

# Apostila 6: Testes Automatizados em Django REST Framework

---

## 🎯 Objetivos de Aprendizagem

Ao final desta apostila, você terá domínio sobre:

1. **Fundamentos:** Compreender a importância crítica de testes automatizados em APIs profissionais.
  2. **Tipos de Teste:** Diferenciar e aplicar testes unitários, de integração e de sistema.
  3. **Ferramentas:** Dominar o APITestCase do Django REST Framework.
  4. **Massa de Dados:** Criar fixtures eficientes e dados de teste dinâmicos com *Factories*.
  5. **Segurança:** Testar fluxos complexos de autenticação JWT e permissões (RBAC).
  6. **Qualidade:** Medir a cobertura de código (*Code Coverage*) para garantir robustez.
  7. **Metodologia:** Aplicar o ciclo TDD (*Test-Driven Development*) no desenvolvimento de APIs.
- 

## Capítulo 1: Por Que Testar? 🤔

### 1.1. O Custo de Não Testar

Imagine o seguinte cenário real em uma sexta-feira à tarde:

O Cenário:

Um desenvolvedor precisa corrigir um filtro na listagem de tarefas e faz esta alteração aparentemente "inocente":

```
# Um desenvolvedor faz esta alteração:  
class TarefaListCreateAPIView(generics.ListCreateAPIView):  
    # ANTES: queryset = Tarefa.objects.filter(user=self.request.user)  
    # DEPOIS (O Erro):  
    queryset = Tarefa.objects.all() # Mudou de filter() para all() por engano  
    # ...
```

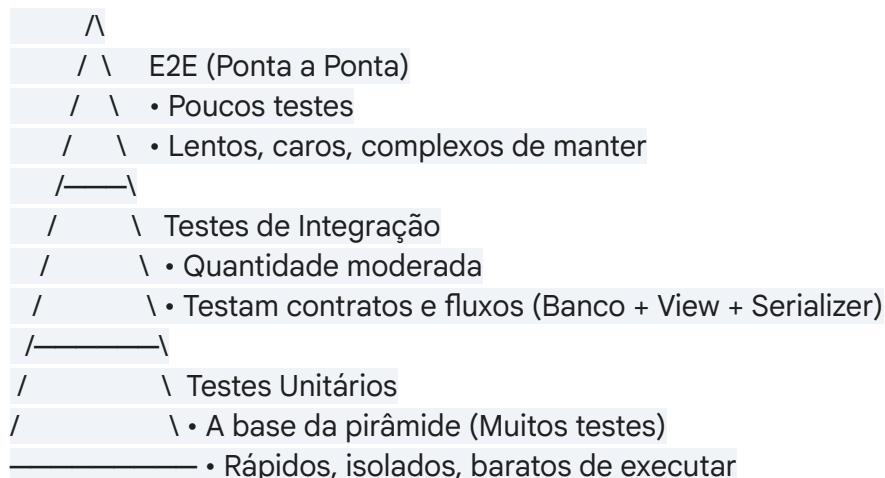
- **Consequência:** A API perde o isolamento de dados. Agora, todos os usuários conseguem ver as tarefas privadas de todos os outros usuários. É um vazamento de dados grave (LGPD).
- **Detecção:**

- **X Sem testes:** O erro vai para produção. É descoberto apenas quando clientes furiosos começam a reclamar ou quando dados vazam. O custo de correção é altíssimo (imagem da empresa, processos legais).
- **✓ Com testes:** O sistema de CI/CD (Integração Contínua) roda python manage.py test. O teste test\_usuario\_ve\_apenas\_suas\_tarefas FALHA. O deploy é bloqueado automaticamente. O erro é corrigido em 2 minutos.

## 1.2. A Pirâmide de Testes

Para garantir qualidade sem sacrificar a velocidade, seguimos a Pirâmide de Testes:

Plaintext



### Aplicação em APIs REST:

- **Unitários:** Testam componentes isolados sem depender de banco de dados ou rede externa (Ex: validação de um campo no Serializer, lógica de um método no Model).
- **Integração:** Testam se as peças funcionam juntas (Ex: Requisição POST → Salva no Banco → Retorna JSON correto). No Django, o APITestCase geralmente cai aqui.
- **E2E:** Simulam um cliente real (Ex: Selenium ou Cypress chamando a API).

## 1.3. O Manifesto do Código Testável

1. **✓ Código testado é código confiável:** Você dorme tranquilo após o deploy.
2. **✓ Testes são documentação executável:** Eles mostram exatamente como o código deve se comportar, sem ambiguidade.
3. **✓ Refatoração sem testes é risco:** Se você não tem testes, você tem

- medo de melhorar o código antigo.
4.  **Bug encontrado = Teste criado:** Antes de corrigir um bug, crie um teste que reproduza esse bug.
- 

## Capítulo 2: Configuração do Ambiente de Testes

### 2.1. Estrutura de Arquivos

Organização é fundamental. O Django descobre testes automaticamente em qualquer arquivo que comece com `test_`.

**Estrutura Recomendada:**

```
core/
└── tests/      # Pacote Python (pasta com __init__.py)
    ├── __init__.py  # Necessário para ser um pacote
    ├── test_models.py  # Testa lógica de negócio e banco
    ├── test_serializers.py # Testa entrada/saída e validação
    ├── test_views.py   # Testa endpoints (Integração)
    └── test_permissions.py # Testa regras de acesso
└── models.py
└── views.py
└── ...
```

**Comandos para criar a estrutura:**

```
mkdir core/tests
touch core/tests/__init__.py
touch core/tests/test_views.py
touch core/tests/test_models.py
touch core/tests/test_serializers.py
```

### 2.2. Configuração em `settings.py`

Por padrão, o Django cria um banco de dados vazio a cada execução de testes. Podemos otimizar isso usando SQLite em memória, que é muito mais rápido que escrever no disco.

No arquivo `config/settings.py`, adicione:

```
import sys

# ... configurações existentes ...

# Configuração para acelerar testes
# Se o comando 'test' estiver sendo executado, trocamos o banco para SQLite em memória
if 'test' in sys.argv:
    DATABASES['default'] = {
        'ENGINE': 'django.db.backends.sqlite3',
        'NAME': ':memory:', # Banco reside apenas na RAM
    }
```

⚠️ **Atenção:** Embora rápido, o SQLite pode se comportar de forma ligeiramente diferente do PostgreSQL (ex: restrições de integridade, tratamento de datas). Em ambientes de CI/CD (Integração Contínua) profissional, recomenda-se usar o mesmo banco da produção (Dockerizado).

## 2.3. Primeiro Teste (Sanity Check)

Vamos garantir que o *test runner* do Django está funcionando.

Edite core/tests/test\_views.py:

```
# core/tests/test_views.py
from django.test import TestCase

class SanityTestCase(TestCase):
    """Testa se o ambiente de testes está configurado corretamente."""

    def test_verdade_universal(self):
        """O mais simples dos testes: verifica se True é True."""
        self.assertTrue(True)
```

**Execute:**

```
python manage.py test
```

**Saída esperada:**

```
Creating test database for alias 'default'...
System check identified no issues (0 silenced).
.
-----
Ran 1 test in 0.001s
```

```
OK
Destroying test database for alias 'default'...
```

O ponto . indica um teste que passou.

---

## Capítulo 3: Testando Models

Testamos models para garantir que a estrutura do banco, métodos personalizados (`__str__`, propriedades) e regras de negócio funcionam.

### 3.1. Teste de Criação e Validação

Edite `core/tests/test_models.py`:

```
# core/tests/test_models.py
from django.test import TestCase
from django.contrib.auth.models import User
from django.db import IntegrityError
from core.models import Tarefa

class TarefaModelTest(TestCase):
    """Testes focados nas regras de negócio do Model Tarefa."""

    def setUp(self):
        ....
```

Método executado ANTES de CADA teste.  
Ideal para criar o estado inicial necessário.

```
    """
    self.user = User.objects.create_user(
        username='testuser',
        password='testpass123'
    )

    def test_criacao_tarefa_sucesso(self):
        """Testa se uma tarefa é criada corretamente com os campos padrão."""
        tarefa = Tarefa.objects.create(
            user=self.user,
            titulo='Tarefa de Teste',
            concluida=False
        )

        # Assertions (Verificações)
        self.assertEqual(tarefa.titulo, 'Tarefa de Teste')
        self.assertFalse(tarefa.concluida)
        self.assertEqual(tarefa.user, self.user)
        # Verifica se o auto_now_add funcionou
        self.assertIsNotNone(tarefa.criada_em)

    def test_str_representation(self):
        """
        Testa o método __str__.
        Importante pois é como o objeto aparece no Admin do Django.
        """
        tarefa = Tarefa.objects.create(
            user=self.user,
            titulo='Estudar Testes',
            concluida=True
        )

        # Supondo que seu __str__ seja: f'{self.titulo} ({'✓' if self.concluida else '✗'})'
        esperado = 'Estudar Testes (✓)'
        self.assertEqual(str(tarefa), esperado)

    def test_user_eh_obrigatorio(self):
        """Testa a integridade do banco: user não pode ser null."""
        with self.assertRaises(IntegrityError):
            Tarefa.objects.create(
                titulo='Sem dono',
                concluida=False
            )
            # user ausente propositalmente
```

)

Para rodar apenas este arquivo:

```
python manage.py test core.tests.test_models
```

## Capítulo 4: Testando Serializers

Serializers são a "porta de entrada" dos dados. Testamos se eles validam o que entra (input) e formatam corretamente o que sai (output).

### 4.1. Validação de Dados

Edite core/tests/test\_serializers.py:

```
# core/tests/test_serializers.py
from django.test import TestCase
from django.contrib.auth.models import User
from core.models import Tarefa
from core.serializers import TarefaSerializer

class TarefaSerializerTest(TestCase):

    def setUp(self):
        self.user = User.objects.create_user(
            username='testuser',
            password='testpass123'
        )

        self.tarefa_data = {
            'titulo': 'Tarefa Serializada',
            'concluida': False
            # Note que 'user' não entra aqui, pois geralmente vem do request context
        }

    def test_serializer_com_dados_validos(self):
        """Testa se o serializer aceita dados corretos."""
        serializer = TarefaSerializer(data=self.tarefa_data)
```

```
    self.assertTrue(serializer.is_valid())
    self.assertEqual(serializer.validated_data['titulo'], 'Tarefa Serializada')

def test_serializer_sem_titulo(self):
    """Testa se o campo obrigatório 'titulo' é validado."""
    dados_invalidos = {'concluida': False}
    serializer = TarefaSerializer(data=dados_invalidos)

    self.assertFalse(serializer.is_valid())
    # Verifica se a chave do erro é exatamente 'titulo'
    self.assertIn('titulo', serializer.errors)

def test_serializer_titulo_muito_longo(self):
    """Testa validação de max_length (ex: 200 caracteres)."""
    dados_invalidos = {
        'titulo': 'a' * 201, # Cria string com 201 caracteres
        'concluida': False
    }
    serializer = TarefaSerializer(data=dados_invalidos)

    self.assertFalse(serializer.is_valid())
    self.assertIn('titulo', serializer.errors)

def test_deserializacao_output(self):
    """Testa se a conversão de Objeto Python -> JSON está correta."""
    tarefa = Tarefa.objects.create(
        user=self.user,
        titulo='Tarefa Existente',
        concluida=True
    )

    serializer = TarefaSerializer(tarefa)
    data = serializer.data

    self.assertEqual(data['titulo'], 'Tarefa Existente')
    self.assertTrue(data['concluida'])
    self.assertEqual(data['id'], tarefa.id)
```

# Capítulo 5: Testando Endpoints (APITestCase)

Aqui reside o poder do DRF. Usamos APITestCase para simular requisições HTTP reais.

## 5.1. Configuração Básica

Edite core/tests/test\_views.py:

```
# core/tests/test_views.py
from rest_framework.test import APITestCase
from rest_framework import status
from django.contrib.auth.models import User, Group
from django.urls import reverse
from core.models import Tarefa

class TarefaAPITest(APITestCase):
    """
    Testes de Integração para os endpoints de Tarefa.
    Simula o ciclo completo: Request -> URL -> View -> Serializer -> DB -> Response
    """

    def setUp(self):
        """Prepara a massa de dados para os testes."""
        # 1. Criar usuários
        self.user1 = User.objects.create_user(username='usuario1', password='senha123')
        self.user2 = User.objects.create_user(username='usuario2', password='senha456')

        # 2. Criar e atribuir grupos (se sua lógica depender disso)
        self.grupo_comum = Group.objects.create(name='Comum')
        self.user1.groups.add(self.grupo_comum)

        # 3. Criar tarefas iniciais
        self.tarefa_user1 = Tarefa.objects.create(
            user=self.user1, titulo='Tarefa do User 1', concluida=False
        )

        self.tarefa_user2 = Tarefa.objects.create(
            user=self.user2, titulo='Tarefa do User 2', concluida=True
        )

    # ----- Helpers -----
```

```

def obter_token(self, username, password):
    """
    Método auxiliar para realizar login e obter o token JWT.
    Evita repetição de código nos testes.
    """

    url = reverse('token_obtain_pair') # Nome da rota definida no urls.py
    response = self.client.post(url, {
        'username': username,
        'password': password
    }, format='json')

    return response.data['access']

```

## 5.2. Teste de Autenticação JWT

```

# ----- Testes de Auth -----

def test_acesso_sem_token_negado(self):
    """Testa que endpoint protegido retorna 401 para anônimos."""
    url = reverse('tarefas-list')
    response = self.client.get(url)

    self.assertEqual(response.status_code, status.HTTP_401_UNAUTHORIZED)

def test_login_credenciais_validas(self):
    """Testa se login correto retorna tokens."""
    url = reverse('token_obtain_pair')
    response = self.client.post(url, {
        'username': 'usuario1',
        'password': 'senha123'
    }, format='json')

    self.assertEqual(response.status_code, status.HTTP_200_OK)
    self.assertIn('access', response.data)
    self.assertIn('refresh', response.data)

def test_login_credenciais_invalidadas(self):
    """Testa rejeição de login com senha errada."""
    url = reverse('token_obtain_pair')
    response = self.client.post(url, {
        'username': 'usuario1',
        'password': 'senhaerrada'
    }, format='json')

    self.assertEqual(response.status_code, status.HTTP_401_UNAUTHORIZED)
    self.assertIn('detail', response.data)

```

```
    }, format='json')

    self.assertEqual(response.status_code, status.HTTP_401_UNAUTHORIZED)
```

### 5.3. Teste de CRUD Completo

```
# ----- Testes de CRUD -----

def test_listar_tarefas_apenas_do_usuario_logado(self):
    """
    Testa o isolamento de dados: User1 só vê suas próprias tarefas.
    """

    # Autentica como User1
    token = self.obter_token('usuario1', 'senha123')
    self.client.credentials(HTTP_AUTHORIZATION=f'Bearer {token}')

    url = reverse('tarefas-list')
    response = self.client.get(url)

    self.assertEqual(response.status_code, status.HTTP_200_OK)
    self.assertEqual(len(response.data), 1) # User1 tem 1 tarefa, User2 tem outra
    self.assertEqual(response.data[0]['titulo'], 'Tarefa do User 1')


def test_criar_tarefa(self):
    """
    Testa criação (POST) com injeção automática do usuário logado.
    """

    token = self.obter_token('usuario1', 'senha123')
    self.client.credentials(HTTP_AUTHORIZATION=f'Bearer {token}')

    url = reverse('tarefas-list')
    dados = {
        'titulo': 'Nova Tarefa via API',
        'concluida': False
    }

    # format='json' é crucial para APIs REST
    response = self.client.post(url, dados, format='json')

    self.assertEqual(response.status_code, status.HTTP_201_CREATED)
    self.assertEqual(response.data['titulo'], 'Nova Tarefa via API')
```

```
# Verificação extra: Confere se salvou no banco corretamente
tarefa_criada = Tarefa.objects.get(id=response.data['id'])
self.assertEqual(tarefa_criada.user, self.user1)

def test_atualizar_tarefa_propria(self):
    """Testa atualização parcial (PATCH) em tarefa própria."""
    token = self.obter_token('usuario1', 'senha123')
    self.client.credentials(HTTP_AUTHORIZATION=f'Bearer {token}')

    url = reverse('tarefas-detail', kwargs={'pk': self.tarefa_user1.pk})
    dados = {'concluida': True}

    response = self.client.patch(url, dados, format='json')

    self.assertEqual(response.status_code, status.HTTP_200_OK)
    self.assertTrue(response.data['concluida'])

def test_nao_pode_acessar_tarefa_de_outro_usuario(self):
    """
    Segurança: Tenta acessar via ID uma tarefa que pertence a outro user.
    """

    token = self.obter_token('usuario1', 'senha123')
    self.client.credentials(HTTP_AUTHORIZATION=f'Bearer {token}')

    # Tenta acessar tarefa do user2
    url = reverse('tarefas-detail', kwargs={'pk': self.tarefa_user2.pk})
    response = self.client.get(url)

    # O ideal é 404 Not Found (para não revelar que o ID existe)
    self.assertEqual(response.status_code, status.HTTP_404_NOT_FOUND)
```

## 5.4. Teste de Permissões (RBAC)

Testando regras de negócio baseadas em Cargos (Roles).

```
def test_usuario_comum_nao_pode_deletar(self):
    """Testa que usuário sem permissão específica recebe 403 Forbidden."""
    token = self.obter_token('usuario1', 'senha123')
    self.client.credentials(HTTP_AUTHORIZATION=f'Bearer {token}')

    url = reverse('tarefas-detail', kwargs={'pk': self.tarefa_user1.pk})
    response = self.client.delete(url)

    self.assertEqual(response.status_code, status.HTTP_403_FORBIDDEN)

def test_gerente_pode_deletar(self):
    """Testa que usuário com cargo 'Gerente' consegue deletar."""
    # Setup específico deste teste: Dar poder de Gerente
    grupo_gerente = Group.objects.create(name='Gerente')
    self.user1.groups.add(grupo_gerente)

    token = self.obter_token('usuario1', 'senha123')
    self.client.credentials(HTTP_AUTHORIZATION=f'Bearer {token}')

    url = reverse('tarefas-detail', kwargs={'pk': self.tarefa_user1.pk})
    response = self.client.delete(url)

    self.assertEqual(response.status_code, status.HTTP_204_NO_CONTENT)

    # Verifica se sumiu do banco
    self.assertFalse(Tarefa.objects.filter(pk=self.tarefa_user1.pk).exists())
```

## Capítulo 6: Testando Cadastro de Usuários

Continuando em core/tests/test\_views.py ou criando um novo arquivo:

Python

```
class RegisterAPITest(APITestCase):
    """Testes específicos para o endpoint de registro público."""

    def setUp(self):
        # Necessário pois a View de registro atribui este grupo
        self.grupo_comum = Group.objects.create(name='Comum')

    def test_cadastro_usuario_sucesso(self):
        """Testa o fluxo feliz de cadastro."""
        url = reverse('register')
        dados = {
            'username': 'novousuario',
            'email': 'novo@email.com',
            'password': 'senha_segura_123'
        }

        response = self.client.post(url, dados, format='json')

        self.assertEqual(response.status_code, status.HTTP_201_CREATED)
        self.assertIn('username', response.data)
        #💡 Segurança: A senha NUNCA deve ser retornada na resposta
        self.assertNotIn('password', response.data)

        # Verifica criação no banco
        user = User.objects.get(username='novousuario')
        # Verifica se a senha foi hasheada (não está em texto plano)
        self.assertTrue(user.check_password('senha_segura_123'))
        # Verifica atribuição de grupo default
        self.assertTrue(user.groups.filter(name='Comum').exists())

    def test_cadastro_username_duplicado(self):
        """Testa a validação de unicidade do banco."""
        User.objects.create_user(username='existente', password='123')
```

```

url = reverse('register')
dados = {
    'username': 'existente', # Username já em uso
    'email': 'outro@email.com',
    'password': 'senha123'
}

response = self.client.post(url, dados, format='json')

self.assertEqual(response.status_code, status.HTTP_400_BAD_REQUEST)
self.assertIn('username', response.data)

```

## Capítulo 7: Cobertura de Código (Coverage)

A métrica de "cobertura" indica qual porcentagem do seu código foi executada pelos testes.

### 7.1. Instalação

```
pip install coverage
```

### 7.2. Executando com Coverage

Vamos rodar os testes monitorando a execução:

```
# Rodar testes apontando a 'source' para a pasta atual (.) e testando o app 'core'
coverage run --source='.' manage.py test core
```

```
# Gerar relatório simples no terminal
coverage report
```

**Saída esperada (Exemplo):**

Name	Stmts	Miss	Cover
<hr/>			
core/models.py	25	0	100%
core/serializers.py	18	2	89%
core/views.py	45	5	89%
core/permissions.py	12	1	92%
<hr/>			
TOTAL	100	8	92%

## 7.3. Visualização HTML

Para entender *quais* linhas não foram testadas:

```
# Gera uma pasta 'htmlcov'  
coverage html
```

Abra htmlcov/index.html no seu navegador. Você verá:

-  **Verde:** Código testado.
-  **Vermelho:** Código que seus testes nunca tocaram (Cuidado aqui!).
- **Meta de Mercado:** Tente manter acima de **80%**.

---

# Capítulo 8: Boas Práticas e Patterns

## 8.1. Princípio AAA (Arrange-Act-Assert)

Todo teste deve ter três fases claras:

```
def test_exemplo_aaa(self):  
    # 1. ARRANGE (Preparar): Setup de dados e cenário  
    user = User.objects.create_user(username='test', password='123')  
  
    # 2. ACT (Agir): Executar a ação que queremos testar  
    tarefa = Tarefa.objects.create(user=user, titulo='Teste')  
  
    # 3. ASSERT (Verificar): Validar se o resultado é o esperado  
    self.assertEqual(tarefa.titulo, 'Teste')
```

## 8.2. Factories (Padrão de Projeto)

Criar objetos manualmente (User.objects.create...) torna-se cansativo em projetos grandes.  
Use factory\_boy para criar dados falsos inteligentes.

## Instalação:

```
pip install factory-boy
```

## Criando Factories (core/tests/factories.py):

```
import factory
from django.contrib.auth.models import User
from core.models import Tarefa

class UserFactory(factory.django.DjangoModelFactory):
    class Meta:
        model = User

    # Cria users únicos: user0, user1, user2...
    username = factory.Sequence(lambda n: f'user{n}')
    # Cria e-mails baseados no username
    email = factory.LazyAttribute(lambda obj: f'{obj.username}@test.com')

class TarefaFactory(factory.django.DjangoModelFactory):
    class Meta:
        model = Tarefa

    # Cria automaticamente um User para esta tarefa
    user = factory.SubFactory(UserFactory)
    # Gera uma frase aleatória
    titulo = factory.Faker('sentence', nb_words=4)
    concluida = False
```

## Uso nos testes:

```
# Em vez de criar manualmente...
# user = User.objects.create_user(...)

# Use factories:
tarefa = TarefaFactory() # Cria user e tarefa magicamente
usuarios = UserFactory.create_batch(10) # Cria 10 usuários de uma vez
```

### 8.3. Testes Parametrizados (SubTests)

Para testar várias entradas sem duplicar código, use subTest:

Python

```
from django.test import TestCase

class ValidacaoTituloTest(TestCase):

    def test_titulos_invalidos_variados(self):
        """Testa múltiplos casos de borda em um único método."""
        titulos_invalidos = ['', ' ', 'a', '123'] # Lista de inputs ruins

        for titulo in titulos_invalidos:
            # O subTest isola a falha. Se um falhar, os outros continuam rodando.
            with self.subTest(titulo=titulo):
                serializer = TarefaSerializer(data={'titulo': titulo})
                self.assertFalse(serializer.is_valid())
```

### 8.4. setUp vs setUpTestData

- `setUp()`: Roda antes de **CADA** método de teste. Bom para garantir isolamento total, mas lento se criar muitos dados.
- `setUpTestData()`: Roda **UMA VEZ** por classe. Dados são criados no início e revertidos no final da classe. Muito mais rápido para dados de leitura.

```
class PerformanceTest(TestCase):

    @classmethod
    def setUpTestData(cls):
        """
        Executado UMA VEZ. Ótimo para criar usuários e configurações estáticas.
        """

        cls.user = User.objects.create_user(username='test', password='123')

    def setUp(self):
        """
        Executado VÁRIAS VEZES. Use para dados que serão alterados/deletados.
        """
```

```
    self.tarefa = Tarefa.objects.create(  
        user=self.user,  
        titulo='Teste Volátil'  
    )
```

## Capítulo 9: TDD (Test-Driven Development)

TDD é uma disciplina de design. Você escreve o teste *antes* do código funcional.

### 9.1. O Ciclo Red-Green-Refactor

1.  **RED (Falha):** Escreva um teste para uma funcionalidade que ainda não existe. Ele DEVE falhar.
2.  **GREEN (Passa):** Escreva o código MÍNIMO necessário para fazer o teste passar. Não se preocupe com beleza agora.
3.  **REFACTOR (Refatora):** Melhore o código (limpeza, performance) mantendo os testes verdes.

### 9.2. Exemplo Prático: Campo "prioridade"

#### Passo 1: Escrever o teste (que falhará)

No core/tests/test\_models.py:

```
def test_tarefa_com_prioridade(self):  
    """Requisito: Tarefa deve ter campo prioridade."""  
    tarefa = Tarefa.objects.create(  
        user=self.user,  
        titulo='Urgente',  
        prioridade='alta' # Campo ainda não existe!  
    )  
    self.assertEqual(tarefa.prioridade, 'alta')
```

Execute python manage.py test.

Resultado:  TypeError: 'prioridade' is an invalid keyword argument.

#### Passo 2: Implementar o mínimo

No core/models.py:

```
class Tarefa(models.Model):
    # ... campos existentes

    PRIORIDADE_CHOICES = [
        ('baixa', 'Baixa'),
        ('media', 'Média'),
        ('alta', 'Alta'),
    ]

    prioridade = models.CharField(
        max_length=10,
        choices=PRIORIDADE_CHOICES,
        default='media'
    )
```

Rode as migrações e o teste novamente.

Resultado: OK.

Passo 3: Refatorar

Agora que temos certeza que funciona, podemos adicionar a lógica no Serializer, criar filtros por prioridade, etc.

---



## Exercícios Práticos

### Exercício 1: Teste de Validação Customizada (TDD)

No Serializer, implemente a seguinte regra de negócio: "Tarefas com prioridade 'alta' não podem ser criadas já concluídas".

**Requisitos:**

1. Escreva o teste (em test\_serializers.py) **ANTES** de codificar.
2. O teste deve passar dados: {"titulo": "Urgente", "prioridade": "alta", "concluida": true}.
3. O teste deve verificar que serializer.is\_valid() é False.
4. Implemente o método validate() no Serializer para fazer o teste passar.

## Exercício 2: Teste de Endpoint de Estatísticas

Crie e teste um novo endpoint /api/tarefas/estatisticas/ que retorna um JSON customizado:

```
{  
    "total": 10,  
    "concluidas": 6,  
    "pendentes": 4  
}
```

### Requisitos:

1. Crie 10 tarefas no setUp (use TarefaFactory se quiser).
2. Garanta que o endpoint retorna 401 Unauthorized se não enviar token.
3. Garanta que o endpoint retorna 200 OK com token.
4. Verifique se os números retornados batem com os dados criados.
5. **Desafio:** O endpoint deve contar apenas tarefas do usuário logado.

## Exercício 3: Teste de Performance (SLA)

Implemente um teste que garante que a listagem de 1000 tarefas não demore mais que 2 segundos.

### Dicas:

- Use TarefaFactory.create\_batch(1000) no setUpTestData.
- Use a biblioteca time para medir:  

```
Python  
start = time.time()  
self.client.get(url)  
end = time.time()  
self.assertLess(end - start, 2.0)
```
- Se falhar, investigue o problema de N+1 queries e use select\_related('user') na View.

## Checklist de Verificação

Antes de prosseguir para a Apostila 7, garanta que:

- [ ] Todos os testes passam (python manage.py test).
  - [ ] Cobertura de código está acima de 80% (coverage report).
  - [ ] Testou fluxos de sucesso E de erro (ex: login errado).
  - [ ] Testou isolamento de dados (Usuário A não vê dados de Usuário B).
  - [ ] Testou permissões (Usuário Comum não deleta tarefas).
  - [ ] Compreendeu o ciclo TDD (Red-Green-Refactor).
  - [ ] Criou pelo menos um teste usando Factory Boy.
- 

## Resumo da Apostila

O que aprendemos:

-  **Tipos de Testes:** A diferença prática entre testar uma função (Unitário) e um endpoint completo (Integração).
  -  **APITestCase:** A ferramenta poderosa do DRF para simular clientes HTTP.
  -  **Autenticação:** Como injetar tokens JWT nos testes usando credentials().
  -  **Fixtures e Factories:** Estratégias para povoar o banco de dados antes dos testes.
  -  **Assertions:** O vocabulário dos testes (assertEqual, assertTrue, assertIn).
  -  **Coverage:** Como gerar relatórios visuais de qualidade de código.
  -  **TDD:** A filosofia de escrever o teste antes da funcionalidade.
- 

## Referências

- [Django Testing Documentation](#)
  - [DRF Testing Guide](#)
  - [Coverage.py Documentation](#)
  - [Factory Boy Documentation](#)
  - Livro: "Test-Driven Development with Python" (Harry Percival)
- 

## Parabéns!

Sua API agora tem um "colete à prova de balas". Qualquer bug introduzido no futuro será detectado imediatamente pelo comando test. Você agora tem liberdade para refatorar e melhorar seu código sem medo!

**Próxima Apostila (7):** Migraremos do **SQLite** para **PostgreSQL**, preparando a aplicação para o mundo real. Aprenderemos sobre:

- Docker Compose para banco de dados.
  - Variáveis de ambiente (python-decouple).
  - Backup e Restore.
  - Performance e Indexação.
- 

## Dica Final: Git Hook

Para nunca esquecer de rodar os testes, automatize isso no Git:

1. Crie o arquivo `.githooks/pre-commit`:

Bash

```
#!/bin/bash
echo "Rodeando testes antes do commit..."
python manage.py test
if [ $? -ne 0 ]; then
    echo "✗ Testes falharam! Commit bloqueado."
    exit 1
fi
```

2. Ative o hook:

Bash

```
git config --local core.hooksPath .githooks/
chmod +x .githooks/pre-commit
```

Agora, se o código estiver quebrado, o Git não deixará você fazer commit!