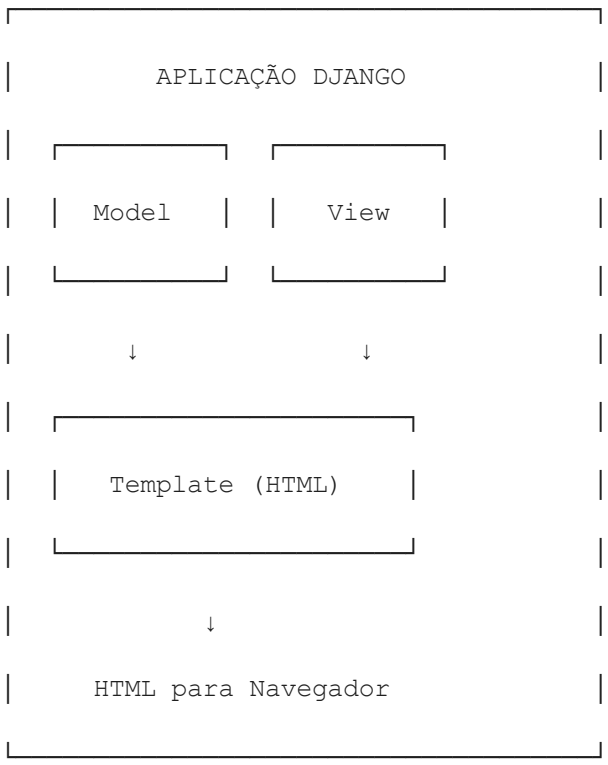
📖 Apostila 1: Arquitetura REST, Verbos HTTP e Setup Completo da API

Capítulo 1: Do Monolítico à API REST 🌐

1.1. A Evolução das Arquiteturas Web

O Paradigma Tradicional: MVT (Model-View-Template)

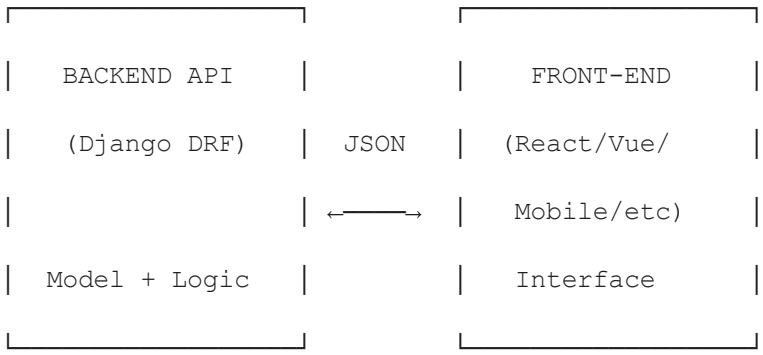
No Django tradicional, construímos aplicações **monolíticas**:



Características:

- Tudo em uma aplicação: banco, lógica e interface
- O servidor gera HTML completo
- Dificulta trabalho em equipe (front e back acoplados)
- Dificulta criação de apps mobile

O Novo Paradigma: REST API



Vantagens:

- **Separação de responsabilidades**: back-end cuida apenas dos dados
- **Flexibilidade**: um back-end, múltiplos clientes (web, mobile, IoT)
- **Trabalho paralelo**: equipes front e back trabalham independentemente
- **Escalabilidade**: cada camada pode escalar separadamente

1.2. O Protocolo HTTP: A Linguagem da Web

HTTP é o protocolo de comunicação da internet. Em APIs REST, usamos os **verbos HTTP** para expressar intenções:

Verbo HTTP	Intenção CRUD	Descrição Detalhada	Exemplo
GET	**READ**	Solicita visualização de dados. Nunca modifica nada. Pode ser repetido sem efeitos colaterais (idempotente).	`GET /api/tarefas/` → Lista todas `GET /api/tarefas/5/` → Mostra a tarefa 5
POST	**CREATE**	Envia dados para criar um novo recurso. Cada chamada cria um novo registro.	`POST /api/tarefas/` → Cria nova tarefa
PUT	**UPDATE (Total)**	Substitui completamente um recurso. Você deve enviar TODOS os campos obrigatórios.	`PUT /api/tarefas/5/` → Atualiza tarefa 5 integralmente
PATCH	**UPDATE (Parcial)**	Atualiza apenas alguns campos de um recurso. Você envia só o que quer mudar.	`PATCH /api/tarefas/5/` → Atualiza apenas `concluida`
DELETE	**DELETE**	Remove um recurso específico do sistema.	`DELETE /api/tarefas/5/` → Remove tarefa 5

1.3. Anatomia de uma Requisição HTTP

Toda comunicação HTTP tem uma estrutura:

```
POST /api/tarefas/ HTTP/1.1

Host: api.exemplo.com

Content-Type: application/json

Authorization: Bearer eyJ0eXAiOiJKV1QiLCJhbGc...



{

  "titulo": "Estudar DRF",

  "concluida": false

}
```

Componentes:

1. **Linha de requisição**: Verbo + URL + Versão HTTP
2. **Headers**: Metadados (tipo de conteúdo, autenticação, etc)
3. **Body**: Dados enviados (no POST/PUT/PATCH)

Resposta do servidor:

```
HTTP/1.1 201 Created

Content-Type: application/json



{

  "id": 42,

  "titulo": "Estudar DRF",

  "concluida": false,

  "criada_em": "2025-11-29T14:30:00Z"

}
```

1.4. REST: Representational State Transfer

REST é um **estilo arquitetural** com princípios:

1. **Cliente-Servidor**: Separação de responsabilidades
2. **Stateless**: Cada requisição é independente (sem sessão no servidor)
3. **Recursos identificados por URLs**: `/api/tarefas/5/` identifica a tarefa 5
4. **Operações via verbos HTTP**: GET, POST, PUT, PATCH, DELETE
5. **Representações**: Dados trafegam como JSON

Exemplo de API RESTful:

```
GET    /api/tarefas/          → Lista todas as tarefas

POST   /api/tarefas/          → Cria nova tarefa

GET    /api/tarefas/5/        → Mostra tarefa 5

PUT    /api/tarefas/5/        → Atualiza tarefa 5 (total)

PATCH  /api/tarefas/5/        → Atualiza tarefa 5 (parcial)

DELETE /api/tarefas/5/        → Deleta tarefa 5
```

Capítulo 2: Setup Profissional do Projeto 🛠️

2.1. Instalação e Estrutura Base

Passo 1: Criar e Ativar Ambiente Virtual

```
# Criar ambiente virtual

python -m venv venv



# Ativar (Linux/Mac)

source venv/bin/activate



# Ativar (Windows)

venv\Scripts\activate
```

Por que usar venv?

- Isola dependências do projeto
- Evita conflitos entre versões de bibliotecas
- Facilita replicação do ambiente

Passo 2: Instalar Dependências

```
pip install django djangorestframework django-environ
djangorestframework-simplejwt
```

Pacotes instalados:

- **django**: Framework web
- **djangorestframework**: Toolkit para APIs REST
- **django-environ**: Gerenciamento de variáveis de ambiente
- **djangorestframework-simplejwt**: Autenticação JWT

Passo 3: Criar Projeto e App

```
# Criar projeto Django

django-admin startproject config .



# Criar app core

python manage.py startapp core
```

Estrutura resultante:

```
meu_projeto/
├── venv/
├── config/
│   ├── __init__.py
│   ├── settings.py      # Configurações do projeto
│   ├── urls.py          # URLs principais
│   └── wsgi.py
├── core/
│   ├── __init__.py
│   ├── models.py        # Models (tabelas)
│   ├── views.py         # Views (lógica)
│   ├── serializers.py   # (vamos criar)
│   └── urls.py          # (vamos criar)
├── manage.py
└── .env                 # (vamos criar)
```

2.2. Segurança: Variáveis de Ambiente

Por que usar .env?

- Nunca exponha SECRET_KEY no código
- Facilita diferentes configurações (dev, produção)
- Protege credenciais sensíveis

Passo 1: Criar arquivo .env

Na raiz do projeto, crie `.env`:

```
SECRET_KEY=django-insecure-sua-chave-super-secreta-aqui-#$%&*()

DEBUG=True
```

Como gerar uma SECRET_KEY segura:

```
# Execute no terminal Python

from django.core.management.utils import get_random_secret_key

print(get_random_secret_key())
```

Passo 2: Configurar settings.py

Edite `config/settings.py`:

```
import os

import environ

from pathlib import Path


# Build paths

BASE_DIR = Path(__file__).resolve().parent.parent


# Inicializar django-environ

env = environ.Env(

    DEBUG=(bool, False)  # Valor padrão caso não esteja no .env

)


# Ler o arquivo .env

environ.Env.read_env(os.path.join(BASE_DIR, '.env'))


# SECURITY WARNING: usar variável de ambiente!

SECRET_KEY = env('SECRET_KEY')


# SECURITY WARNING: não usar DEBUG=True em produção!
```

```
DEBUG = env('DEBUG')


ALLOWED_HOSTS = []


# Application definition

INSTALLED_APPS = [

    'django.contrib.admin',

    'django.contrib.auth',

    'django.contrib.contenttypes',

    'django.contrib.sessions',

    'django.contrib.messages',

    'django.contrib.staticfiles',


    # Third party

    'rest_framework',

    'rest_framework_simplejwt',


    # Local apps

    'core',

]


# ... (resto do arquivo padrão)
```

2.3. Migrações e Usuário Admin

```
# Aplicar migrações iniciais

python manage.py migrate



# Criar superusuário

python manage.py createsuperuser

# Username: admin

# Email: admin@exemplo.com

# Password: (escolha uma senha segura)
```

O que as migrações fazem:

- Criam tabelas no banco de dados
- Sincronizam Models com o schema do banco
- Versionam mudanças no banco

Capítulo 3: O Ciclo M-S-V-U (Model-Serializer-View-URL) 🔄

3.1. Model: A Estrutura dos Dados

O Model define a **estrutura das tabelas** no banco de dados.

Edite `core/models.py`:

```
from django.db import models

from django.contrib.auth.models import User


class Tarefa(models.Model):

    """

    Model para representar uma tarefa de usuário.


    Cada tarefa tem:

    - Um dono (user)

    - Um título

    - Um status de conclusão

    - Data de criação automática

    """


    # ForeignKey: Relacionamento Many-to-One

    # Cada Tarefa pertence a UM usuário

    # Um usuário pode ter VÁRIAS tarefas

    # on_delete=CASCADE: Se o usuário for deletado, suas tarefas também são

    user = models.ForeignKey(

        User,
```

```
    on_delete=models.CASCADE,
    related_name='tarefas',  # Permite user.tarefas.all()
    verbose_name='Usuário'
)

# CharField: Campo de texto com limite
titulo = models.CharField(
    max_length=200,
    verbose_name='Título'
)

# BooleanField: Campo verdadeiro/falso
concluida = models.BooleanField(
    default=False,
    verbose_name='Concluída'
)

# DateTimeField: Data e hora
# auto_now_add=True: Preenche automaticamente na criação
criada_em = models.DateTimeField(
    auto_now_add=True,
    verbose_name='Criada em'
)
```

```
 class Meta:
        verbose_name = 'Tarefa'
        verbose_name_plural = 'Tarefas'
        ordering = ['-criada_em']  # Mais recentes primeiro


    def __str__(self):
        """Representação em string (usado no admin)"""
        return f"{self.titulo} ({'✓' if self.concluida else '✗'})"
```

Tipos de campos comuns:

Campo	Descrição	Exemplo
`CharField`	Texto curto	Título, nome
`TextField`	Texto longo	Descrição, conteúdo
`IntegerField`	Número inteiro	Idade, quantidade
`BooleanField`	Verdadeiro/Falso	Status, ativo
`DateTimeField`	Data e hora	Criado em, atualizado
`ForeignKey`	Relacionamento 1:N	Usuário, categoria
`ManyToManyField`	Relacionamento N:N	Tags, grupos

Criar e aplicar migração:

```
# Criar arquivo de migração
python manage.py makemigrations core


# Aplicar no banco
python manage.py migrate
```

Registrar no Admin (para testes visuais):

Edite `core/admin.py`:

```
from django.contrib import admin

from .models import Tarefa


@admin.register(Tarefa)

class TarefaAdmin(admin.ModelAdmin):

    list_display = ['id', 'titulo', 'user', 'concluida', 'criada_em']

    list_filter = ['concluida', 'criada_em']

    search_fields = ['titulo', 'user__username']
```

3.2. Serializer: O Tradutor JSON ↔ Python

O Serializer é a **ponte** entre objetos Python e JSON.

Crie `core/serializers.py`:

```
from rest_framework import serializers

from .models import Tarefa


class TarefaSerializer(serializers.ModelSerializer):

    """

    Serializer para o Model Tarefa.


    Responsabilidades:

    1. Converter Tarefa → JSON (serialização)

    2. Converter JSON → Tarefa (desserialização)

    3. Validar dados de entrada

    """
```

```
    class Meta:
        model = Tarefa
        fields = ['id', 'user', 'titulo', 'concluida', 'criada_em']

        # Campos gerados automaticamente (não aceitos na entrada)
        read_only_fields = ['id', 'criada_em']
```

Como funciona:

```
# SERIALIZAÇÃO (Python → JSON)
tarefa = Tarefa.objects.get(id=1)
serializer = TarefaSerializer(tarefa)
print(serializer.data)
# Output: {'id': 1, 'user': 2, 'titulo': 'Estudar', 'concluida': False, ...}

# DESSERIALIZAÇÃO (JSON → Python)
data = {'titulo': 'Nova tarefa', 'concluida': False, 'user': 1}
serializer = TarefaSerializer(data=data)
if serializer.is_valid():
    tarefa = serializer.save()  # Cria objeto no banco
```

Validações automáticas do Serializer:

- `max_length` do CharField
- Tipo correto (booleano, inteiro, etc)
- Campos obrigatórios
- ForeignKeys válidas

3.3. View: A Lógica de Negócio

Views processam requisições e retornam respostas.

Edite `core/views.py`:

```
from rest_framework.views import APIView

from rest_framework.response import Response

from rest_framework import status

from .models import Tarefa

from .serializers import TarefaSerializer


class ListaTarefasAPIView(APIView):

    """

    View para listar todas as tarefas (GET).


    Endpoints:

        GET /api/tarefas/ - Lista todas as tarefas

    """


    def get(self, request, format=None):

        """

        Retorna lista de todas as tarefas do banco.


        Returns:

            Response: JSON com lista de tarefas e status 200

        """

        # 1. BUSCAR: ORM do Django busca todos os registros
```

```
        tarefas = Tarefa.objects.all()

        # 2. SERIALIZAR: Converter objetos Python → JSON
        # many=True: indica que é uma lista de objetos
        serializer = TarefaSerializer(tarefas, many=True)

        # 3. RESPONDER: Retornar JSON com status HTTP
        return Response(serializer.data, status=status.HTTP_200_OK)
```

Fluxo de execução:

```
1. Cliente envia: GET /api/tarefas/
         ↓
2. Django roteia para: ListaTarefasAPIView.get()
         ↓
3. View busca no banco: Tarefa.objects.all()
         ↓
4. Serializer converte: [<Tarefa>, <Tarefa>] → JSON
         ↓
5. Response retorna: JSON + status 200
         ↓
6. Cliente recebe: [{"id": 1, ...}, {"id": 2, ...}]
```

3.4. URL: O Mapeamento de Rotas

URLs conectam caminhos HTTP às Views.

Passo 1: Criar URLs do app

Crie `core/urls.py`:

```
from django.urls import path

from .views import ListaTarefasAPIView


# Namespace do app (útil para reverse())

app_name = 'core'


urlpatterns = [

    # /api/tarefas/ → ListaTarefasAPIView

    path('tarefas/', ListaTarefasAPIView.as_view(), name='lista-tarefas'),

]
```

Passo 2: Incluir no projeto

Edite `config/urls.py`:

```
from django.contrib import admin

from django.urls import path, include


urlpatterns = [

    # Admin do Django

    path('admin/', admin.site.urls),


    # URLs do app core (prefixo: /api/)

    path('api/', include('core.urls')),
```

```
]
```

Resultado:

- `http://localhost:8000/api/tarefas/` → `ListaTarefasAPIView`

Capítulo 4: Teste Completo do GET 🧪

4.1. Criar Dados de Teste via Admin

```
# Rodar servidor

python manage.py runserver
```

1. Acesse: `http://localhost:8000/admin/`
2. Faça login com o superusuário
3. Clique em "Tarefas" → "Adicionar Tarefa"
4. Crie 3-5 tarefas de exemplo

4.2. Testar no Navegador (Browsable API)

Acesse: `http://localhost:8000/api/tarefas/`

Você verá a **Browsable API** do DRF:

- Interface HTML interativa
- Lista de tarefas em JSON formatado
- Informações de headers, método, etc

Exemplo de resposta:

```
[

    {

        "id": 1,

        "user": 1,

        "titulo": "Estudar Django REST Framework",

        "concluida": false,

        "criada_em": "2025-11-29T10:30:00Z"

    },
```

```
    {
        "id": 2,
        "user": 1,
        "titulo": "Implementar API de tarefas",
        "concluida": true,
        "criada_em": "2025-11-29T11:45:00Z"
    }
]
```

4.3. Testar com Postman/Insomnia

Configuração da requisição:

```
Método: GET
URL: http://localhost:8000/api/tarefas/
Headers: (vazio por enquanto)
Body: (não necessário no GET)
```

Resultado esperado:

- Status: `200 OK`
- Body: JSON com lista de tarefas

4.4. Testar com cURL (linha de comando)

```
curl http://localhost:8000/api/tarefas/
```

Capítulo 5: Conceitos Avançados e Troubleshooting 🔧

5.1. Status Codes HTTP Importantes

Código	Nome	Significado	Quando usar
200	OK	Sucesso genérico	GET, PUT, PATCH bem-sucedidos
201	Created	Recurso criado	POST bem-sucedido
204	No Content	Sucesso sem corpo	DELETE bem-sucedido
400	Bad Request	Dados inválidos	Validação falhou
401	Unauthorized	Não autenticado	Token ausente/inválido
403	Forbidden	Sem permissão	Usuário não é dono do recurso
404	Not Found	Recurso não existe	ID inexistente
500	Server Error	Erro no servidor	Bug no código

5.2. Debugging Comum

Problema: "ModuleNotFoundError: No module named 'rest_framework'"

```
# Solução: Instalar DRF

pip install djangorestframework



# Adicionar em INSTALLED_APPS

'rest_framework',
```

Problema: "Table doesn't exist"

```
# Solução: Executar migrações

python manage.py makemigrations

python manage.py migrate
```

Problema: "CSRF verification failed"

Para APIs, desabilite CSRF na View:

```
from django.views.decorators.csrf import csrf_exempt

from django.utils.decorators import method_decorator


@method_decorator(csrf_exempt, name='dispatch')

class ListaTarefasAPIView(APIView):

    # ...
```

Ou use autenticação JWT (veremos na Apostila 4).

5.3. Boas Práticas de Código

```
# ❌ EVITE: Código repetido

def get(self, request):

    tarefas = Tarefa.objects.all()

    data = []

    for tarefa in tarefas:

        data.append({

            'id': tarefa.id,

            'titulo': tarefa.titulo,

            # ...

        })

    return Response(data)


# ✅ PREFIRA: Use Serializers

def get(self, request):

    tarefas = Tarefa.objects.all()

    serializer = TarefaSerializer(tarefas, many=True)

    return Response(serializer.data)
```

📝 Exercícios Práticos

Exercício 1: Adicionar Campo ao Model

Adicione um campo `descricao` (TextField) ao Model Tarefa:

```
descricao = models.TextField(blank=True, null=True)
```

1. Crie a migração
2. Aplique no banco
3. Adicione ao Serializer
4. Teste criando uma tarefa com descrição no admin

Exercício 2: Criar View de Contagem

Crie uma view que retorne a contagem de tarefas:

```
class ContagemTarefasAPIView(APIView):

    def get(self, request):

        total = Tarefa.objects.count()

        concluidas = Tarefa.objects.filter(concluida=True).count()

        pendentes = total - concluidas



        return Response({

            'total': total,

            'concluidas': concluidas,

            'pendentes': pendentes

        })
```

Mapeie em `/api/tarefas/contagem/`.

Exercício 3: Filtrar por Usuário

Modifique o GET para retornar apenas tarefas de um usuário específico:

```
def get(self, request):

    user_id = request.query_params.get('user_id')

    if user_id:

        tarefas = Tarefa.objects.filter(user_id=user_id)

    else:

        tarefas = Tarefa.objects.all()



    serializer = TarefaSerializer(tarefas, many=True)

    return Response(serializer.data)
```

Teste: `http://localhost:8000/api/tarefas/?user_id=1`

🎓 Resumo da Apostila

O que aprendemos:

1. ✅ Diferença entre MVT e REST API
2. ✅ Verbos HTTP e seus propósitos
3. ✅ Setup profissional com django-environ
4. ✅ Segurança com variáveis de ambiente
5. ✅ Ciclo completo: Model → Serializer → View → URL
6. ✅ Primeira View funcional (GET)
7. ✅ Testes com Browsable API, Postman e cURL

Próximos passos (Apostila 2):

- Implementar POST (criar tarefas)
- Validação de dados com Serializers
- Tratamento de erros
- Status codes adequados

📚 Referências

- Django REST Framework - Official Docs
- HTTP Methods - MDN
- REST API Best Practices
- Django Models Reference