidade do nosso sistema!

MODELAGEM

Como vamos fazer um sistema para busca e listagem de Fundos Imobiliários, a modelagem de dados do nosso projeto deve refletir campos que façam sentido.

Para auxiliar nessa tarefa, vamos utilizar alguns parâmetros de uma tabela muito interessante do site **FundsExplorer** (acesse esta tabela através [deste link](#)):



CÓDIGO DO FUNDO	SETOR	PREÇO ATUAL	LIQUIDEZ DIÁRIA	DIVIDENDO	DIVIDEND YIELD	DY (3M) ACUMULADO	DY (6M) ACUMULADO
ABCP11	Shoppings	R\$ 71,45	6035.0	R\$ 0,05	0,06%	1,30%	2,69%
AFCR11	Híbrido	R\$ 113,49	4773.0	R\$ 1,07	0,93%	2,75%	5,25%
AFOF11	Títulos e Val. Mob.	R\$ 224,98	7.0	R\$ 8,76	N/A	N/A	N/A
AIEC11	Lajes Corporativas	R\$ 88,49	2504.0	R\$ 0,57	0,63%	1,87%	3,67%
ALMI11	Lajes Corporativas	R\$ 110,10	276.0	R\$ 0,00	0,00%	0,00%	0,00%
ALZR11	Logística	R\$ 127,50	9284.0	R\$ 0,61	0,47%	1,32%	2,22%
ARCT11	Híbrido	R\$ 117,40	3687.0	R\$ 2,31	2,08%	6,66%	10,73%
ARRI11	Títulos e Val. Mob.	R\$ 109,80	486.0	R\$ 1,05	0,92%	3,16%	6,14%
ATSA11	Shoppings	R\$ 108,14	36.0	R\$ 0,25	0,18%	0,59%	1,30%

Para o nosso projeto, vamos usar os seguintes atributos:

- **Código do Fundo:** código identificador do Fundo.
- **Setor:** setor do Fundo Imobiliário.
- **Dividend Yield médio (12 meses):** Dividend Yield mostra quanto um fundo paga de Dividendos (divisão de lucros) sobre o valor atual da cota.
- **Vacância Financeira:** importante métrica que mostra ao investidor quantos ativos (prédios, galpões, escritórios) estão desocupados.
- **Quantidade de Ativos:** quantos ativos são administrados pelo Fundo.

Com isso em mãos, podemos criar a modelagem da entidade `FundoImobiliário`.

Iremos utilizar o ORM (Mapeamento Objeto-Relacional) do próprio Django.

Temos um post completo sobre a camada Model do Django, [confira aqui!](#)

Ao analisar essa tabela do FundsExplorer, nossa modelagem pode ser implementada da seguinte forma (`api/models.py`):

```
from django.db import models
import uuid

class FundoImobiliario(models.Model):
    SETOR_CHOICES = [
        ('hospital', 'Hospital'), ('hotel', 'Hotel'), ('hibrido',
        'Híbrido'),
        ('lajes_corporativas', 'Lajes Corporativas'),
        ('logistica', 'Logística'), ('outros', 'Outros'),
        ('residencial', 'Residencial'),
        ('titulos_valores_mobiliarios', 'Títulos e Val. Mob.')
    ]

    id = models.UUIDField(
        primary_key=True,
```

```
default=uuid.uuid4,  
null=False,  
blank=True)  
  
codigo = models.CharField(  
    max_length=8,  
    null=False,  
    blank=False)  
  
setor = models.CharField(  
    max_length=30,  
    null=False,  
    blank=False,  
    choices=SETOR_CHOICES)  
  
dividend_yield_medio_12m = models.DecimalField(  
    null=False,  
    blank=False,  
    max_digits=5,  
    decimal_places=2)  
  
vacancia_financeira = models.DecimalField(  
    null=False,  
    blank=False,  
    max_digits=5,  
    decimal_places=2)  
  
vacancia_fisica = models.DecimalField(  
    null=False,  
    blank=False,  
    max_digits=5,  
    decimal_places=2)  
  
quantidade_ativos = models.IntegerField(  
    null=False,  
    blank=False,  
    default=0)
```

Com a nossa modelagem criada, precisamos gerar o arquivo de Migrações para atualizar o banco de dados.

Fazemos isso com o comando `makemigrations` (também temos um post no blog sobre isso, [veja!](#)).

Execute:

```
python3 manage.py makemigrations api
```

Agora vamos aplicar a migração ao Banco de Dados com o comando `migrate` (saiba mais sobre esse comando do Django no nosso [Blog!](#)).

Para isso, vamos executar o seguinte comando:

```
python3 manage.py migrate
```

Com a modelagem pronta, podemos ir para o próximo passo: definir os **Serializers**!

SERIALIZERS

Os *serializers* do DRF são componentes **essenciais** do *framework*.

Eles traduzem entidades complexas, como *querysets* e instâncias de classes em representações simples que podem ser usadas no tráfego da web, como JSON e XML.

Esse processo é chamado de **Serialização**.

Serializers também servem para fazer o caminho contrário: a **Desserialização**. Ou seja, transformam representações simples (como JSON e XML) em representações complexas, instanciando objetos, por exemplo.

E agora vamos aplicar este conceito em nosso projeto. E para isso, vamos criar o arquivo onde vão ficar os **Serializers** da nossa API.

Vamos criar um arquivo chamado `serializers.py` dentro da pasta `api/`.

O DRF disponibiliza diversos tipos de *serializers* que podemos utilizar, como:

- **BaseSerializer**: classe base para construção de *serializers* mais genéricos.
- **ModelSerializer**: auxilia a criação de serializadores baseados em Modelos.

- `HyperlinkedModelSerializer`: similar ao `ModelSerializer`, contudo retorna um link para representar o relacionamento entre entidades (`ModelSerializer` retorna, por padrão, o `id` da entidade relacionada).

Iremos utilizar o `ModelSerializer` na entidade `FundoImobiliario`.

Para isso, precisamos declarar sobre qual modelo aquele serializador irá operar e quais os campos que ele deve se preocupar. Nosso `Serializer` pode ser implementado da seguinte maneira:

```
from rest_framework import serializers
from api.models import FundoImobiliario

class FundoImobiliarioSerializer(serializers.ModelSerializer):
    class Meta:
        model = FundoImobiliario
        fields = [
            'id',
            'codigo',
            'setor',
            'dividend_yield_medio_12m',
            'vacancia_financeira',
            'vacancia_fisica',
            'quantidade_ativos'
        ]
```

Nesse `Serializer`:

- `model = FundoImobiliario` define qual modelo esse *serializer* deve serializar.
- `fields` define os campos que serão serializados.

*** Obs: é possível definir que todos os campos da entidade de modelo devem ser serializados usando `fields = '_all_'`, contudo eu prefiro mostrar os campos explicitamente.**

Com isso, finalizamos mais uma etapa do passo a passo do DRF!

Vamos à terceira etapa: a criação de `Views`.

VIEWSETS

As *ViewSets* definem quais operações REST estarão disponíveis e como seu sistema vai responder às solicitações direcionadas à sua API.

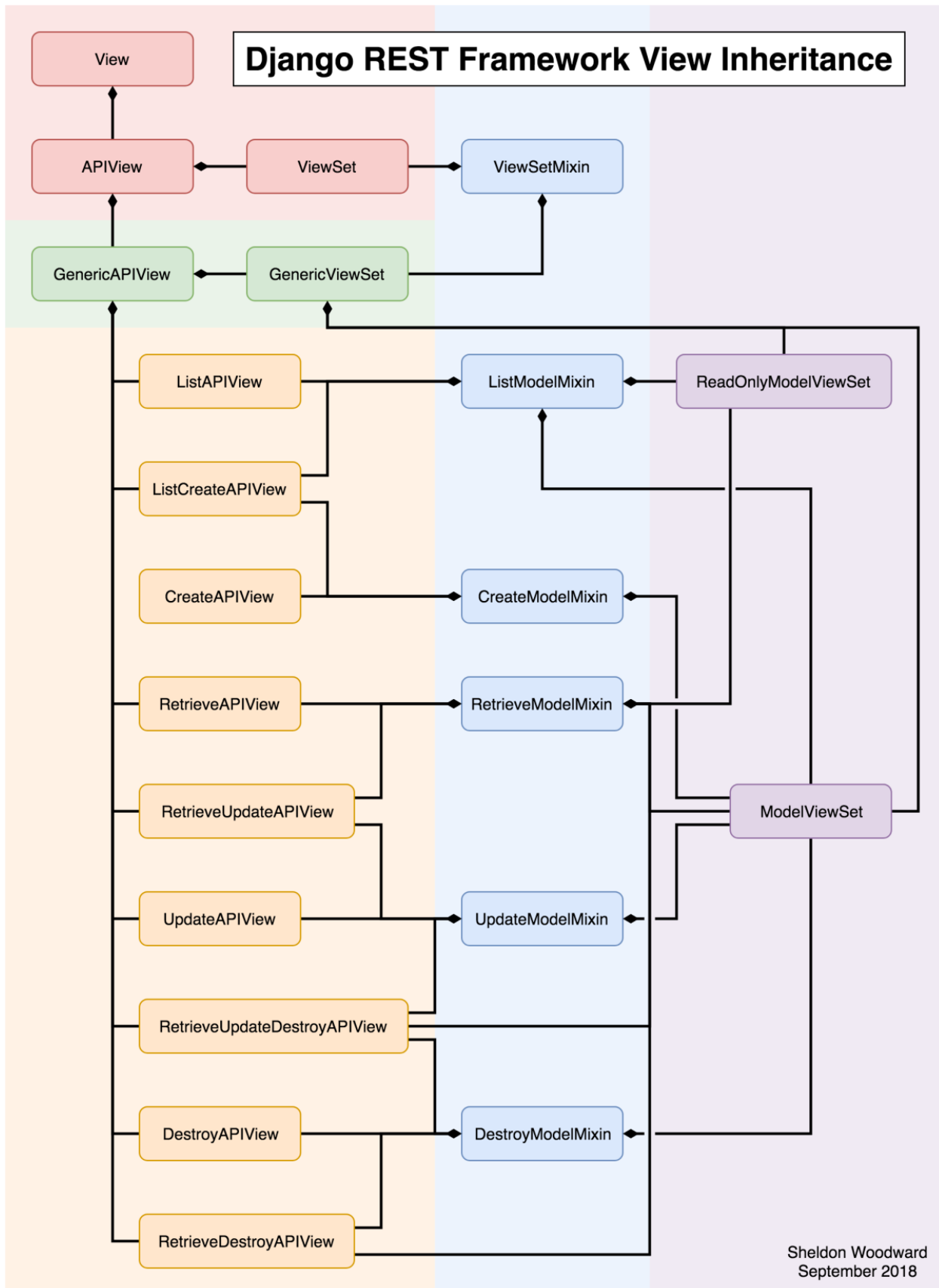
Em outros *frameworks*, são chamados de *Resources* ou *Controllers*.

ViewSets herdam e adicionam lógica às *Views* padrão do Django.

Suas responsabilidades são:

- Receber os dados da Requisição (formato JSON ou XML);
- Validar os dados de acordo com as regras definidas na modelagem e no *Serializer*;
- Desserializar a Requisição e instanciar objetos;
- Processar **regras de negócio** (aqui é onde implementamos a lógica dos nossos sistemas);
- Formular uma Resposta HTTP e retorná-la a quem chamou sua API.

Segue uma imagem que mostra o diagrama de herança das classes do DRF, que nos ajuda a entender melhor a estrutura interna do *framework*:



Na imagem:

- Na parte superior, temos a classe `View` padrão do Django.
- `APIView` e `ViewSet` são classes do DRF que herdam de `View` e que trazem algumas configurações específicas para transformá-las em APIs, como métodos `get()` para tratar requisições HTTP GET e `post()` para tratar requisições HTTP POST.
- Logo abaixo, temos a `GenericAPIView` - que é a classe base para views genéricas - e a `GenericViewSet` - que é a base para as `ViewSets` (a parte da direita em roxo na imagem).
- No meio, em azul, temos os `Mixins`. Eles são os blocos de código responsáveis por **realmente** implementar as ações desejadas.
- Em seguida temos as `Views` que disponibilizam as funcionalidades da nossa API, como se fossem blocos de Lego. Elas estendem dos `Mixins` para construir a funcionalidade desejada (seja listagem, seja deleção e etc).

Por exemplo: se você quiser criar uma API que disponibilize apenas listagem de uma determinada Entidade você poderia escolher a `ListAPIView`.

Agora, se você precisar construir uma API que disponibilize apenas as operações de criação e listagem, você poderia utilizar a `ListCreateAPIView`.

Agora se você precisar construir uma API “com tudo dentro” (isto é: criação, deleção, atualização e listagem), escolha a `ModelViewSet`! Perceba pela imagem que ela estende todos os `Mixins` disponíveis.

Para entender de vez:

- Os **Mixins** são como os componentes dos sanduíches do **Subway**
- As **Views** são como o Subway: você monta o seu, ingrediente à ingrediente...
- As **ViewSets** são como o McDonalds: seu sanduíche já vem montado!

Desculpa, mas eu não paro de pensar em comida =D

Percebe-se, portanto, que o DRF disponibiliza diversos tipos de `Views` e `ViewSets` que podem ser customizados de acordo com a necessidade do sistema.

Para isso, estude bem a documentação (*ou fique ligado no Blog da [Python Academy!](#)*)

Para facilitar a nossa vida, vamos utilizar a `ModelViewSet`!

No DRF, por convenção, implementamos as `Views/ViewSets` no arquivo `views.py` dentro do *app* em questão.

Esse arquivo já é criado quando utilizamos o comando `django-admin startapp api`, portanto não precisamos criá-lo.

Agora, vamos ver como é “difícil” criar um `ModelViewSet` (não se espantem com a complexidade - **contém ironia**):

```
from api.serializers import FundoImobiliarioSerializer
from rest_framework import viewsets, permissions
from api.models import FundoImobiliario

class FundoImobiliarioViewSet(viewsets.ModelViewSet):
    queryset = FundoImobiliario.objects.all()
    serializer_class = FundoImobiliarioSerializer
    permission_classes = [permissions.IsAuthenticated]
```

Uai, cadê o resto?!

Aí é que mora o **amor** e o **ódio** às *Class-Based-Views* (CBVs).

Amor: pois quem defende as CBVs afirma que elas aumentam a produtividade porque não temos de escrever dezenas de linhas de código.

Ódio: pois muito do funcionamento do *framework* fica escondido dos olhos dos desenvolvedores.

Todo o código para tratamento de Requisições HTTP, serialização e desserialização de objetos e formulação de Respostas HTTP está dentro das classes que herdamos direta ou indiretamente.

Em nossa classe `FundoImobiliarioViewSet` apenas precisamos declarar os seguintes parâmetros:

- `queryset`: configura o *queryset* base para ser utilizado pela API. Ele é utilizado na ação de listar, por exemplo.
- `serializer_class`: configura qual *Serializer* deverá ser usado para consumir dados que chegam à API e produzir dados que serão enviados como resposta.
- `permission_classes`: lista contendo as permissões necessárias para acessar o *endpoint* exposto por essa *ViewSet*. Nesse caso, irá permitir apenas o acesso a usuários autenticados (`permissions.IsAuthenticated`).

E assim, finalizamos o terceiro passo: **o de criação das *ViewSets* da nossa aplicação.**

E agora vamos à configuração das URLs!

ROUTERS

Os *Routers* auxiliam a geração das URLs em nossa aplicação.

Como o padrão arquitetural REST possui padrões bem definidos de estrutura de URLs, o DRF as gera automaticamente para nós, já seguindo o padrão correto.

Basta utilizarmos seus ***Routers***!

Para isso, primeiro crie o arquivo `urls.py` em `api/urls.py`.

Agora veja como é simples!

```
from rest_framework.routers import DefaultRouter
from api.views import FundoImobiliarioViewSet

app_name = 'api'

router = DefaultRouter(trailing_slash=False)
router.register(r'fundos', FundoImobiliarioViewSet)

urlpatterns = router.urls
```

Vamos entender:

- `app_name` é necessário para dar contexto às URLs geradas. Esse parâmetro especifica o *namespace* das *URLConfs* adicionadas.
- `DefaultRouter` é o *Router* que escolhemos para geração automática das URLs. O parâmetro `trailing_slash` especifica que não é necessário o uso de barras / no final da URL digitada por quem consome nossa API.
- O método `register` recebe dois parâmetros: o primeiro é o prefixo que será usado na URL (no nosso caso: `http://localhost:8000/fundos`) e o segundo é a *View* que irá responder as URLs com esse prefixo.
- Por último, temos o `urlpatterns` do Django, que utilizamos para expor as URLs desse app.

Agora precisamos adicionar as URLs específicas do nosso app `api` ao projeto.

Para isso, abra o arquivo `fundsfinder/urls.py` e adicione as seguintes linhas (*URLConf* do nosso app e a *URLConf* de autenticação do DRF):

```
from django.contrib import admin
from django.urls import path, include

urlpatterns = [
    path('api/v1/', include('api.urls', namespace='api')),
    path('api-auth/', include('rest_framework.urls', namespace='rest_framework')),
    path('admin/', admin.site.urls),
]
```

***Obs:** Como boa prática, use sempre o prefixo `api/v1/` para manter a compatibilidade caso precise evoluir sua API para V2 (`api/v2/`)!

Utilizando apenas essas linhas de código, podemos perceber a quantidade de endpoints que o DRF gerou automaticamente para nossa API:

URL	Método HTTP	Ação
/api/v1	GET	Raíz da API gerada automaticamente
/api/v1/fundos	GET	Listagem de todos os elementos
/api/v1/fundos	POST	Criação de novo elemento
/api/v1/fundos/{lookup}	GET	Recuperar elemento pelo ID
/api/v1/fundos/{lookup}	PUT	Atualização de elemento por ID
/api/v1/fundos/{lookup}	PATCH	Atualização parcial por ID (<i>partial update</i>)
/api/v1/fundos/{lookup}	DELETE	Deleção de elemento por ID

Aqui, **{lookup}** é o parâmetro utilizado pelo DRF para identificar unicamente um elemento.

Vamos supor que um Fundo tenha ID igual à **ef249e21**.

Podemos excluí-lo, por exemplo, enviando uma requisição HTTP **DELETE** para a URL:

```
http://localhost:8000/api/v1/fundos/ef249e21
```

Ou podemos criar um novo Fundo enviando uma requisição **POST** para a URL **http://localhost:8000/api/v1/fundos** e os valores dos campos no corpo da requisição, assim:

```
{
  "codigo": "XPLG11",
  "setor": "logistica",
```

```
"dividend_yield_medio_12m": "6.30",
"vacancia_financeira": "7.87",
"vacancia_fisica": "12.36",
"quantidade_ativos": 19
}
```

Dessa forma, nossa API retornaria um código HTTP **201 Created**, significando que um objeto foi criado, e a resposta seria:

```
{
  "id": "a4139c66-cf29-41b4-b73e-c7d203587df9",
  "codigo": "XPLG11",
  "setor": "logistica",
  "dividend_yield_medio_12m": "6.30",
  "vacancia_financeira": "7.87",
  "vacancia_fisica": "12.36",
  "quantidade_ativos": 19
}
```

Podemos testar nossa URL de diversas formas: através de código Python, através de um Frontend (Angular, React, Vue.js) ou através do [Postman](#), por exemplo.

E com isso, finalizamos **TUDO** (ou quase tudo)!

- Modelagem;
- *Serializers*;
- *ViewSets*;
- Routers.

E você pode estar se perguntando:

“E quando vou ver tudo finalmente funcionando?!”

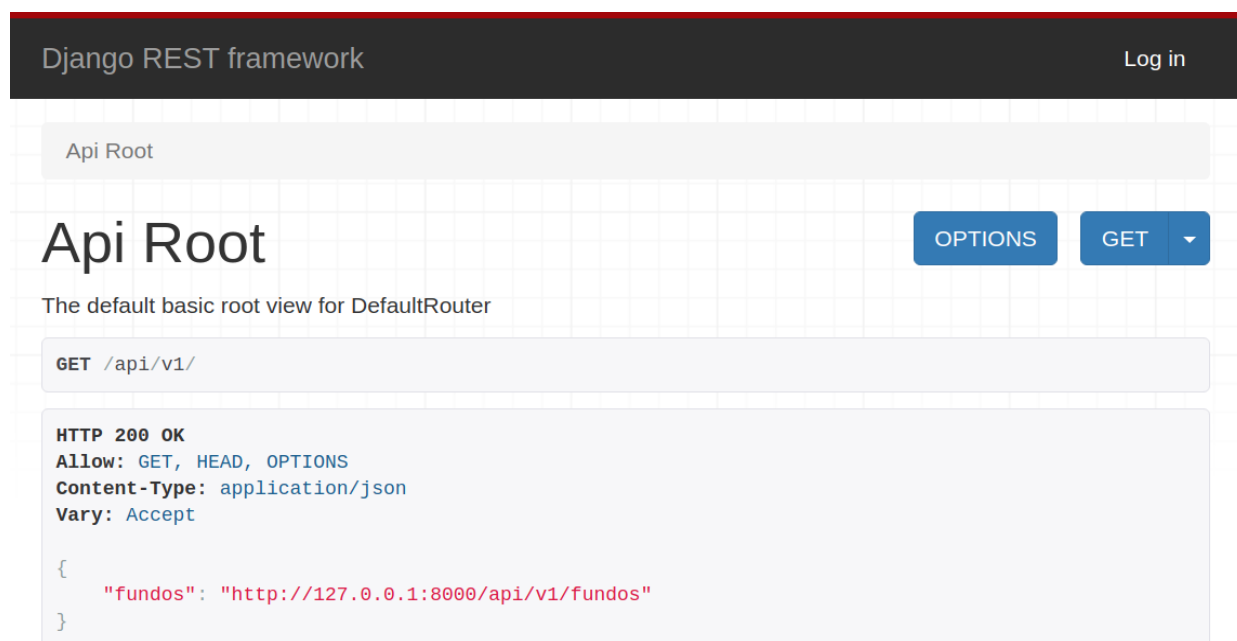
Para isso, vamos para a próxima seção, que é uma das funcionalidades mais impressionantes do DRF, que é a sua **Interface Navegável**.

INTERFACE NAVEGÁVEL

Com essa ferramenta - disponível por padrão no DRF - podemos testar nossa API e verificar seus valores de uma maneira visual muito simples de se utilizar.

Para acessá-la, navegue em seu browser para: `http://localhost:8000/api/v1/`.

Você deverá ver o seguinte:



Ao clicarmos em `http://127.0.0.1:8000/api/v1/fundos` deve ter aparecido a mensagem:

```
{
  "detail": "Authentication credentials were not provided."
}
```

Lembra da configuração `permission_classes` que usamos para configurar a `ViewSet`?

Ela definiu que apenas usuários autenticados (`permissions.isAuthenticated`) podem interagir com a API.

Então devemos clicar em “*Log in*” no canto superior direito e usar as credenciais usadas no comando `createsuperuser`, que executamos no início do Ebook.

Agora, olha que útil! A tela que deve aparecer:

Django REST framework

admin ▾

Api Root / Fundo Imobiliario List

Fundo Imobiliario List

OPTIONSGET ▾

GET /api/v1/fundos

HTTP 200 OK
Allow: GET, POST, HEAD, OPTIONS
Content-Type: application/json
Vary: Accept

[]

Raw dataHTML form

Id

Codigo

Setor

Hospital ▾

Dividend yield
medio 12m

Vacancia
financeira

Vacancia fisica

Quantidade ativos

POST

Aqui podemos brincar com a interface, inserindo dados e explorando à vontade.

Ao adicionar dados e - em seguida - se atualizar a página, será disparada uma requisição **HTTP GET** à API, retornando os dados que você acabou de cadastrar:

Api Root / Fundo Imobiliario List

Fundo Imobiliario List

OPTIONS

GET ▾

GET /api/v1/fundos

HTTP 200 OK

Allow: GET, POST, HEAD, OPTIONS

Content-Type: application/json

Vary: Accept

```
[
  {
    "id": "c41c2a9d-7aa4-4d07-b3b1-7a45df55ac79",
    "codigo": "MAXR11",
    "setor": "hibrido",
    "dividend_yield_medio_12m": "5.90",
    "vacancia_financeira": "0.00",
    "vacancia_fisica": "5.00",
    "quantidade_ativos": 10
  },
  {
    "id": "d7d9e0c5-1885-4298-ae12-3f8aa8eea755",
    "codigo": "HGLG11",
    "setor": "logistica",
    "dividend_yield_medio_12m": "4.80",
    "vacancia_financeira": "5.00",
    "vacancia_fisica": "17.00",
    "quantidade_ativos": 21
  },
  {
    "id": "ef249e21-43cf-47e4-9aac-0ed26af2d0ce",
    "codigo": "XPLG11",
    "setor": "logistica",
    "dividend_yield_medio_12m": "6.30",
    "vacancia_financeira": "7.87",
    "vacancia_fisica": "12.36",
    "quantidade_ativos": 19
  }
]
```

Incrível, não é mesmo?!

E ainda podemos adicionar funcionalidades de **Filtragem de dados**, **Busca Textual** e **Ordenação dos resultados** com pouquíssimas linhas de código e deixar a nossa API ainda mais completa.

Segue pro nosso próximo capítulo e vamos aprender **como!**

CAPÍTULO 8

FILTROS, BUSCA TEXTUAL E ORDENAÇÃO

O DRF nos possibilita adicionar as funcionalidades de ordenação de resultados, busca textual e filtragem de dados de maneira **extremamente simples!**

Para nos auxiliar, vamos começar instalando o pacote `django-filter`.

Na raíz do seu projeto, com o ambiente virtual instalado e ativado, execute:

```
pip install django-filter
```

Em seguida, adicione `django_filters` à variável `INSTALLED_APPS` do arquivo `settings.py` do projeto.

Feito isso, podemos começar!

FILTROS DE BUSCA

Suponha que seja necessário adicionar filtros para que os usuários possam consumir nossa API da seguinte forma:

```
http://localhost:8000/api/v1/fundos?setor=hibrido
```

Nesse exemplo, queremos que nossa API retorne apenas os Fundos Imobiliários cujo campo `setor=hibrido`.

Para utilizar filtros em nossa API, adicionamos os atributos `filter_backends` e `filterset_fields` às ViewSets:

- Em `filter_backends` colocamos o *Backend* que irá processar os filtros;
- Em `filterset_fields` adicionamos quais campos queremos disponibilizar para que seja feita a filtragem.

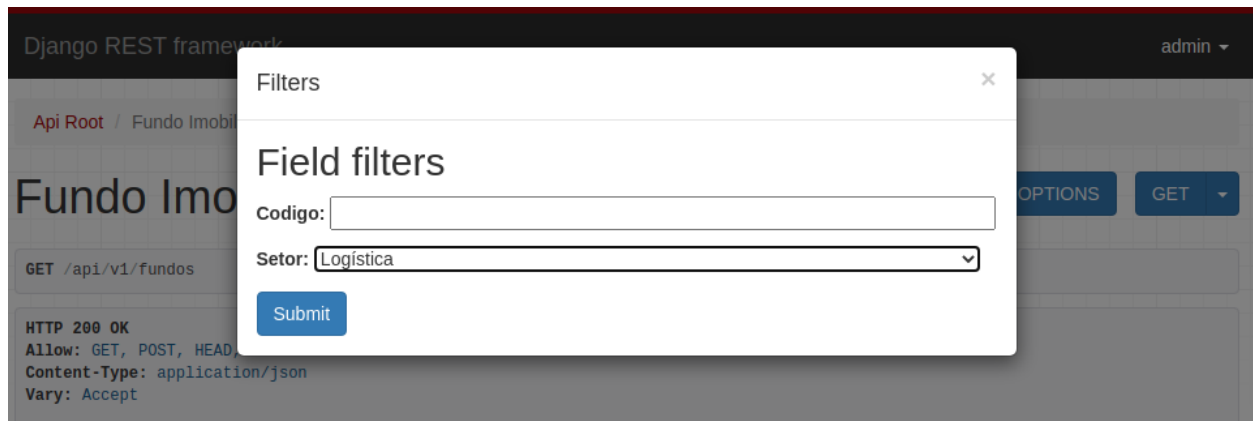
Com isso, nossa `ViewSet` fica assim:

```
from django_filters.rest_framework import DjangoFilterBackend
from api.serializers import FundoImobiliarioSerializer
from rest_framework import viewsets, permissions
from api.models import FundoImobiliario

class FundoImobiliarioViewSet(viewsets.ModelViewSet):
    queryset = FundoImobiliario.objects.all()
    serializer_class = FundoImobiliarioSerializer
    permission_classes = [permissions.IsAuthenticated]
    filter_backends = [DjangoFilterBackend]
    filterset_fields = ['codigo', 'setor']
```

Dessa forma podemos filtrar Fundos Imobiliários pelo `codigo` e `setor`!

Agora, abra a interface navegável e veja que o DRF adicionou um botão “*Filters*”: nele podemos testar os filtros!



Teste a URL `http://localhost:8000/api/v1/fundos?setor=hibrido` com o Postman e veja que show!

BUSCA TEXTUAL

A busca textual irá adicionar a funcionalidade de realizar buscas dentro de determinados valores de texto armazenados na base de dados.

Contudo, a busca só funciona para campos de texto, como `CharField` e `TextField`.

Para utilizar a busca textual, devemos promover duas alterações em nossa `ViewSet`:

- Novamente alterar o atributo `filter_backends`, **adicionando** o `Backend SearchFilter` que irá processar a busca; e
- Adicionar o atributo `search_fields`, contendo os campos que permitirão a busca.

Assim, nossa `ViewSet` fica da seguinte maneira:

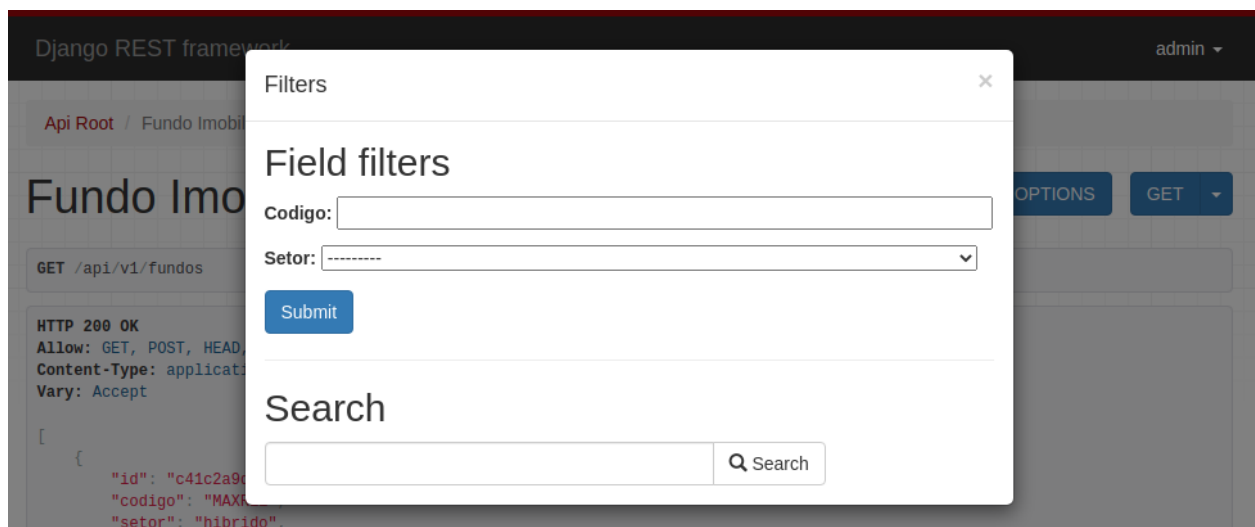
```

from django_filters.rest_framework import DjangoFilterBackend
from rest_framework.filters import SearchFilter
from api.serializers import FundoImobiliarioSerializer
from rest_framework import viewsets, permissions
from api.models import FundoImobiliario

class FundoImobiliarioViewSet(viewsets.ModelViewSet):
    queryset = FundoImobiliario.objects.all()
    serializer_class = FundoImobiliarioSerializer
    permission_classes = [permissions.IsAuthenticated]
    filter_backends = [DjangoFilterBackend, SearchFilter]
    filterset_fields = ['codigo', 'setor']
    search_fields = ['codigo', 'setor']

```

Agora abra a **interface navegável**, clique novamente em *Filters* e veja que foi adicionada o campo de busca **Search**!



Nele você pode realizar busca por valores contidos em **codigo** (ex: ALZR11, HGLG11, etc) e **setor** (ex: hibrido, hospital, residencial, etc).

DIZ SE NÃO É INCRÍVEL!!!

ORDENAÇÃO DE RESULTADOS

E por último, mas não menos importante: a **Ordenação**. Essa funcionalidade irá definir a ordem que os resultados devem ser apresentados.

A funcionalidade de Ordenação é configurada de forma semelhante aos Filtros e à Busca Textual. Para isso, é necessário:

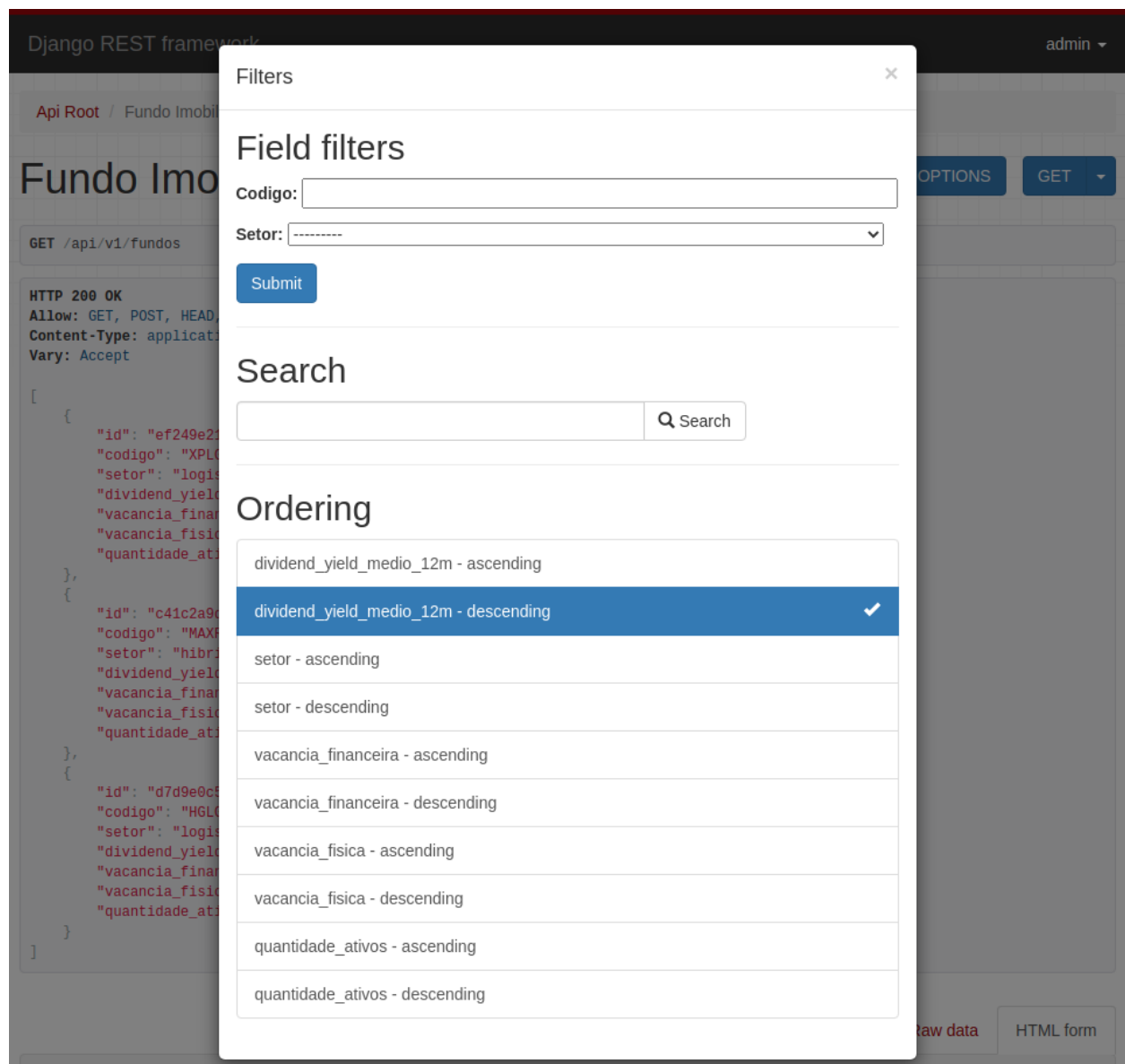
- Adicionar o *Backend OrderingFilter* ao atributo `filter_backends`;
- Adicionar o atributo `ordering_fields`, que são quais campos poderão ser ordenados;
- Adicionar opcionalmente o atributo `ordering` que configura a ordenação que será aplicada por padrão ao chamar endpoints que retornem mais de um resultado.

Portanto, a versão final da nossa `ViewSet` de Fundos Imobiliários deverá ficar assim:

```
from django_filters.rest_framework import DjangoFilterBackend
from rest_framework.filters import SearchFilter, OrderingFilter
from api.serializers import FundoImobiliarioSerializer
from rest_framework import viewsets, permissions
from api.models import FundoImobiliario

class FundoImobiliarioViewSet(viewsets.ModelViewSet):
    queryset = FundoImobiliario.objects.all()
    serializer_class = FundoImobiliarioSerializer
    permission_classes = [permissions.IsAuthenticated]
    filter_backends = [DjangoFilterBackend, SearchFilter, OrderingFilter]
    filterset_fields = ['codigo', 'setor']
    search_fields = ['codigo', 'setor']
    ordering = ['-dividend_yield_medio_12m']
    ordering_fields = [
        'dividend_yield_medio_12m',
        'setor',
        'vacancia_financeira',
        'vacancia_fisica',
        'quantidade_ativos']
```

Veja que a Interface Navegável já foi atualizada e adicionou a seção *Ordering*, com os campos configurados:



Demais né?!

E agora vamos ao **último** capítulo do nosso Ebook, e vamos falar sobre aspectos do Django Rest Framework que são controlados através de configurações específicas, como paginação, autenticação, permissões, formatação de dados...

CONFIGURAÇÕES ESPECÍFICAS

Além das funcionalidades que tratamos anteriormente, é possível adicionar configurações adicionais ao arquivo `settings.py` para que nossa API fique **ainda mais incrível!**

Por exemplo, se quisermos adicionar **Paginação** à nossa API, podemos fazer simplesmente isso:

```
REST_FRAMEWORK = {  
    'DEFAULT_PAGINATION_CLASS': 'rest_framework.pagination.PageNumberPagination',  
    'PAGE_SIZE': 10  
}
```

Perceba a diferença do resultado de uma Requisição GET, por exemplo, à `http://127.0.0.1:8000/api/v1/fundos`

Teremos este resultado:

```
[
  {
    "id": "ef249e21-43cf-47e4-9aac-0ed26af2d0ce",
    "codigo": "XPLG11",
    "setor": "logistica",
    "dividend_yield_medio_12m": "6.30",
    "vacancia_financeira": "7.87",
    "vacancia_fisica": "12.36",
    "quantidade_ativos": 19
  },
  {
    "id": "c41c2a9d-7aa4-4d07-b3b1-7a45df55ac79",
    "codigo": "MAXR11",
    "setor": "hibrido",
    "dividend_yield_medio_12m": "5.90",
    "vacancia_financeira": "0.00",
    "vacancia_fisica": "5.00",
    "quantidade_ativos": 10
  },
  {
    "id": "d7d9e0c5-1885-4298-ae12-3f8aa8eea755",
    "codigo": "HGLG11",
    "setor": "logistica",
    "dividend_yield_medio_12m": "4.80",
    "vacancia_financeira": "5.00",
    "vacancia_fisica": "17.00",
    "quantidade_ativos": 21
  }
]
```

E após adicionar a configuração de **Paginação**:

```
{
  "count": 3,
  "next": null,
  "previous": null,
  "results": [
    {
```

```

        "id": "ef249e21-43cf-47e4-9aac-0ed26af2d0ce",
        "codigo": "XPLG11",
        "setor": "logistica",
        "dividend_yield_medio_12m": "6.30",
        "vacancia_financeira": "7.87",
        "vacancia_fisica": "12.36",
        "quantidade_ativos": 19
    },
    {
        "id": "c41c2a9d-7aa4-4d07-b3b1-7a45df55ac79",
        "codigo": "MAXR11",
        "setor": "hibrido",
        "dividend_yield_medio_12m": "5.90",
        "vacancia_financeira": "0.00",
        "vacancia_fisica": "5.00",
        "quantidade_ativos": 10
    },
    {
        "id": "d7d9e0c5-1885-4298-ae12-3f8aa8eea755",
        "codigo": "HGLG11",
        "setor": "logistica",
        "dividend_yield_medio_12m": "4.80",
        "vacancia_financeira": "5.00",
        "vacancia_fisica": "17.00",
        "quantidade_ativos": 21
    }
]
}

```

Note que foram adicionados campos que servem para a aplicação chamadora se localizar nos resultados:

- **count**: Contém a quantidade de resultados retornados.
- **next**: Contém a próxima página de resultados.
- **previous**: Contém a página anterior de resultados.
- **results**: A página atual de resultados.

Existem diversas outras configurações muito úteis! Trago aqui algumas:

- `DEFAULT_AUTHENTICATION_CLASSES` para configurar o método padrão de autenticação utilizado para consumir a API:

```
REST_FRAMEWORK = {  
    ...  
    DEFAULT_AUTHENTICATION_CLASSES: [  
        'rest_framework.authentication.SessionAuthentication',  
        'rest_framework.authentication.BasicAuthentication'  
    ]  
    ...  
}
```

- `DEFAULT_PERMISSION_CLASSES` para configurar o conjunto padrão de permissões necessárias para acessar a API (à nível global).

```
REST_FRAMEWORK = {  
    ...  
    DEFAULT_PERMISSION_CLASSES: ['rest_framework.permissions.AllowAny']  
    ...  
}
```

Obs: Também é possível definir essa configuração por View, utilizando o atributo `permissions_classes` (que utilizamos na nossa `FundoImobiliarioViewSet`).

- `DATE_INPUT_FORMATS` para configurar formatos de datas aceitos pela API:

```
REST_FRAMEWORK = {  
    ...  
    'DATE_INPUT_FORMATS': ['%d/%m/%Y', '%Y-%m-%d', '%d-%m-%y', '%d-%m-%Y']  
    ...  
}
```

A configuração acima fará a API permitir os seguintes formatos de data, por exemplo: '25/10/2006', '2006-10-25', '25-10-2006'.

Veja mais configurações [acessando aqui a Documentação](#).

FINALIZAÇÃO

UM ATÉ BREVE...

Chegamos ao fim do nosso ebook!

Mas, como você sabe, o Django e o DRF estão em constante evolução. Por isso, é bom você se manter atualizado nas novidades lendo, pesquisando e acompanhando o mundo do Django.

E deixo aqui novamente o convite para você conhecer a **Jornada Python**: lá você vai aprender do básico ao avançado de Python e Django, com projetos completos, dicas de carreira, certificado, suporte à dúvidas, além de dar continuidade aos seus estudos de Python, com conteúdos em vídeo, Quizzes, projetos e muito mais!

Clique na imagem abaixo agora mesmo para conhecer a **Jornada Python**!

