

Django APIテスト開発 学習ガイド 🚀

はじめに

このガイドでは、Django REST FrameworkのAPIテストを初心者でも理解できるように、ステップバイステップで解説します。

📖 学習の流れ

STEP 1: Djangoテストの基本概念

テストとは？

- コードが期待通りに動作するかを自動的に検証
- バグを早期発見し、品質を向上
- リファクタリング時の安全性を確保

Django TestCaseの基本

```
python

from django.test import TestCase

class MyTestCase(TestCase):
    def test_something(self):
        # テストコード
        self.assertEqual(expected, actual)
```

重要なメソッド

```
python

# セットアップ (テスト前の準備)
def setUp(self):          # 各テストメソッド実行前
    pass

@classmethod
def setUpTestData(cls):    # テストクラス実行前 (1回のみ)
    pass

# アサーション (検証)
self.assertEqual(a, b)     # a == b
self.assertTrue(condition) # condition が True
self.assertIn(item, list)  # item が list に含まれる
```

STEP 2: APIテストの構造理解

基本的なAPIテストパターン

python

```
def test_api_normal_case(self):  
    # 1. テストデータ準備  
    data = {'key': 'value'}  
  
    # 2. API呼び出し  
    response = self.client.post('/api/endpoint/', data)  
  
    # 3. 結果検証  
    self.assertEqual(200, response.status_code)  
    self.assertEqual(expected_data, response.json())
```

Django Test Client

python

```
from django.test import Client  
  
# GETリクエスト  
response = self.client.get('/api/users/')  
  
# POSTリクエスト  
response = self.client.post('/api/users/', {  
    'name': 'John',  
    'email': 'john@example.com'  
})  
  
# レスponse確認  
print(response.status_code) # 200, 404, 500など  
print(response.json())      # JSONレスponse  
print(response.content)     # バイト形式の内容
```

STEP 3: テストデータの管理

setUpTestDataの使い方

python

```
class APITestCase(TestCase):
    @classmethod
    def setUpTestData(cls):
        """テストクラス全体で使うデータを作成"""
        # ユーザー作成
        cls.user = User.objects.create(
            username='testuser',
            email='test@example.com'
        )

        # 商品作成
        cls.product = Product.objects.create(
            name='テスト商品',
            price=1000
        )

    def test_get_product(self):
        """作成したテストデータを使用"""
        response = self.client.get(f'/api/products/{self.product.id}/')
        self.assertEqual(200, response.status_code)
```

テストデータのパターン

python

正常なデータ

```
valid_data = {  
    'name': '山田太郎',  
    'age': 30,  
    'email': 'yamada@example.com'  
}
```

異常なデータ

```
invalid_data = {  
    'name': '',          # 必須項目が空  
    'age': -1,          # 不正な値  
    'email': 'invalid'  # 不正な形式  
}
```

境界値データ

```
boundary_data = {  
    'age': 0,           # 最小値  
    'age': 150,        # 最大値  
    'name': 'a' * 100, # 最大長  
}
```

STEP 4: 正常系テストの書き方

基本的な正常系テスト

python

```
def test_create_user_success(self):
    """ユーザー作成が正常に行われること"""
    # テストデータ準備
    user_data = {
        'username': 'newuser',
        'email': 'newuser@example.com',
        'password': 'securepassword123'
    }

    # API呼び出し
    response = self.client.post('/api/users/', user_data)

    # レスポンス検証
    self.assertEqual(201, response.status_code)

    # レスポンス内容検証
    response_data = response.json()
    self.assertEqual('newuser', response_data['username'])
    self.assertEqual('newuser@example.com', response_data['email'])

    # データベース検証
    user = User.objects.get(username='newuser')
    self.assertEqual('newuser@example.com', user.email)
```

複数パターンのテスト

python

```
def test_get_agency_by_code(self):
    """代理店コードで代理店情報を取得"""
    test_cases = [
        ('0001', 'A代理店'),
        ('0002', 'B代理店'),
        ('0003', 'C代理店'),
    ]

    for agency_code, expected_name in test_cases:
        with self.subTest(agency_code=agency_code):
            response = self.client.post('/api/agency/', {
                'agency_code': agency_code
            })

            self.assertEqual(200, response.status_code)
            data = response.json()
            self.assertEqual(expected_name, data['agency_name'])
```

STEP 5: 異常系テストの書き方

バリデーションエラーのテスト

python

```
def test_create_user_validation_error(self):
    """バリデーションエラーが正しく処理されること"""
    # 不正なデータ
    invalid_data = {
        'username': '',          # 必須項目が空
        'email': 'invalid-email', # 不正なメール形式
        'age': -1                # 不正な年齢
    }

    # API呼び出し
    response = self.client.post('/api/users/', invalid_data)

    # エラーレスポンス検証
    self.assertEqual(400, response.status_code)

    # エラー内容確認
    errors = response.json()
    self.assertIn('username', errors)
    self.assertIn('email', errors)
    self.assertIn('age', errors)
```

認証エラーのテスト

python

```
def test_unauthorized_access(self):
    """認証が必要なAPIに未認証でアクセス"""
    response = self.client.get('/api/private-data/')
    self.assertEqual(401, response.status_code)

def test_forbidden_access(self):
    """権限がないユーザーでアクセス"""
    # 一般ユーザーでログイン
    self.client.force_login(self.normal_user)

    # 管理者専用API呼び出し
    response = self.client.get('/api/admin-only/')
    self.assertEqual(403, response.status_code)
```

データ不存在のテスト

python

```
def test_get_nonexistent_user(self):
    """存在しないユーザーの取得"""
    response = self.client.get('/api/users/999/')
    self.assertEqual(404, response.status_code)
```

STEP 6: セキュリティテスト

パラメータ改ざん検出テスト

python

```
def test_parameter_tampering_detection(self):
    """パラメータ改ざんが検出されること"""
    # 正しいハッシュ値を計算
    correct_hash = calculate_hash('param1', 'param2')

    # 間違ったハッシュ値を設定
    tampered_data = {
        'param1': 'value1',
        'param2': 'value2',
        'hash': 'wrong_hash_value' # 改ざんされたハッシュ
    }

    response = self.client.post('/api/secure-endpoint/', tampered_data)

    # セキュリティエラーを確認
    self.assertEqual(400, response.status_code)
    self.assertIn('改ざんが検出されました', response.json()['message'])
```

SQLインジェクション対策テスト

python

```
def test_sql_injection_prevention(self):
    """SQLインジェクション攻撃が防がれること"""
    malicious_input = "'; DROP TABLE users; --"

    response = self.client.get(f'/api/search/?q={malicious_input}')

    # 正常にエスケープされて処理される
    self.assertEqual(200, response.status_code)

    # テーブルが削除されていないことを確認
    self.assertTrue(User.objects.exists())
```

STEP 7: ログ出力のテスト

LogCaptureの使い方

python

```
from testfixtures import LogCapture

def test_error_logging(self):
    """エラー時にログが正しく出力されること"""
    with LogCapture('myapp.views') as log:
        # エラーを発生させるAPI呼び出し
        response = self.client.post('/api/error-endpoint/')

        # エラーレスポンス確認
        self.assertEqual(500, response.status_code)

        # ログ出力確認
        log.check(
            ('myapp.views', 'ERROR', 'エラーが発生しました'),
        )

def test_security_warning_log(self):
    """セキュリティ警告ログの出力テスト"""
    with LogCapture('security') as log:
        # 改ざんされたリクエスト送信
        response = self.client.post('/api/secure/', {
            'data': 'tampered_data',
            'hash': 'wrong_hash'
        })

        # 警告ログ確認
        log.check(
            ('security', 'WARNING', 'パラメータ改ざんを検出'),
        )
```

STEP 8: モックとパッチの活用

外部API呼び出しのモック

python

```
from unittest.mock import patch

@patch('myapp.services.external_api_call')
def test_external_api_integration(self, mock_api):
    """外部API呼び出しをモックしてテスト"""
    # モックの戻り値を設定
    mock_api.return_value = {'status': 'success', 'data': 'test'}

    # API呼び出し
    response = self.client.post('/api/process-external-data/')

    # モックが呼び出されたことを確認
    mock_api.assert_called_once()

    # レスポンス確認
    self.assertEqual(200, response.status_code)
```

日時のモック

python

```
from freezegun import freeze_time

@freeze_time("2024-01-01 12:00:00")
def test_time_dependent_feature(self):
    """日時に依存する機能のテスト"""
    response = self.client.get('/api/current-time/')

    data = response.json()
    self.assertEqual('2024-01-01 12:00:00', data['timestamp'])
```

実践のポイント

1. テストケース設計

テストケースの分類

python

```
class UserAPITest(TestCase):
    # 正常系
    def test_create_user_success(self):
        pass

    def test_get_user_success(self):
        pass

    # 異常系 - バリデーション
    def test_create_user_invalid_email(self):
        pass

    def test_create_user_missing_required_field(self):
        pass

    # 異常系 - 権限
    def test_get_user_unauthorized(self):
        pass

    def test_update_user_forbidden(self):
        pass

    # 境界値
    def test_create_user_max_length_name(self):
        pass
```

テストの命名規則

python

 良い例：何をテストするかが明確

```
def test_create_user_with_valid_data_should_return_201(self):
    pass
```

```
def test_get_user_with_invalid_id_should_return_404(self):
    pass
```

 悪い例：何をテストするかが不明

```
def test_user(self):
    pass
```

```
def test_api_call(self):
    pass
```

2. テストデータ管理

ファクトリパターンの活用

python

```
class UserFactory:
    @staticmethod
    def create_user(**kwargs):
        defaults = {
            'username': 'testuser',
            'email': 'test@example.com',
            'is_active': True
        }
        defaults.update(kwargs)
        return User.objects.create(**defaults)

# 使用例
def test_get_user(self):
    user = UserFactory.create_user(username='specific_user')
    response = self.client.get(f'/api/users/{user.id}/')
    self.assertEqual(200, response.status_code)
```

テストデータの分離

python

```
# テストデータを別ファイルに定義
# test_data.py
AGENCY_DATA = {
    'default': {
        'agency_cd': '0001',
        'agency_name': 'デフォルト代理店',
        'phone_number': '0120-000-001'
    },
    'partner_a': {
        'agency_cd': '0002',
        'agency_name': 'パートナーA',
        'phone_number': '0120-000-002'
    }
}


# テストファイルで使用
from .test_data import AGENCY_DATA


class AgencyAPITest(TestCase):
    @classmethod
    def setUpTestData(cls):
        for key, data in AGENCY_DATA.items():
            Agency.objects.create(**data)
```

3. アサーションのベストプラクティス

適切なアサーションメソッドの選択

python

```
#  良い例：目的に応じたアサーション
self.assertEqual(200, response.status_code)
self.assertIn('error', response.json())
self.assertIsNone(user.deleted_at)
self.assertTrue(user.is_active)
self.assertGreater(len(users), 0)

#  悪い例：不適切なアサーション
self.assertTrue(response.status_code == 200)
self.assertEqual(True, user.is_active)
```

```
# 等値比較
# 包含確認
# None確認
# Boolean確認
# 大小比較
```

```
# 冗長
# 不自然
```

エラーメッセージの活用

python

```
# カスタムエラーメッセージ
self.assertEqual(
    expected_count,
    actual_count,
    f"Expected {expected_count} users, but got {actual_count}"
)

# コンテキスト情報付きアサーション
with self.subTest(user_id=user.id):
    response = self.client.get(f'/api/users/{user.id}/')
    self.assertEqual(200, response.status_code)
```

デバッグとトラブルシューティング

1. テスト失敗時のデバッグ

レスポンス内容の確認

python

```
def test_debug_response(self):
    response = self.client.post('/api/users/', data)

    # デバッグ用出力
    print(f"Status: {response.status_code}")
    print(f"Content: {response.content}")
    print(f"JSON: {response.json()}")

    # 詳細なアサーション
    self.assertEqual(201, response.status_code,
                     f"Expected 201, got {response.status_code}. "
                     f"Response: {response.content}")
```

データベース状態の確認

python

```
def test_debug_database(self):
    # テスト実行前の状態確認
    initial_count = User.objects.count()
    print(f"Initial user count: {initial_count}")

    # API呼び出し
    response = self.client.post('/api/users/', data)

    # テスト実行後の状態確認
    final_count = User.objects.count()
    print(f"Final user count: {final_count}")

    # 変化量の確認
    self.assertEqual(initial_count + 1, final_count)
```

2. よくある問題と解決法

データベーストランザクションの問題

python

```
# 問題：テスト中のデータが他のテストに影響
class ProblematicTest(TestCase):
    def test_a(self):
        User.objects.create(username='test')
        # このデータが次のテストに残る可能性

# 解決：適切なクリーンアップ
class GoodTest(TestCase):
    def tearDown(self):
        User.objects.all().delete() # テスト後のクリーンアップ
```

非同期処理のテスト

python

```
import time

def test_async_processing(self):
    # 非同期処理を開始
    response = self.client.post('/api/async-process/')
    self.assertEqual(202, response.status_code) # Accepted

    # 処理完了を待つ
    for _ in range(10): # 最大10秒待機
        status_response = self.client.get('/api/process-status/')
        if status_response.json()['status'] == 'completed':
            break
        time.sleep(1)
    else:
        self.fail("Async process did not complete in time")
```



よくあるエラーと対処法

1. テストデータ関連

IntegrityError: UNIQUE constraint failed

対処法：

python

```
# 重複データの確認
def setUpTestData(cls):
    # 既存データをチェック
    if not User.objects.filter(username='testuser').exists():
        User.objects.create(username='testuser')
```

2. APIクライアント関連

AttributeError: 'WSGIRequest' object has no attribute 'json'

対処法：


```
python
```

```
# 正しいレスポンス取得方法
```

```
import json
```

```
response = self.client.post('/api/endpoint/')
```

```
data = json.loads(response.content) # または response.json()
```

3. モック関連

```
AssertionError: Expected call not found
```

対処法：

```
python
```

```
# モックの呼び出し確認
```

```
with patch('module.function') as mock_func:
```

```
    # API呼び出し
```

```
    response = self.client.post('/api/endpoint/')
```

```
    # 呼び出し確認 (引数も含めて)
```

```
    mock_func.assert_called_with(expected_arg1, expected_arg2)
```

学習の次のステップ

1. パフォーマンステスト

- レスポンス時間の測定
- 負荷テスト
- メモリ使用量監視

2. 統合テスト

- 複数のAPIの連携テスト
- エンドツーエンドテスト
- システム全体の動作確認

3. テスト自動化

- CI/CDパイプライン
- 継続的テスト
- テストレポート生成

4. 高度なテスト技法

- プロパティベーステスト

- 契約テスト
- カオスエンジニアリング



参考資料

- [Django Testing Documentation](#)
- [Django REST Framework Testing](#)
- [Python unittest Documentation](#)
- [testfixtures Documentation](#)

頑張って学習を続けましょう！🚀