



GUIA COMPLETO DE DJANGO 2025: DO BÁSICO AO AVANÇADO

Guia definitivo Django 5.1 em 2025: instalação, MTV, models, views, templates, forms, admin, REST API, PostgreSQL, MySQL, middlewares, async. Tutorial completo para iniciantes e avançados.

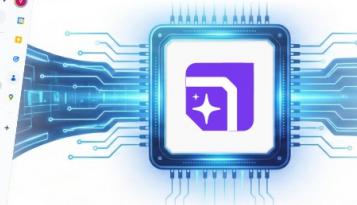
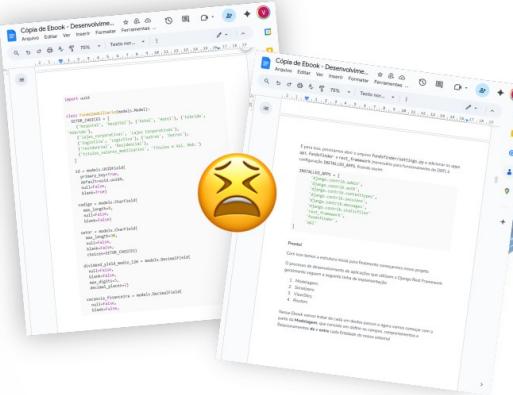
Gere ebooks como este com



Ebookr

em <https://ebookr.ai>

Crie ebooks profissionais incríveis em minutos com IA



Chega de formatar texto no Google Docs, Word ou ferramentas que só te fazem perder tempo...

E deixe que nossa IA faça o trabalho pesado!



Capas gerados por IA



Infográficos feitos por IA



Edite em Markdown em Tempo Real



Adicione Banners Promocionais

TESTE AGORA



PRIMEIRO CAPÍTULO 100% GRÁTIS

Salve salve Pythonista!

Django se consolidou como um dos frameworks web mais poderosos e completos do ecossistema Python.

Em 2025, com a versão 5.1 LTS (Long Term Support), o Django continua evoluindo, adicionando suporte robusto para **operações assíncronas, WebSockets, APIs RESTful** e mantendo sua filosofia de “baterias incluídas”.

Este é o **guião definitivo de Django em 2025**: um artigo pilar que reúne tudo o que você precisa saber para dominar o framework, desde a instalação até o desenvolvimento de aplicações web complexas e APIs profissionais.

Se você está começando agora ou quer aprofundar seus conhecimentos, este guia é para você! 

O que é Django e por que usar em 2025?

Django é um framework web de alto nível escrito em Python que incentiva o desenvolvimento rápido e o design limpo e pragmático.

Criado em 2005, Django amadureceu ao longo de duas décadas e hoje é a escolha de empresas como **Instagram, Spotify, Mozilla, Pinterest e NASA**.

Por que Django continua relevante em 2025?

1. Baterias Incluídas Django vem com tudo que você precisa out-of-the-box:

- ORM poderoso para banco de dados
- Sistema de autenticação completo

- Painel administrativo automático
- Proteção contra vulnerabilidades (SQL Injection, XSS, CSRF)
- Sistema de templates robusto
- Gerenciamento de arquivos estáticos e media

2. Suporte Async Completo (Django 5.1)

- Views assíncronas (`async def`)
- Middlewares assíncronos
- Suporte ASGI nativo
- WebSockets integrados
- Melhor performance em operações I/O

3. Ecossistema Rico

- Django REST Framework para APIs
- Django Channels para real-time
- Celery para tarefas assíncronas
- Integração perfeita com PostgreSQL, MySQL, Oracle

4. Comunidade Ativa

- LTS (Long Term Support) até 2026
- Atualizações regulares de segurança
- Documentação excepcional
- Comunidade global gigantesca

5. Produtividade O lema “Don’t Repeat Yourself” (DRY) do Django significa que você escreve menos código e entrega mais rápido.

Instalação e Configuração do Django 5.1

Pré-requisitos

Antes de começar, certifique-se de ter:

- **Python 3.10+** instalado
- **pip** (gerenciador de pacotes Python)
- **Ambiente virtual** configurado

Criando um Ambiente Virtual

É fundamental usar ambientes virtuais para isolar dependências do projeto:

```
# Criar ambiente virtual
python -m venv venv

# Ativar no Linux/Mac
source venv/bin/activate

# Ativar no Windows
venv\Scripts\activate
```

Para entender mais sobre ambientes virtuais, [leia nosso guia completo sobre Virtualenv](#).

Instalando Django 5.1

Com o ambiente virtual ativado, instale a versão mais recente:

```
pip install django

# Verificar versão instalada
python -m django --version
```

Versão atual (2025): Django 5.1 LTS com suporte estendido até abril de 2026.

Criando Seu Primeiro Projeto

Django fornece o utilitário `django-admin` para iniciar projetos:

```
django-admin startproject meu_projeto
cd meu_projeto
```

Isso cria a seguinte estrutura:

```
meu_projeto/
    manage.py
    meu_projeto/
        __init__.py
        settings.py
        urls.py
        asgi.py
        wsgi.py
```

Arquivos importantes:

- `manage.py` - Interface de linha de comando do projeto
- `settings.py` - Configurações do projeto

- `urls.py` - Roteamento de URLs
- `asgi.py` - Configuração ASGI (async)
- `wsgi.py` - Configuração WSGI (tradicional)

Django 5.1: ASGI é o padrão recomendado para novos projetos, oferecendo suporte completo a operações assíncronas.

Para um tutorial passo a passo, confira: [Seu primeiro projeto Django em 15 minutos.](#)

Executando o Servidor de Desenvolvimento

```
python manage.py runserver
```

Acesse `http://127.0.0.1:8000/` e veja a página de boas-vindas do Django!



Arquitetura MTV: Model-Template-View

Django segue o padrão **MTV** (Model-Template-View), uma variação do MVC (Model-View-Controller):

Model (Modelo)

Representa a **camada de dados** - a estrutura do banco de dados e a lógica de negócios.

Template (Template)

Representa a **camada de apresentação** - como os dados são exibidos ao usuário (HTML).

View (Visão)

Representa a **camada de controle** - processa requisições HTTP e retorna respostas.

Fluxo de uma requisição Django:

1. Usuário acessa URL → urls.py
2. URL mapeia para View → views.py
3. View consulta Model → models.py
4. Model retorna dados do banco de dados
5. View processa dados e renderiza Template → templates/*.html
6. Template é retornado como resposta HTTP ao usuário

Para entender profundamente a arquitetura, leia: [Django: Introdução ao framework](#).



Novidade: quer criar ebooks profissionais usando IA, com capa gerada automaticamente, infográficos e exportação em PDF? Crie sobre qualquer tema no [Ebookr.ai!](#)



Crie Ebooks profissionais incríveis em minutos com IA



Chega de formatar texto no Google Docs, Word ou ferramentas que só te fazem perder tempo...

... e deixe que nossa IA faça o trabalho pesado!

TESTE AGORA! PRIMEIRO CAPÍTULO 100% GRÁTIS

Capas gerados por IA

Adicione banners Promocionais

Edite em Markdown em Tempo Real

Infográficos feitos por IA

Models: A Camada de Dados do Django

Models definem a estrutura dos seus dados e são mapeados automaticamente para tabelas no banco de dados através do poderoso **ORM do Django**.

Definindo um Model

```
# models.py
from django.db import models

class Post(models.Model):
    titulo = models.CharField(max_length=200)
    conteudo = models.TextField()
    autor = models.ForeignKey('auth.User', on_delete=models.CASCADE)
    data_publicacao = models.DateTimeField(auto_now_add=True)
    data_atualizacao = models.DateTimeField(auto_now=True)
    publicado = models.BooleanField(default=False)

    class Meta:
        ordering = ['-data_publicacao']
        verbose_name_plural = 'Posts'

    def __str__(self):
        return self.titulo
```

Tipos de Campos Mais Usados

- `CharField` - Texto curto (`max_length` obrigatório)
- `TextField` - Texto longo
- `IntegerField` - Números inteiros
- `DecimalField` - Números decimais precisos
- `DateTimeField` - Data e hora
- `BooleanField` - Verdadeiro/Falso
- `ForeignKey` - Relacionamento um-para-muitos
- `ManyToManyField` - Relacionamento muitos-para-muitos
- `EmailField` - E-mail com validação

- `URLField` - URL com validação
- `ImageField` - Upload de imagens

Migrations: Gerenciando o Esquema do Banco

Django usa **migrations** para gerenciar alterações no banco de dados:

```
# Criar migrations baseadas em mudanças nos models
python manage.py makemigrations

# Aplicar migrations ao banco de dados
python manage.py migrate

# Ver SQL que será executado (sem executar)
python manage.py sqlmigrate app_name 0001

# Verificar problemas sem executar
python manage.py makemigrations --check
```

Dica Pro: Use `--check` em pipelines de CI/CD para detectar models não migrados.

Artigos relacionados:

- [Django: A Camada Model](#)
- [O comando makemigrations do Django](#)
- [O comando migrate do Django](#)

Django ORM: Consultando Dados

O ORM do Django permite consultas complexas sem escrever SQL:

```
# Buscar todos os posts
Post.objects.all()

# Filtrar posts publicados
Post.objects.filter(publicado=True)

# Buscar um post específico
Post.objects.get(id=1)

# Excluir posts não publicados
Post.objects.exclude(publicado=True)

# Ordenar por data
Post.objects.order_by('-data_publicacao')

# Limitar resultados
Post.objects.all()[:5]

# Busca com múltiplos filtros
Post.objects.filter(
    publicado=True,
    data_publicacao__year=2025
)

# Relacionamentos (joins automáticos)
Post.objects.select_related('autor').all()

# Agregações
from django.db.models import Count
Post.objects.aggregate(total=Count('id'))
```

Métodos de QuerySet essenciais:

- `filter()` - Filtrar registros

- `exclude()` - Excluir registros
- `get()` - Buscar um único registro
- `first()` / `last()` - Primeiro/último registro
- `exists()` - Verificar se existe
- `count()` - Contar registros
- `select_related()` - Otimizar ForeignKey (JOIN)
- `prefetch_related()` - Otimizar ManyToMany

Views: Processando Requisições

Views são funções ou classes que recebem requisições HTTP e retornam respostas.

Function-Based Views (FBV)

```
# views.py
from django.shortcuts import render, get_object_or_404
from .models import Post

def lista_posts(request):
    posts = Post.objects.filter(publicado=True)
    return render(request, 'blog/lista.html', {'posts': posts})

def detalhe_post(request, post_id):
    post = get_object_or_404(Post, id=post_id)
    return render(request, 'blog/detalhe.html', {'post': post})
```

Class-Based Views (CBV)

Django oferece views genéricas que economizam muito código:

```
# views.py
from django.views.generic import ListView, DetailView, CreateView
from .models import Post

class PostListView(ListView):
    model = Post
    template_name = 'blog/lista.html'
    context_object_name = 'posts'
    paginate_by = 10

    def get_queryset(self):
        return Post.objects.filter(publicado=True)

class PostDetailView(DetailView):
    model = Post
    template_name = 'blog/detalhe.html'
    context_object_name = 'post'

class PostCreateView(CreateView):
    model = Post
    fields = ['titulo', 'conteudo']
    template_name = 'blog/criar.html'
    success_url = '/posts/'
```

CBVs genéricas mais usadas:

- `ListView` - Listar registros
- `DetailView` - Exibir um registro
- `CreateView` - Criar registro
- `UpdateView` - Atualizar registro
- `DeleteView` - Deletar registro

- `TemplateView` - Renderizar template simples
- `FormView` - Processar formulários

Django 5.1: CBVs agora suportam métodos assíncronos (`async def get()`, `async def post()`).

Artigos relacionados:

- [Django: A Camada View](#)
- [Como utilizar as Class Based Views do Django](#)

Roteamento de URLs

Configure URLs para mapear para views:

```
# urls.py
from django.urls import path
from . import views

urlpatterns = [
    path('posts/', views.PostListView.as_view(), name='lista_posts'),
    path('posts/<int:pk>/', views.PostDetailView.as_view(),
         name='detalhe_post'),
    path('posts/criar/', views.PostCreateView.as_view(),
         name='criar_post'),
]
```

Django 5.1 usa `path()` como padrão. Para expressões regulares complexas, use `re_path()`:

```
from django.urls import re_path

urlpatterns = [
    re_path(r'^posts/(?P<year>[0-9]{4})/$', views.posts_por_ano),
]
```

Templates: Renderizando HTML

Templates separam lógica de apresentação usando a **Django Template Language (DTL)**.

Sintaxe Básica DTL

```
{% raw %}  
<!-- templates/blog/lista.html -->  
<!DOCTYPE html>  
<html>  
<head>  
    <title>Blog</title>  
</head>  
<body>  
    <h1>Posts do Blog</h1>  
  
    {% if posts %}  
        <ul>  
            {% for post in posts %}  
                <li>  
                    <h2>{{ post.titulo }}</h2>  
                    <p>{{ post.conteudo|truncatewords:30 }}</p>  
                    <small>Por {{ post.autor.username }} em  
{{ post.data_publicacao|date:"d/m/Y" }}</small>  
                </li>  
            {% endfor %}  
        </ul>  
    {% else %}  
        <p>Nenhum post encontrado.</p>  
    {% endif %}  
</body>  
</html>  
{% endraw %}
```

Tags DTL Essenciais

- `{% raw %}{% if %}{% endraw %}` - Condicional
- `{% raw %}{% for %}{% endraw %}` - Loop
- `{% raw %}{% block %}{% endraw %}` - Define bloco para herança
- `{% raw %}{% extends %}{% endraw %}` - Herda de template base

- `{% raw %}{% include %}{% endraw %}` - Inclui outro template
- `{% raw %}{% url %}{% endraw %}` - Gera URL reversa
- `{% raw %}{% csrf_token %}{% endraw %}` - Token CSRF para forms

Filtros Úteis

- `{% raw %}{{ valor|date:"d/m/Y" }}{% endraw %}` - Formatar data
- `{% raw %}{{ texto|truncatewords:10 }}{% endraw %}` - Truncar palavras
- `{% raw %}{{ texto|lower }}{% endraw %}` - Minúsculas
- `{% raw %}{{ texto|upper }}{% endraw %}` - Maiúsculas
- `{% raw %}{{ lista|length }}{% endraw %}` - Tamanho da lista
- `{% raw %}{{ valor|default:"N/A" }}{% endraw %}` - Valor padrão

Template Inheritance (Herança)

```
{% raw %}  
<!-- templates/base.html -->  
<!DOCTYPE html>  
<html>  
<head>  
    <title>{% block title %}Meu Site{% endblock %}</title>  
</head>  
<body>  
    <nav>  
        <!-- Navegação -->  
    </nav>  
  
    <main>  
        {% block content %}{% endblock %}  
    </main>  
  
    <footer>  
        <!-- Rodapé -->  
    </footer>  
</body>  
</html>  
  
<!-- templates/blog/lista.html -->  
{% extends 'base.html' %}  
  
{% block title %}Posts - {{ block.super }}{% endblock %}  
  
{% block content %}  
    <h1>Lista de Posts</h1>  
    <!-- Conteúdo específico -->  
{% endblock %}  
{% endraw %}
```

Django 5.1: Suporte completo para **Jinja2** como backend de templates alternativo para projetos que precisam de mais flexibilidade.

Artigo relacionado:

- [Django: A Camada Template](#)

Forms: Validação e Processamento

Django Forms facilitam criação e validação de formulários HTML.

Django Forms

```
# forms.py
from django import forms

class ContatoForm(forms.Form):
    nome = forms.CharField(max_length=100)
    email = forms.EmailField()
    mensagem = forms.CharField(widget=forms.Textarea)

    def clean_email(self):
        email = self.cleaned_data['email']
        if not email.endswith('@exemplo.com'):
            raise forms.ValidationError('Use e-mail corporativo')
        return email
```

ModelForms

Cria forms automaticamente baseados em Models:

```
# forms.py
from django import forms
from .models import Post

class PostForm(forms.ModelForm):
    class Meta:
        model = Post
        fields = ['titulo', 'conteudo', 'publicado']
        widgets = {
            'conteudo': forms.Textarea(attrs={'rows': 5}),
        }
        labels = {
            'titulo': 'Título do Post',
        }
```

Processando Forms em Views

```
# views.py
from django.shortcuts import render, redirect
from .forms import PostForm

def criar_post(request):
    if request.method == 'POST':
        form = PostForm(request.POST)
        if form.is_valid():
            form.save()
            return redirect('lista_posts')
    else:
        form = PostForm()

    return render(request, 'blog/criar.html', {'form': form})
```

Renderizando Forms em Templates

```
{% raw %}  
<!-- templates/blog/criar.html -->  
<form method="post">  
    {% csrf_token %}  
    {{ form.as_p }}  
    <button type="submit">Salvar</button>  
</form>  
  
<!-- Ou com mais controle -->  
<form method="post">  
    {% csrf_token %}  
    {% for field in form %}  
        <div class="form-group">  
            {{ field.label_tag }}  
  
            {% if field.errors %}  
                <div class="error">{{ field.errors }}</div>  
            {% endif %}  
        </div>  
    {% endfor %}  
    <button type="submit">Salvar</button>  
</form>  
{% endraw %}
```

Artigo relacionado:

- [Formulários do Django com Django Forms](#)

Django Admin: Painel Administrativo Poderoso

Uma das features mais impressionantes do Django é o **Admin automático**.

Configurando o Admin

```
# admin.py
from django.contrib import admin
from .models import Post

@admin.register(Post)
class PostAdmin(admin.ModelAdmin):
    list_display = ['titulo', 'autor', 'data_publicacao', 'publicado']
    list_filter = ['publicado', 'data_publicacao']
    search_fields = ['titulo', 'conteudo']
    prepopulated_fields = {'slug': ('titulo',)}
    date_hierarchy = 'data_publicacao'
    ordering = ['-data_publicacao']
    list_editable = ['publicado']

    fieldsets = (
        ('Informações Básicas', {
            'fields': ('titulo', 'slug', 'autor')
        }),
        ('Conteúdo', {
            'fields': ('conteudo',)
        }),
        ('Publicação', {
            'fields': ('publicado', 'data_publicacao'),
            'classes': ('collapse',)
        }),
    )
)
```

Criando Superusuário

```
python manage.py createsuperuser
```

Acesse `/admin/` e faça login!

Recursos do Admin Django 5.1:

-  Dark Mode automático
-  Acessibilidade WCAG 2.1 AA
-  Interface responsiva
-  Filtros avançados
-  Busca integrada
-  Ações em lote customizáveis
-  Inline editing de relacionamentos

Artigo relacionado:

- [O Painel Administrativo do Django](#)

Middlewares: Processamento de Requisições

Middlewares interceptam requisições/respostas para adicionar funcionalidades globais.

Estrutura de um Middleware

```
# middleware.py
class LogMiddleware:
    def __init__(self, get_response):
        self.get_response = get_response

    def __call__(self, request):
        # Código executado ANTES da view
        print(f'Requisição para: {request.path}')

        response = self.get_response(request)

        # Código executado DEPOIS da view
        print(f'Status: {response.status_code}')

        return response

    def process_exception(self, request, exception):
        # Executado quando há exceção
        print(f'Erro: {exception}')
        return None
```

Async Middlewares (Django 5.1)

```
# middleware.py
class AsyncLogMiddleware:
    async_capable = True

    def __init__(self, get_response):
        self.get_response = get_response

    @async def __call__(self, request):
        # Operações assíncronas
        await self.log_request(request)
        response = await self.get_response(request)
        return response

    @async def log_request(self, request):
        # Lógica assíncrona
        pass
```

Registrando Middleware

```
# settings.py
MIDDLEWARE = [
    'django.middleware.security.SecurityMiddleware',
    'django.contrib.sessions.middleware.SessionMiddleware',
    'django.middleware.common.CommonMiddleware',
    'django.middleware.csrf.CsrfViewMiddleware',
    'meu_app.middleware.LogMiddleware', # Seu middleware
    ...
]
```

Artigo relacionado:

- Como criar Middlewares no Django

Bancos de Dados: PostgreSQL e MySQL

Por padrão, Django usa **SQLite** para desenvolvimento, mas para produção você deve usar bancos de dados mais robustos como **PostgreSQL** ou **MySQL**.

Por que mudar do **SQLite**?

- SQLite não suporta múltiplas conexões simultâneas
- Falta recursos avançados (full-text search, JSON fields complexos)
- Não é adequado para alta concorrência
- PostgreSQL/MySQL são otimizados para produção

Configurando PostgreSQL

PostgreSQL é o banco de dados mais recomendado para Django. Ele oferece:

- Recursos avançados (JSONB, full-text search, arrays)
- Excelente performance
- Conformidade com SQL
- Suporte completo do Django para features específicas

Passo 1: Configure o dicionário `DATABASES` no seu `settings.py`:

```
# settings.py
DATABASES = {
    'default': { # Nome da conexão (pode ter múltiplas)
        'ENGINE': 'django.db.backends.postgresql', # Driver PostgreSQL
        'NAME': 'meu_banco', # Nome do banco de dados
        'USER': 'usuario', # Usuário do PostgreSQL
        'PASSWORD': 'senha', # Senha (use variáveis de ambiente!)
        'HOST': 'localhost', # Servidor (localhost para dev, IP/
                             # domínio para prod)
        'PORT': '5432', # Porta padrão do PostgreSQL
    }
}
```

Segurança: Nunca deixe senhas hardcoded! Use variáveis de ambiente:
'PASSWORD': os.environ.get('DB_PASSWORD')

Passo 2: Instale o driver (adaptador) Python para PostgreSQL:

```
# psycopg2 (tradicional, estável)
pip install psycopg2-binary

# psycopg3 (Django 5.1+ - recomendado para novos projetos)
pip install psycopg[binary]
```

Qual driver usar?

- `psycopg2-binary` : Versão tradicional, binários pré-compilados. Use em desenvolvimento.
- `psycopg2` : Versão para compilar localmente. Use em produção (mais otimizado).
- `psycopg3` : Nova geração, suporte async completo, melhor performance. Recomendado para Django 5.1+.

Django 5.1: *psycopg3 oferece melhor performance, suporte async nativo e API modernizada. Se está começando um projeto novo, use psycopg3!*

Configurando MySQL

MySQL e **MariaDB** são alternativas populares, especialmente em ambientes de hospedagem compartilhada.

Quando usar MySQL:

- Já tem expertise com MySQL
- Infraestrutura existente usa MySQL
- Hospedagem compartilhada (mais comum que PostgreSQL)
- Integração com outras ferramentas MySQL

Passo 1: Configure no `settings.py`:

```
# settings.py
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.mysql', # Driver MySQL
        'NAME': 'meu_banco', # Nome do banco
        'USER': 'usuario', # Usuário MySQL
        'PASSWORD': 'senha', # Senha (use variável de ambiente!)
        'HOST': 'localhost', # Servidor
        'PORT': '3306', # Porta padrão MySQL
        'OPTIONS': {
            'charset': 'utf8mb4', # Charset para suportar emojis e
            # caracteres especiais
            'init_command': "SET sql_mode='STRICT_TRANS_TABLES'", #
            # Modo estrito
        },
    }
}
```

Por que `utf8mb4`? É a versão completa do UTF-8 no MySQL, suportando emojis (🚀) e caracteres de 4 bytes.

Passo 2: Instale o driver MySQL:

```
# mysqlclient (recomendado - mais rápido, escrito em C)
pip install mysqlclient

# PyMySQL (alternativa - Python puro, mais fácil de instalar)
pip install pymysql
```

Qual driver usar?

- `mysqlclient` : Mais rápido (código C), mas pode ter problemas de instalação em Windows.
- `PyMySQL` : Mais lento, mas 100% Python, instala em qualquer lugar.

Se usar PyMySQL, adicione no `__init__.py` do projeto:

```
import pymysql  
pymysql.install_as_MySQLdb() # Faz PyMySQL se passar por MySQLdb
```

Django 5.1 requer MySQL 8.0+ ou MariaDB 10.5+. Versões antigas não são mais suportadas!

Artigos relacionados:

- [Como conectar Django ao PostgreSQL](#)
- [Como conectar Django ao MySQL](#)

Múltiplos Bancos de Dados

Django permite usar **múltiplos bancos de dados** no mesmo projeto. Isso é útil para:

- Separar dados de aplicação de analytics
- Ler de réplicas (read-only databases)
- Isolar dados de diferentes clientes (multi-tenancy)
- Migrar gradualmente entre bancos

Exemplo: Aplicação principal em PostgreSQL, analytics em outro banco:

```

# settings.py
DATABASES = {
    'default': { # Banco principal da aplicação
        'ENGINE': 'django.db.backends.postgresql',
        'NAME': 'principal',
        'USER': 'app_user',
        'PASSWORD': 'senha',
        'HOST': 'db.exemplo.com',
    },
    'analytics': { # Banco separado para analytics/relatórios
        'ENGINE': 'django.db.backends.postgresql',
        'NAME': 'analytics',
        'USER': 'analytics_user',
        'PASSWORD': 'outra_senha',
        'HOST': 'analytics.exemplo.com',
    }
}

```

Como usar bancos específicos:

```

# Ler do banco de analytics
posts_analytics = Post.objects.using('analytics').all()

# Salvar no banco de analytics
post = Post(titulo='Teste')
post.save(using='analytics')

# Deletar do banco de analytics
post.delete(using='analytics')

# Query que junta dados de ambos os bancos NÃO é possível!
# Você precisa fazer queries separadas e juntar no Python

```

Database Routers (Avançado): Para controlar automaticamente qual model vai para qual banco, crie um router:

```

# database_router.py
class AnalyticsRouter:
    """Direciona models do app 'analytics' para o banco 'analytics'"""

    def db_for_read(self, model, **hints):
        if model._meta.app_label == 'analytics':
            return 'analytics'
        return None

    def db_for_write(self, model, **hints):
        if model._meta.app_label == 'analytics':
            return 'analytics'
        return None

# settings.py
DATABASE_ROUTERS = ['caminho.database_router.AnalyticsRouter']

```

Django REST Framework: Criando APIs Profissionais

Django REST Framework (DRF) é uma biblioteca poderosa que transforma seu projeto Django em uma **API RESTful completa**.

O que é uma API REST? REST (Representational State Transfer) é um padrão arquitetural para criar APIs web que usam métodos HTTP (GET, POST, PUT, DELETE) para operar sobre recursos.

Por que usar DRF?

- Serialização automática de models para JSON
- Autenticação e permissões prontas
- Interface web para testar a API (Browsable API)
- Paginação, filtragem e busca automáticas

- Documentação automática (com Swagger/OpenAPI)

Instalação

Passo 1: Instale o DRF:

```
pip install djangorestframework
```

Passo 2: Adicione às apps instaladas:

```
# settings.py
INSTALLED_APPS = [
    'django.contrib.admin',
    'django.contrib.auth',
    # ... outras apps padrão do Django

    'rest_framework', # Adicione o DRF aqui

    # Suas apps
    'blog',
]
```

Serializers: Convertendo Models em JSON

Serializers são a ponte entre seus **Models Django** (objetos Python) e **JSON** (formato usado em APIs).

O que fazem:

1. **Serialização:** Model → JSON (para enviar ao cliente)
2. **Desserialização:** JSON → Model (ao receber dados do cliente)
3. **Validação:** Garantem que dados recebidos estão corretos

Exemplo prático: Vamos criar um serializer para nosso model `Post` :

```

# serializers.py
from rest_framework import serializers
from .models import Post

class PostSerializer(serializers.ModelSerializer):
    # Campo customizado: pega o username do autor ao invés do ID
    # source='autor.username' = acessa o atributo username do objeto
    # autor
    # read_only=True = não pode ser alterado via API (calculado
    # automaticamente)
    autor_nome = serializers.CharField(
        source='autor.username',
        read_only=True
    )

    # Campo calculado: conta quantos comentários o post tem
    total_comentarios = serializers.SerializerMethodField()

    class Meta:
        model = Post # Model que será serializado

        # Campos que serão incluídos no JSON
        fields = [
            'id',
            'titulo',
            'conteudo',
            'autor_nome', # Campo customizado
            'data_publicacao',
            'total_comentarios' # Campo calculado
        ]

        # Campos que não podem ser alterados via API
        # (são preenchidos automaticamente pelo Django)
        read_only_fields = ['data_publicacao']

    def get_total_comentarios(self, obj):
        """Método para o SerializerMethodField"""
        # obj = instância do Post sendo serializada
        return obj.comentarios.count()

    def validate_titulo(self, value):

```

```

"""
Validação customizada para o campo titulo.
Django chama automaticamente validate_<nome_do_campo>().
"""

if len(value) < 5:
    raise serializers.ValidationError(
        'Título deve ter pelo menos 5 caracteres'
    )
if 'spam' in value.lower():
    raise serializers.ValidationError(
        'Título contém conteúdo proibido'
    )
return value

def validate(self, data):
    """
    Validação que envolve múltiplos campos.
    Chamado depois de todas as validações individuais.
    """

    # Exemplo: título e conteúdo não podem ser iguais
    if data.get('titulo') == data.get('conteudo'):
        raise serializers.ValidationError(
            'Título e conteúdo não podem ser idênticos'
        )
    return data

```

O que acontece quando usamos este serializer:

```

# Serialização (Model → JSON)
post = Post.objects.get(id=1)
serializer = PostSerializer(post)
print(serializer.data)
# Output: {
#   'id': 1,
#   'titulo': 'Meu Post',
#   'conteudo': 'Conteúdo...',
#   'autor_nome': 'joao', # Veio de autor.username
#   'data_publicacao': '2025-07-20T10:00:00Z',
#   'total_comentarios': 5 # Calculado pelo método
# }

# Dessaerialização (JSON → Model)
data = {
    'titulo': 'Novo Post',
    'conteudo': 'Conteúdo do post...',
}
serializer = PostSerializer(data=data)
if serializer.is_valid(): # Valida os dados
    post = serializer.save(autor=request.user) # Cria o Post
    print(f'Post criado: {post.id}')
else:
    print(serializer.errors) # Mostra erros de validação

```

ViewSets: CRUD Automático

ViewSets são classes que fornecem **operações CRUD completas** (Create, Read, Update, Delete) com poucas linhas de código.

O que um **ViewSet** faz automaticamente:

- `list()` - GET /api/posts/ (listar todos)
- `create()` - POST /api/posts/ (criar novo)
- `retrieve()` - GET /api/posts/{id}/ (buscar um)

- `update()` - PUT /api/posts/{id}/ (atualizar completo)
- `partial_update()` - PATCH /api/posts/{id}/ (atualizar parcial)
- `destroy()` - DELETE /api/posts/{id}/ (deletar)

Exemplo prático:ViewSet completo com todas as features:

```

# views.py

from rest_framework import viewsets, status
from rest_framework.permissions import IsAuthenticated,
    IsAuthenticatedOrReadOnly
from rest_framework.decorators import action
from rest_framework.response import Response
from django_filters.rest_framework import DjangoFilterBackend
from rest_framework import filters

from .models import Post
from .serializers import PostSerializer


class PostViewSet(viewsets.ModelViewSet):
    """
    ViewSet para gerenciar Posts.
    Fornece endpoints: list, create, retrieve, update, destroy.
    """

    # QuerySet base - apenas posts publicados
    queryset = Post.objects.filter(publicado=True).select_related('autor')

    # Serializer que será usado para converter dados
    serializer_class = PostSerializer

    # Permissões: Leitura pública, escrita apenas autenticado
    permission_classes = [IsAuthenticatedOrReadOnly]

    # Habilita filtragem, busca e ordenação
    filter_backends = [
        DjangoFilterBackend, # Filtragem por campos exatos
        filters.SearchFilter, # Busca textual
        filters.OrderingFilter # Ordenação
    ]

    # Campos que podem ser filtrados
    # Ex: /api/posts/?autor=5&publicado=true
    filterset_fields = ['autor', 'publicado', 'data_publicacao']

    # Campos onde busca textual funciona
    # Ex: /api/posts/?search=django

```

```

search_fields = ['titulo', 'conteudo', 'autor__username']

# Campos que podem ser usados para ordenar
# Ex: /api/posts/?ordering=-data_publicacao
ordering_fields = ['data_publicacao', 'titulo']
ordering = ['-data_publicacao'] # Ordenação padrão

def get_queryset(self):
    """
    Customiza o queryset baseado no usuário.
    Chamado automaticamente pelo DRF.
    """
    queryset = super().get_queryset()

    # Se usuário autenticado, mostra seus posts não publicados
    # também
    if self.request.user.is_authenticated:
        queryset = Post.objects.filter(
            publicado=True
        ) | Post.objects.filter(
            autor=self.request.user
        )

    return queryset

def perform_create(self, serializer):
    """
    Customiza a criação de um post.
    Define o autor automaticamente como o usuário logado.
    """
    serializer.save(autor=self.request.user)

@action(detail=True, methods=['post'])
def publicar(self, request, pk=None):
    """
    Ação customizada: publicar um post.
    Acessível em: POST /api/posts/{id}/publicar/
    """
    post = self.get_object()

    # Verifica se usuário é o autor
    if post.autor != request.user:

```

```

    return Response(
        {'erro': 'Apenas o autor pode publicar'},
        status=status.HTTP_403_FORBIDDEN
    )

post.publicado = True
post.save()

serializer = self.get_serializer(post)
return Response(serializer.data)

[action(detail=False, methods=['get'])]
def meus_posts(self, request):
    """
    Ação customizada: listar posts do usuário logado.
    Acessível em: GET /api/posts/meus_posts/
    """
    posts = Post.objects.filter(autor=request.user)

    # Página os resultados
    page = self.paginate_queryset(posts)
    if page is not None:
        serializer = self.get_serializer(page, many=True)
        return self.get_paginated_response(serializer.data)

    serializer = self.get_serializer(posts, many=True)
    return Response(serializer.data)

```

O que cada parte faz:

- `get_queryset()`: Customiza quais objetos são retornados
- `perform_create()`: Adiciona lógica extra ao criar
- `@action()`: Cria endpoints customizados além do CRUD padrão
- `detail=True`: Ação funciona em um objeto específico (`/posts/{id}/publicar/`)
- `detail=False`: Ação funciona na coleção (`/posts/meus_posts/`)

Routers: URLs Automáticas

Routers do DRF criam automaticamente todas as URLs necessárias para seus ViewSets.

Sem Router (modo manual - trabalhoso):

```
# urls.py - EVITE FAZER ASSIM!
urlpatterns = [
    path('posts/', PostViewSet.as_view({'get': 'list', 'post':
        'create'})),
    path('posts/<int:pk>/', PostViewSet.as_view({
        'get': 'retrieve',
        'put': 'update',
        'patch': 'partial_update',
        'delete': 'destroy'
    })),
]
```

Com Router (recomendado - automático):

```
# urls.py
from django.urls import path, include
from rest_framework.routers import DefaultRouter
from .views import PostViewSet

# Cria o router
router = DefaultRouter()

# Registra o ViewSet
# 'posts' = prefixo da URL
# PostViewSet = a classe ViewSet
router.register(r'posts', PostViewSet, basename='post')

# Inclui as URLs geradas pelo router
urlpatterns = [
    path('api/', include(router.urls)),
]
```

URLs geradas automaticamente pelo Router:

Método HTTP	URL	Ação	Descrição
GET	/api/posts/	list	Lista todos os posts
POST	/api/posts/	create	Cria novo post
GET	/api/posts/{id}/	retrieve	Busca post específico
PUT	/api/posts/{id}/	update	Atualiza post completo
PATCH	/api/posts/{id}/	partial_update	Atualiza parcialmente
DELETE	/api/posts/{id}/	destroy	Deleta post
POST	/api/posts/{id}/publicar/	publicar	Ação custom
GET	/api/posts/meus_posts/	meus_posts	Ação custom

Exemplo de uso da API:

```

# Listar todos os posts
curl http://localhost:8000/api/posts/

# Buscar um post específico
curl http://localhost:8000/api/posts/5/

# Criar novo post (precisa autenticação)
curl -X POST http://localhost:8000/api/posts/ \
-H "Authorization: Token seu-token-aqui" \
-H "Content-Type: application/json" \
-d '{"titulo": "Novo Post", "conteudo": "Conteúdo..."}'

# Filtrar posts por autor
curl http://localhost:8000/api/posts/?autor=5

# Buscar por texto
curl http://localhost:8000/api/posts/?search=django

# Ordenar por data
curl http://localhost:8000/api/posts/?ordering=-data_publicacao

```

Autenticação e Permissões

DRF oferece **autenticação** (quem é o usuário?) e **permissões** (o que ele pode fazer?).

Tipos de Autenticação:

- `SessionAuthentication` : Usa sessões Django (cookies)
- `TokenAuthentication` : Token fixo por usuário
- `JWTAuthentication` : JSON Web Tokens (mais seguro)
- `BasicAuthentication` : HTTP Basic (username:password em base64)

Tipos de Permissões:

- `AllowAnonymous` : Acesso público total
- `IsAuthenticated` : Apenas usuários logados
- `IsAdminUser` : Apenas administradores
- `IsAuthenticatedOrReadOnly` : Leitura pública, escrita autenticada

Configuração global:

```

# settings.py
REST_FRAMEWORK = {
    # Métodos de autenticação aceitos
    'DEFAULT_AUTHENTICATION_CLASSES': [
        # Token: usuário envia "Authorization: Token abc123"
        'rest_framework.authentication.TokenAuthentication',
        # Session: usa cookies do Django (para Browsable API)
        'rest_framework.authentication.SessionAuthentication',
    ],

    # Permissão padrão: apenas usuários autenticados
    'DEFAULT_PERMISSION_CLASSES': [
        'rest_framework.permissions.IsAuthenticated',
    ],

    # Paginação automática
    'DEFAULT_PAGINATION_CLASS': (
        'rest_framework.pagination.PageNumberPagination'
    ),
    'PAGE_SIZE': 20, # 20 itens por página

    # Formatos de resposta aceitos
    'DEFAULT_RENDERER_CLASSES': [
        'rest_framework.renderers.JSONRenderer', # JSON
        'rest_framework.renderers.BrowsableAPIRenderer', # HTML
    ],

    # Throttling (limite de requisições)
    'DEFAULT_THROTTLE_CLASSES': [
        'rest_framework.throttling.AnonRateThrottle', # Anônimos
        'rest_framework.throttling.UserRateThrottle', # Autenticados
    ],
    'DEFAULT_THROTTLE_RATES': {
        'anon': '100/day', # 100 requisições por dia
        'user': '1000/day', # 1000 requisições por dia
    },
}

```

Usando Token Authentication:

```

# Adicione 'rest_framework.authtoken' ao INSTALLED_APPS
INSTALLED_APPS = [
    # ...
    'rest_framework',
    'rest_framework.authtoken', # Para usar tokens
]

# Rode as migrations
# python manage.py migrate

# Crie tokens para usuários
from rest_framework.authtoken.models import Token
from django.contrib.auth.models import User

user = User.objects.get(username='joao')
token = Token.objects.create(user=user)
print(token.key) # "9944b09199c62bcf9418ad846dd0e4bbdfc6ee4b"

# Cliente usa o token:
# Authorization: Token 9944b09199c62bcf9418ad846dd0e4bbdfc6ee4b

```

Permissões customizadas:

```

# permissions.py
from rest_framework import permissions

class IsAuthorOrReadOnly(permissions.BasePermission):
    """
    Permissão customizada: apenas o autor pode editar.
    """

    def has_object_permission(self, request, view, obj):
        # Leitura permitida para todos
        if request.method in permissions.SAFE_METHODS: # GET, HEAD,
            OPTIONS
            return True

        # Escrita apenas para o autor
        return obj.autor == request.user

# views.py
from .permissions import IsAuthorOrReadOnly

class PostViewSet(viewsets.ModelViewSet):
    # ...
    permission_classes = [IsAuthenticated, IsAuthorOrReadOnly]

```

Browsable API

DRF fornece interface web interativa automaticamente:

```

# Acesse no navegador
http://localhost:8000/api/posts/

```

Features da Browsable API:

- Interface amigável para testar endpoints
- Autenticação integrada
- Formulários para POST/PUT

-  Dark mode (DRF 3.15+)

Artigo relacionado:

- [O que é o Django REST Framework](#)

Comandos manage.py Essenciais

O `manage.py` é sua interface de linha de comando para tudo no Django.

Comandos de Desenvolvimento

```
# Iniciar servidor de desenvolvimento
python manage.py runserver
python manage.py runserver 0.0.0.0:8080 # Porta customizada

# Criar nova app
python manage.py startapp nome_da_app

# Shell Python com Django carregado
python manage.py shell

# Shell do banco de dados
python manage.py dbshell
```

Comandos de Banco de Dados

```
# Criar migrations
python manage.py makemigrations
python manage.py makemigrations nome_da_app # App específica

# Aplicar migrations
python manage.py migrate
python manage.py migrate nome_da_app # App específica

# Reverter migration
python manage.py migrate nome_da_app 0003 # Volta para migration 0003

# Ver migrations
python manage.py showmigrations

# SQL da migration (sem executar)
python manage.py sqlmigrate nome_da_app 0001
```

Comandos de Admin

```
# Criar superusuário
python manage.py createsuperuser

# Alterar senha de usuário
python manage.py changepassword username
```

Comandos de Arquivos Estáticos

```
# Coletar arquivos estáticos para produção
python manage.py collectstatic

# Coletar sem confirmação
python manage.py collectstatic --noinput
```

Comandos de Testes

```
# Executar todos os testes  
python manage.py test  
  
# Testar app específica  
python manage.py test nome_da_app  
  
# Teste com cobertura  
python manage.py test --with-coverage
```

Comandos de Verificação

```
# Verificar projeto inteiro  
python manage.py check  
  
# Verificar se há migrations pendentes (CI/CD)  
python manage.py makemigrations --check --dry-run
```

Artigo relacionado:

- Os principais comandos do manage.py

Boas Práticas Django 2025

Estrutura de Projeto

```
projeto/
├── apps/
│   ├── blog/
│   │   ├── migrations/
│   │   ├── templates/blog/
│   │   ├── static/blog/
│   │   ├── models.py
│   │   ├── views.py
│   │   ├── urls.py
│   │   ├── admin.py
│   │   ├── forms.py
│   │   └── tests.py
│   └── usuarios/
├── config/
│   ├── settings/
│   │   ├── base.py
│   │   ├── development.py
│   │   └── production.py
│   ├── urls.py
│   └── wsgi.py
├── static/
├── media/
├── templates/
├── requirements/
│   ├── base.txt
│   ├── development.txt
│   └── production.txt
└── manage.py
```

Settings Separados por Ambiente

```
# config/settings/base.py
# Configurações comuns

# config/settings/development.py
from .base import *

DEBUG = True
ALLOWED_HOSTS = ['localhost', '127.0.0.1']

# config/settings/production.py
from .base import *

DEBUG = False
ALLOWED_HOSTS = ['meusite.com']
SECURE_SSL_REDIRECT = True
```

```
# Usar settings específico
python manage.py runserver --settings=config.settings.development
```

Variáveis de Ambiente

Use `python-decouple` ou `django-environ`:

```
# settings.py
from decouple import config

SECRET_KEY = config('SECRET_KEY')
DEBUG = config('DEBUG', default=False, cast=bool)
DATABASE_URL = config('DATABASE_URL')
```

```
# .env
SECRET_KEY=sua-chave-secreta-aqui
DEBUG=True
DATABASE_URL=postgresql://user:pass@localhost/db
```

Segurança

```
# settings.py (Produção)

# HTTPS
SECURE_SSL_REDIRECT = True
SESSION_COOKIE_SECURE = True
CSRF_COOKIE_SECURE = True

# HSTS
SECURE_HSTS_SECONDS = 31536000
SECURE_HSTS_INCLUDE_SUBDOMAINS = True
SECURE_HSTS_PRELOAD = True

# Outras
SECURE_CONTENT_TYPE_NOSNIFF = True
SECURE_BROWSER_XSS_FILTER = True
X_FRAME_OPTIONS = 'DENY'
```

Performance

1. Use `select_related` e `prefetch_related`

```

# ❌ Ruim - N+1 queries
posts = Post.objects.all()
for post in posts:
    print(post.autor.username) # Query a cada iteração

# ✅ Bom - 1 query com JOIN
posts = Post.objects.select_related('autor').all()
for post in posts:
    print(post.autor.username) # Sem queries extras

```

2. Use Cache

```

# settings.py
CACHES = {
    'default': {
        'BACKEND': 'django.core.cache.backends.redis.RedisCache',
        'LOCATION': 'redis://127.0.0.1:6379/1',
    }
}

# views.py
from django.views.decorators.cache import cache_page

@cache_page(60 * 15) # Cache por 15 minutos
def lista_posts(request):
    posts = Post.objects.all()
    return render(request, 'lista.html', {'posts': posts})

```

3. Use Database Indexing

```
class Post(models.Model):
    titulo = models.CharField(max_length=200, db_index=True)
    slug = models.SlugField(unique=True, db_index=True)

    class Meta:
        indexes = [
            models.Index(fields=['publicado', '-data_publicacao']),
        ]
```

4. Use Django Debug Toolbar (Dev)

```
pip install django-debug-toolbar
```

```
# settings.py (development)
INSTALLED_APPS = [
    # ...
    'debug_toolbar',
]

MIDDLEWARE = [
    'debug_toolbar.middleware.DebugToolbarMiddleware',
    # ...
]

INTERNAL_IPS = ['127.0.0.1']
```

Testes

```
# tests.py
from django.test import TestCase
from .models import Post

class PostModelTest(TestCase):
    @classmethod
    def setUpTestData(cls):
        Post.objects.create(titulo='Teste', conteudo='Conteúdo teste')

    def test_titulo_max_length(self):
        post = Post.objects.get(id=1)
        max_length = post._meta.get_field('titulo').max_length
        self.assertEqual(max_length, 200)

    def test_str_representation(self):
        post = Post.objects.get(id=1)
        self.assertEqual(str(post), 'Teste')

class PostViewTest(TestCase):
    def test_lista_posts_status_code(self):
        response = self.client.get('/posts/')
        self.assertEqual(response.status_code, 200)

    def test_lista_posts_template(self):
        response = self.client.get('/posts/')
        self.assertTemplateUsed(response, 'blog/lista.html')
```

Async Views (Django 5.1)

Django 5.1 trouxe suporte completo para **programação assíncrona**. Views assíncronas são úteis quando:

- Você precisa fazer **múltiplas requisições externas** (APIs, microsserviços)
- Operações de **I/O não-bloqueantes** (ler arquivos, acessar banco de dados)

- **WebSockets** e conexões de longa duração
- **Alta concorrência** com poucos recursos

Diferença sync vs async:

```
# ❌ SYNC (bloqueante) - 1 requisição por vez
import requests

def buscar_dados(request):
    # Bloqueia por 2 segundos esperando resposta
    r1 = requests.get('https://api1.com/dados')    # 2s
    r2 = requests.get('https://api2.com/dados')    # 2s
    r3 = requests.get('https://api3.com/dados')    # 2s
    # Total: 6 segundos (uma após a outra)
    return JsonResponse({'dados': [r1.json(), r2.json(), r3.json()]})

# ✅ ASYNC (não-bloqueante) - múltiplas requisições simultaneamente
import httpx
import asyncio

async def buscar_dados_async(request):
    async with httpx.AsyncClient() as client:
        # Executa as 3 requisições AO MESMO TEMPO
        tasks = [
            client.get('https://api1.com/dados'),
            client.get('https://api2.com/dados'),
            client.get('https://api3.com/dados'),
        ]
        responses = await asyncio.gather(*tasks)
        # Total: ~2 segundos (todas em paralelo)
        dados = [r.json() for r in responses]
    return JsonResponse({'dados': dados})
```

Function-Based Async View:

```

# views.py
from django.http import JsonResponse
import httpx
import asyncio

async def buscar_dados_externos(request):
    """
    View assíncrona que consulta múltiplas APIs externas.
    Django 5.1 detecta automaticamente que é async (async def).
    """

    async with httpx.AsyncClient() as client:
        # Consulta várias APIs em paralelo
        clima_task = client.get('https://api.clima.com/cidade/SP')
        noticias_task = client.get('https://api.noticias.com/ultimas')
        cotacao_task = client.get('https://api.moedas.com/USD')

        # Aguarda todas terminarem (em paralelo)
        clima, noticias, cotacao = await asyncio.gather(
            clima_task,
            noticias_task,
            cotacao_task
        )

    return JsonResponse({
        'clima': clima.json(),
        'noticias': noticias.json(),
        'cotacao': cotacao.json(),
    })

# urls.py
from django.urls import path
from . import views

urlpatterns = [
    # Django roda automaticamente em modo async
    path('dados/', views.buscar_dados_externos),
]

```

Class-Based Async View:

```

# views.py
from django.views import View
from django.http import JsonResponse
import httpx

class AsyncDashboardView(View):
    """
    CBV assíncrona: todos os métodos HTTP podem ser async.
    """

    @async def get(self, request):
        """
        Método GET assíncrono.
        """

        # Busca dados do banco de forma assíncrona
        from .models import Post
        from asgiref.sync import sync_to_async

        # Converte operações sync do ORM para async
        posts = await sync_to_async(list)(
            Post.objects.filter(publicado=True)[:10]
        )

        # Busca dados externos
        async with httpx.AsyncClient() as client:
            stats = await client.get('https://api.analytics.com/stats')

        return JsonResponse({
            'posts_count': len(posts),
            'stats': stats.json(),
        })

    @async def post(self, request):
        """
        Método POST assíncrono.
        """

        import json
        data = json.loads(request.body)

        # Processa dados de forma assíncrona
        result = await self.processar_async(data)

```

```
return JsonResponse({'resultado': resultado})  
  
async def processar_async(self, data):  
    """Método auxiliar assíncrono"""  
    # Simula processamento pesado  
    await asyncio.sleep(0.1)  
    return f"Processado: {data}"
```

Async com Django ORM:

O ORM do Django é síncrono por padrão. Use `sync_to_async` para integrar:

```

# views.py
from django.http import JsonResponse
from asgiref.sync import sync_to_async
from .models import Post

async def lista_posts_async(request):
    """
    View assíncrona que consulta o banco de dados.
    """

    # Converte query síncrona para assíncrona
    @sync_to_async
    def get_posts():
        return list(Post.objects.filter(publicado=True).values(
            'id', 'titulo', 'data_publicacao'
        ))

    # Executa de forma assíncrona
    posts = await get_posts()

    return JsonResponse({'posts': posts}, safe=False)

# Ou use diretamente:
async def criar_post_async(request):
    # Converte operação de criação para async
    post = await sync_to_async(Post.objects.create)(
        titulo='Post Async',
        conteudo='Criado de forma assíncrona',
        autor=request.user
    )

    return JsonResponse({'id': post.id, 'titulo': post.titulo})

```

Quando usar Async:

Use `async` quando:

- Fazer múltiplas requisições HTTP externas
- WebSockets e Server-Sent Events

- Streaming de dados
- I/O pesado (upload/download de arquivos)
- Integração com sistemas externos

NÃO use async quando:

- Operações CPU-bound (cálculos pesados)
- CRUD simples do Django
- Queries normais do ORM
- A view é rápida e simples

Configuração ASGI para Async:

```
# asgi.py (já vem configurado no Django 5.1)
import os
from django.core.asgi import get_asgi_application

os.environ.setdefault('DJANGO_SETTINGS_MODULE', 'projeto.settings')
application = get_asgi_application()

# Rodar com servidor ASGI (uvicorn, daphne, hypercorn)
# uvicorn projeto.asgi:application --reload
```

Instale servidor ASGI:

```
# Uvicorn (recomendado)
pip install uvicorn
python -m uvicorn projeto.asgi:application --reload

# Ou Daphne (do Django Channels)
pip install daphne
daphne projeto.asgi:application
```

Deploy e Produção

Checklist de Produção

```
# Verificar segurança  
python manage.py check --deploy
```

Checklist:

- DEBUG = False
- SECRET_KEY em variável de ambiente
- ALLOWED_HOSTS configurado
- HTTPS habilitado
- Banco de dados de produção (PostgreSQL/MySQL)
- Arquivos estáticos servidos por CDN/Nginx
- Media files em storage externo (S3)
- Logging configurado
- Backups automatizados
- Monitoring (Sentry, New Relic)

Servindo com Gunicorn

```
pip install gunicorn

# WSGI (tradicional)
gunicorn config.wsgi:application --bind 0.0.0.0:8000

# ASGI (async - Django 5.1)
pip install gunicorn[gevent]
gunicorn config.asgi:application -k uvicorn.workers.UvicornWorker
```

Arquivos Estáticos

```
# settings.py
STATIC_URL = '/static/'
STATIC_ROOT = BASE_DIR / 'staticfiles'

MEDIA_URL = '/media/'
MEDIA_ROOT = BASE_DIR / 'mediafiles'
```

```
python manage.py collectstatic
```

Nginx Config

```
server {  
    listen 80;  
    server_name meusite.com;  
  
    location /static/ {  
        alias /var/www/projeto/staticfiles/;  
    }  
  
    location /media/ {  
        alias /var/www/projeto/mediafiles/;  
    }  
  
    location / {  
        proxy_pass http://127.0.0.1:8000;  
        proxy_set_header Host $host;  
        proxy_set_header X-Real-IP $remote_addr;  
    }  
}
```

Recursos Adicionais

Documentação Oficial

- [Django 5.1 Documentation](#)
- [Django REST Framework Documentation](#)

Ferramentas Recomendadas

- **django-extensions** - Comandos úteis extras
- **django-debug-toolbar** - Debug em desenvolvimento

- **celery** - Tarefas assíncronas
- **django-cors-headers** - CORS para APIs
- **django-filter** - Filtros avançados
- **drf-spectacular** - OpenAPI/Swagger para DRF

Comunidade

- Django Forum
- Django Discord
- Stack Overflow Django Tag

Conclusão

Django em 2025 é mais poderoso do que nunca.

Com **Django 5.1 LTS**, você tem:

- Suporte async completo (views, middlewares, ORM)
- Segurança de nível enterprise
- ORM extremamente eficiente
- Admin moderno (dark mode, acessível)
- Ecossistema rico (DRF, Celery, Channels)
- Comunidade ativa e documentação excepcional

Este guia cobriu desde a instalação até deploy em produção, passando por:

- Models e ORM
- Views (FBV e CBV)

- Templates (DTL)
- Forms e validação
- Admin customizado
- Middlewares
- Bancos de dados (PostgreSQL/MySQL)
- Django REST Framework
- Comandos manage.py
- Boas práticas
- Performance e segurança

Próximos passos:

1. Pratique criando um projeto completo
2. Estude Django REST Framework em profundidade
3. Aprenda deploy com Docker
4. Explore Django Channels (WebSockets)
5. Contribua para projetos open source Django

Django é uma escolha sólida para **qualquer tipo de aplicação web** em 2025: desde blogs simples até plataformas complexas com milhões de usuários.

Agora é com você! 

Quer aprofundar ainda mais? Baixe nosso **ebook GRÁTIS de Desenvolvimento Web com Python e Django**:

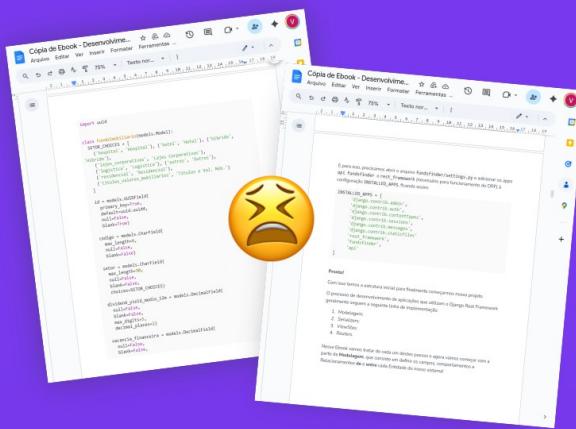
Nos vemos no código! 

Não se esqueça de conferir!



Ebookr

Crie Ebooks profissionais em minutos com IA



Chega de formatar código no Google Docs ou Word



Capas gerados por IA



• Infográficos feitos



Adicione Banners Promocionais

Deixe que nossa IA faça o trabalho pesado



Edite em Markdown em Tempo Real

TESTE AGORA



 PRIMEIRO CAPÍTULO 100% GRÁTIS