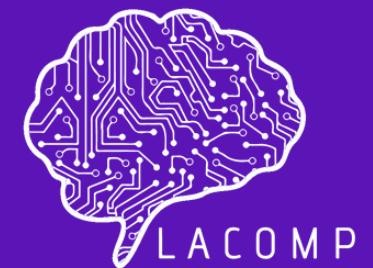


CRIAÇÃO E MANUTENÇÃO DE APIs REST





O QUE É UM
FRAMEWORK?

O QUE É UM FRAMEWORK?

- É uma estrutura pré-definida para resolver um determinado problema.



Vamos usar a analogia dos blocos de Lego para entender o conceito de um framework.



O QUE É DJANGO?



O QUE É DJANGO?

- Django é um framework de desenvolvimento ágil para web, escrito em Python.
- Criado originalmente como sistema para gerenciar um site jornalístico na cidade de Lawrence, no Kansas, tornou-se um projeto de código aberto e foi publicado sob a licença BSD em 2005.

PRINCIPAIS CARACTERÍSTICAS

- Aproveitamento de código (DRY)
- Mapeamento Objeto-Relacional (ORM)
- Interface Administrativa
- Formulários



O QUE SÃO
APIS REST?

O QUE SÃO APIs REST?

- As APIs REST (Representational State Transfer) são um estilo de arquitetura de software que define um conjunto de restrições e princípios para o design de serviços web.
- Recursos (Resources), Verbos HTTP, Comunicação (Request/Response).



O QUE É **DJANGO REST FRAMEWORK?**

O QUE É DJANGO REST FRAMEWORK?

- Django REST Framework (DRF) é um framework poderoso e flexível para construir APIs da Web, usando a arquitetura REST, escrito em Python e usado em conjunto com o Django.



INSTALAÇÃO DO DJANGO

- Em primeiro lugar, devemos criar uma pasta onde estará o nosso projeto com Django.
- Nessa pasta, é recomendável a criação de um ambiente virtual (`virtualenv`) de Python. A criação do ambiente virtual é totalmente opcional, caso não queira criar, pule para o próximo slide!

`python -m venv env`

`env\Scripts\activate`

`source env/bin/activate`

(Windows)

(MacOS e Linux)

INSTALAÇÃO DO DJANGO

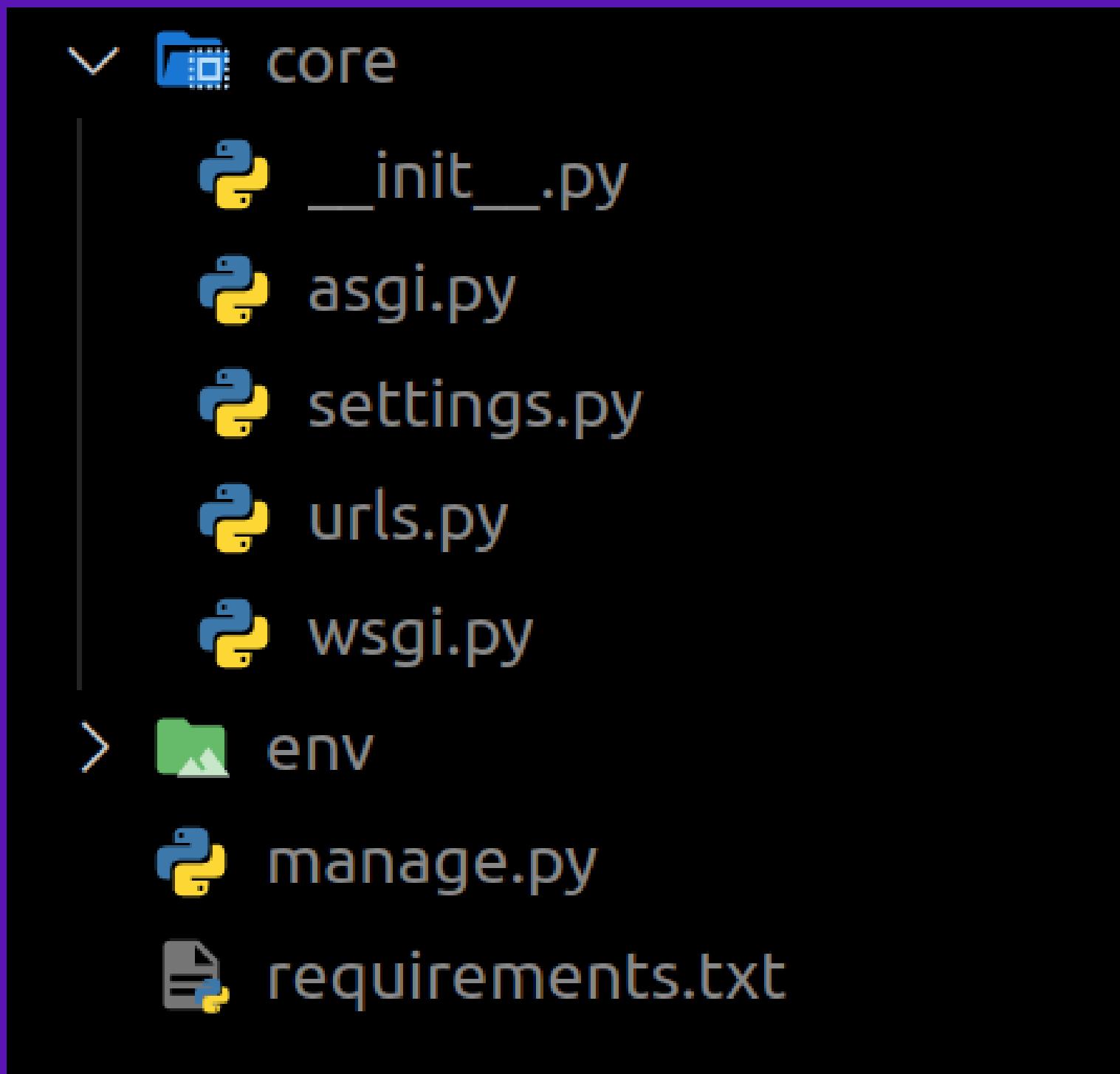
- Com a virtualenv criada e ativa, podemos realizar instalação do Django!
`pip install Django`
- O passo anterior vai instalar, além do Django, módulos necessários para rodar o projeto.
`pip freeze > requirements.txt`
- Para instalar a lista de pacotes do requirements.txt, basta usar:
`pip install -r requirements.txt`



CRIAÇÃO DO PROJETO DJANGO

- Após a instalação do Django e outras dependências, podemos finalmente criar nosso primeiro projeto!
`django-admin startproject core .`
- O ponto ao final do comando é importante porque indica para o script instalar o Django na pasta atual. Caso não o coloque, o script criará uma nova pasta, a partir da pasta atual, e instalará o Django nela.

ESTRUTURA DO PROJETO



MANAGE.PY

- Essa execução é feita da seguinte forma:

`python manage.py comando`

- Alguns dos principais comandos que podemos usar:
 - `runserver`: iniciar um servidor local da aplicação Django.
 - `makemigrations`: cria as migrações do projeto ou de uma app.
 - `migrate`: aplica as migrações do projeto ou de uma app.
 - `showmigrations`: exibe as migrações do projeto ou de uma app.
 - `shell`: inicia o console do Python no contexto do projeto.
 - `createsuperuser`: cria um super usuário para o projeto, com acesso total ao site da administração.
 - `startapp`: cria uma app para o projeto.

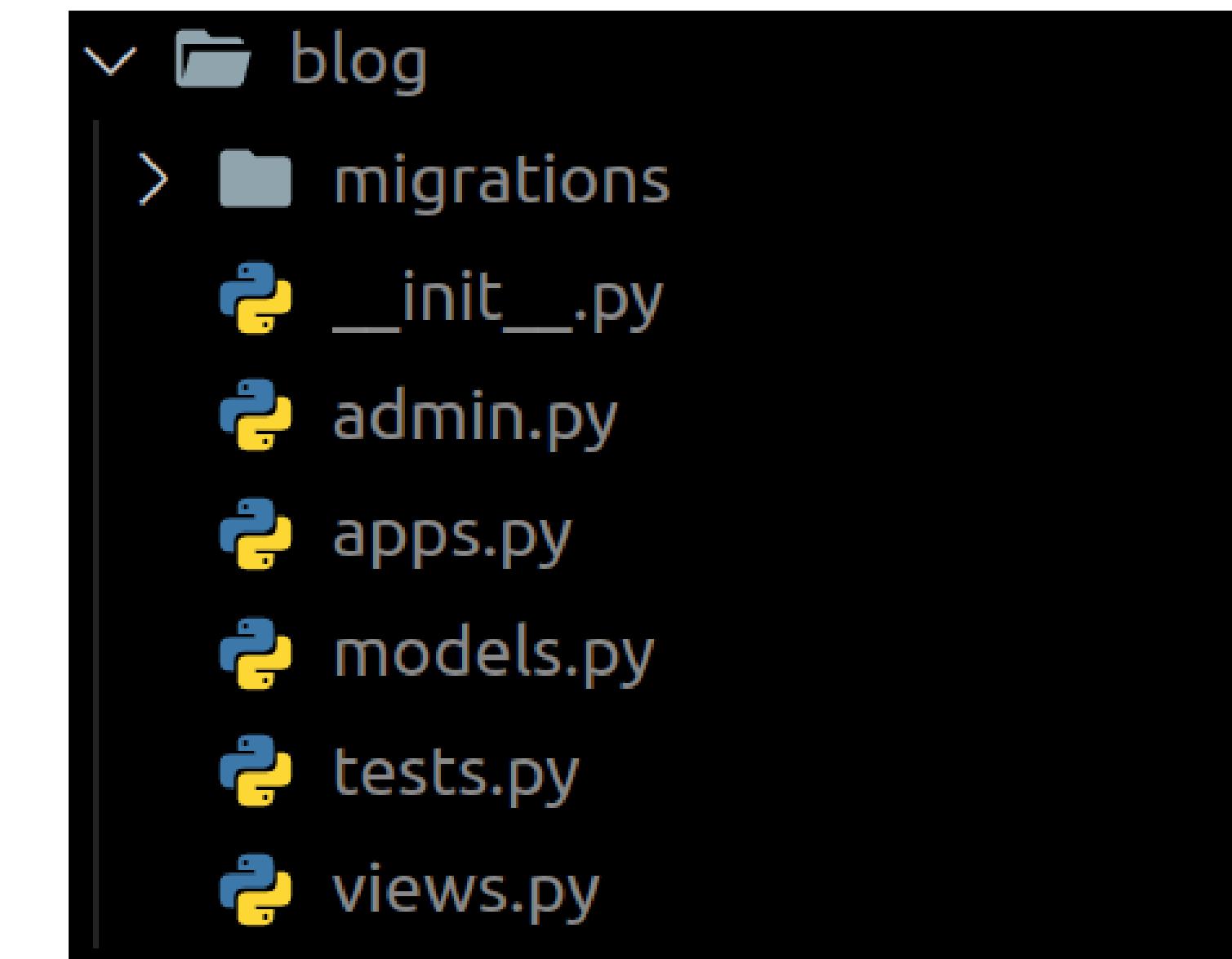
CRIAÇÃO DE APP

- Com o projeto criado, precisamos criar as aplicações que teremos nele. Uma aplicação em Django é um pacote Python que pode ter seus próprios models, urls, formulários e afins. É necessário para separar módulos do sistema que possuem lógicas distintas. Para criar a app, usamos o comando:

```
python manage.py startapp blog
```

CRIAÇÃO DE APP

- Após a criação da app, é criada uma pasta com o referido nome e que deverá ter como resultado um conjunto de pastas e arquivos semelhante com esse:



CONFIGURAÇÃO DO PROJETO

- Agora podemos mexer no arquivo de configuração do projeto, o `settings.py`.
- Aqui podemos definir diversas coisas, como o idioma do projeto, o fuso-horário, adicionar novas dependências, registro de apps e muito mais!



CONFIGURAÇÃO DO PROJETO

- `SECRET_KEY`, como o nome já indica, é a chave secreta do projeto, usada para fornecer uma assinatura criptográfica.
- `DEBUG` é uma flag que indica se o servidor está rodando em modo de depuração ou não.
- `ALLOWED_HOSTS` são os locais com permissão de acessar recursos do projeto.
- `INSTALLED_APPS` é a lista das apps instaladas no projeto, incluindo nossas apps criadas e recursos de terceiros.

CONFIGURAÇÃO DO PROJETO

- **DATABASES** é a lista dos bancos de dados que podem ser usados pelo projeto.
- **LANGUAGE_CODE** é o idioma que o projeto será exibido. Podemos substituir o valor por 'pt-br' para nosso projeto ser exibido em português brasileiro.
- **TIME_ZONE** é o fuso-horário que será adotado pelo projeto. Podemos substituir o valor por 'America/Maceio' para usarmos o fuso-horário da nossa cidade.



MODELS

MODELS

- Os models são a representação das tabelas do banco de dados em Django.
- Para ser reconhecido como um model, uma classe precisa herdar `models.Model` do Django.
- Podemos criar as tabelas diretamente no Django e 'exportá-las' para o banco de dados e vice-versa.
- Assim como no SQL, precisamos definir quais são os atributos de um determinado model (tabela), bem como seus tipos de dados.
- Os atributos da classe também são herdados do pacote `models`.



TIPOS DE ATRIBUTOS COMUNS

- **AutoField:** Inteiro auto incremental, usado em chaves primárias.
- **CharField:** Semelhante ao VARCHAR do SQL, usado em campos de texto curto.
- **TextField:** Campo de texto longo (Semelhante ao TEXT do SQL).
- **IntegerField:** Usado para armazenar valores inteiros.
- **FloatField:** Usado para armazenar números reais.
- **BooleanField:** Usado para armazenar valores booleanos.
- **DateField:** Armazena uma data.
- **EmailField:** Campo de texto com verificação se é um e-mail válido.
- **URLField:** Campo de texto com verificação se é um link válido.

PROPRIEDADE DOS ATRIBUTOS

- Uma propriedade define o comportamento de um determinado atributo e/ou como será seu armazenamento no banco de dados.
- Podemos definir nenhuma ou infinitas propriedades para um atributo, assim seja necessário.
- Caso não especifique as propriedades de um atributo, ele receberá as propriedades básicas daquele tipo.



PROPRIEDADES MAIS COMUNS

- `primary_key`: Valor booleano que define que aquele atributo é a chave primária da tabela.
- `null`: Valor booleano que define se o atributo pode ser nulo.
- `blank`: Valor booleano que define se o atributo pode ser cadastrado vazio.

PROPRIEDADES MAIS COMUNS

- `max_length`: Define o limite de caracteres de um campo de texto.
- `default`: Define o valor padrão para um determinado atributo, que será usado no cadastro, caso não seja informado um valor.
- `choices`: Define as opções possíveis para um campo de texto.



PROPRIEDADES MAIS COMUNS

- `unique`: Valor booleano que define se um atributo deve ter valor único entre as instâncias.
- `auto_now_add`: Valor booleano que define que uma data será instanciada automaticamente no momento do cadastro.
- `on_delete`: Usado em relacionamentos, define o comportamento de uma chave caso a instância referenciada seja excluída. Os valores mais comuns são: `models.CASCADE`, `models.SET_NULL`.

CRIAÇÃO DE MODELS

```
from django.db import models

class Postagem(models.Model):
    id_postagem = models.AutoField(primary_key=True)
    titulo = models.CharField(max_length=200)
    texto = models.TextField(max_length=10000)
    data = models.DateTimeField(auto_now=True)
```

CRIAÇÃO DE MODELS

```
class Comentario(models.Model):
    id_comentario = models.AutoField(primary_key=True)
    comentario = models.TextField(max_length=1000)
    data = models.DateTimeField(auto_now=True)
    # postagem = como fazer a Ligação com a postagem?
```

RELACIONAMENTO (1, 1)

- Um relacionamento (1, 1) é aquele na qual uma instância de uma determinada tabela só pode se relacionar com uma instância da outra tabela participante e vice-versa.
- Em Django, usamos o atributo do tipo `OneToOneField` para representar esse relacionamento:
`chave = models.OneToOneField(Tabela, on_delete=models.CASCADE)`



RELACIONAMENTO (1, N)

- Um relacionamento (1, N) é aquele na qual uma instância de uma determinada tabela pode se relacionar com várias instâncias da outra tabela participante, porém uma instância da segunda tabela só pode se relacionar com uma da primeira.
- Em Django, usamos o atributo do tipo `ForeignKey` para representar esse relacionamento:
`chave = models.ForeignKey(Tabela, on_delete=models.CASCADE)`

RELACIONAMENTO (N, N)

- Um relacionamento (N, N) é aquele na qual uma instância de uma determinada tabela pode se relacionar com várias instâncias da outra tabela participante e vice-versa.
- Em Django, usamos o atributo do tipo `ManyToManyField` para representar esse relacionamento:
`chave = models.ManyToManyField(Tabela)`

CRIAÇÃO DE MODELS

```
class Comentario(models.Model):
    id_comentario = models.AutoField(primary_key=True)
    comentario = models.TextField(max_length=1000)
    data = models.DateTimeField(auto_now_add=True)
    postagem = models.ForeignKey(Postagem, on_delete=models.CASCADE)
```

META OPTIONS DE UM MODEL

- Os Meta options de um model são um conjunto de propriedades que aquele model terá no Django e/ou banco de dados.
- Caso não especifique as opções de um model, ele receberá as propriedades básicas.



OPTIONS MAIS COMUNS

- `ordering`: define a ordenação das instâncias desse model.
- `db_table`: define a tabela no banco de dados que armazenará as instâncias desse model.
- `unique_together`: define uma combinação de campos que só pode existir apenas uma vez entre as instâncias desse model.



AS TABELAS
EXISTEM NO
BANCO DE DADOS?



MIGRAÇÕES NO DJANGO

- Migrações se tratam de scripts em Python que contém as alterações estruturais realizadas nos models.
- As alterações em models só refletirão no banco após a aplicação de suas respectivas migrations.
- Para criar as migrations, usamos o comando:
`python manage.py makemigrations app`
- Para aplicar as migrations, usamos o comando:
`python manage.py migrate app`



EXERCÍCIO DE IMPLEMENTAÇÃO!



DJANGO REST FRAMEWORK

INSTALAÇÃO DO DRF

- O Django REST Framework, assim como o Django, é instalado por meio do instalador de pacotes Pip!
`pip install djangorestframework`
- Após isso, seguimos novamente para o arquivo `settings.py` do projeto para proceder com a instalação.



INSTALAÇÃO DO DRF

- No `settings.py`, adicionamos ‘`rest_framework`’ à lista de `INSTALLED_APPS`.
- Ainda no arquivo `settings.py`, podemos adicionar um dicionário nomeado `REST_FRAMEWORK` que contará com as configurações globais do DRF no projeto.
- Caso não especifique as opções do DRF, ele receberá as configurações padrões.

INSTALAÇÃO DO DRF

- Caso deseje usar a autenticação do DRF, no arquivo `urls.py` localizado na raiz do projeto, adicionamos a seguinte linha à lista de urlpatterns:
`path('api-auth/', include('rest_framework.urls'))`
- Talvez seja necessário importar o `include`. Se for o caso, basta adicionar junto ao `import` de `path`.
- Não se preocupe, ainda veremos mais a fundo sobre a configuração de urls no Django!



SERIALIZERS

- Serialização é o processo de ‘tradução’ dos objetos em um formato que possa ser armazenado e reconstruído posteriormente no mesmo ou em outro ambiente computacional.
- No DRF, esse processo transforma uma instância de um model num objeto JSON, que pode ser enviado como resposta em requisições HTTP.

SERIALIZERS

- Por padrão, o arquivo `serializers.py` não existe em nossas apps, então é necessário criar.
- Nesse arquivo, importamos `serializers`, do pacote `rest_framework`, conforme indicado abaixo:
`from rest_framework import serializers`
- Também precisamos importar os models que foram criados, para que possamos serializá-los.



SERIALIZERS

- Agora precisamos criar as classes que irão serializar cada um dos models.
- Para ser reconhecido como um Serializer, precisamos herdar `serializers.ModelSerializer`.
- Na classe Meta do Serializer, indicamos qual model e quais os campos que serão reconhecidos.

SERIALIZERS

```
from rest_framework import serializers
from .models import *

class PostagemSerializer(serializers.ModelSerializer):
    class Meta:
        model = Postagem
        fields = '__all__'
```



VIEWS

- Views são classes que contém os métodos de CRUD de um determinado model e que recebem uma URL para que possam ser acessadas.
- As views existem tanto no Django quanto no DRF, porém com objetivos diferentes em cada contexto.

VIEWS

- Usaremos o arquivo `views.py`, já presente no projeto, para definir nossas Views.
- Nesse arquivo, importamos `viewsets`, do pacote `rest_framework`, conforme indicado abaixo:
`from rest_framework import viewsets`
- Também precisamos importar os `models` e `serializers` que foram criados, para que possamos usá-los nas Views.



VIEWS

- Agora precisamos criar as classes que possuirão os métodos de CRUD para cada um dos models.
- Para ser reconhecido como uma View, precisamos herdar as características de `viewsets.ModelViewSet` ou `viewsets.ViewSet`.

VIEWSET VS MODELVIEWSET

- Uma ViewSet fornece um conjunto de métodos para criar, listar, detalhar, atualizar e excluir objetos, porém você precisa implementar cada um desses métodos.
- Um ModelViewSet é uma subclasse de ViewSet que também fornece um conjunto de métodos para criar, listar, detalhar, atualizar e excluir objetos, porém esses métodos estão implícitos, sem necessidade da implementação deles, caso não seja necessário.



VIEWSET

- Caso `ViewSet` seja o escolhido, precisamos definir os seguintes métodos para o CRUD:
 - `create`: cadastro de novas instâncias;
 - `list`: lista de instâncias;
 - `retrieve`: detalhamento de uma instância;
 - `update`: atualização completa de uma instância;
 - `partial_update`: atualização parcial de uma instância;
 - `destroy`: exclusão de uma instância.

MODELVIEWSET

```
from rest_framework import viewsets  
from .models import *  
from .serializers import *  
  
class PostagemView(viewsets.ModelViewSet):  
    queryset = Postagem.objects.all()  
    serializer_class = PostagemSerializer
```



**POSSO TER MÉTODOS
ALÉM DO CRUD?**

ACTIONS

- Presente tanto na `ViewSet` quanto no `ModelViewSet`, um `action` são métodos personalizados que podem ser criados no contexto da View.
- Precisa aceitar pelo menos um dos verbos de requisições HTTP.
- Usa o decorator `@action` para indicar que um determinado método é um action.



ACTIONS

- É necessário importar o decorator `action` do pacote `rest_framework.decorators`, conforme indicado abaixo:
`from rest_framework.decorators import action`
- Também é necessário importar `Response` do pacote `rest_framework.response`, conforme indicado abaixo:
`from rest_framework.response import Response`
- Um `action` precisa retornar uma `Response`.

ACTIONS

```
from rest_framework import viewsets
from rest_framework.decorators import action
from rest_framework.response import Response

from .models import *
from .serializers import *

class PostagemView(viewsets.ModelViewSet):
    queryset = Postagem.objects.all()
    serializer_class = PostagemSerializer

    @action(detail=True, methods=['get'])
    def data_postagem(self, request, pk=None):
        postagem = Postagem.objects.get(id_postagem=pk)
        return Response(postagem.data)
```



URLS

- Agora, precisamos definir as URLs do nosso projeto.
- Podemos definir diretamente no arquivo `urls.py` na pasta de configuração do projeto ou definir em cada app.
- Usaremos a segunda opção!

URLS

- Por padrão, uma app não possui o arquivo `urls.py`, então é necessário criar.
- Nesse arquivo, importamos `routers`, do pacote `rest_framework`, conforme indicado abaixo:
`from rest_framework import routers`
- Também precisamos importar as Views que foram criadas.



URLS

- Agora precisamos criar um objeto `router`, que será responsável por criar o roteamento das urls.
`routerBlog = routers.DefaultRouter()`
- Agora, basta definir a rota de cada View que foi criada!
- Na classe Meta do Serializer, indicamos qual model e quais os campos que serão reconhecidos.

URLS

```
from rest_framework import routers
from .views import *

routerBlog = routers.DefaultRouter()
routerBlog.register(r'postagem', PostagemView)
```



URLS

- Agora, seguimos para o `urls.py` na pasta de configuração do projeto.
- Nesse arquivo, precisamos importar os `routers` definidos em cada app e adicioná-los à lista `urlpatterns`.

URLS

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

from blog.urls import routerBlog

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



**EXERCÍCIO DE
IMPLEMENTAÇÃO!**



ALGUMA
DÚVIDA?



MUITO
OBRIGADO!