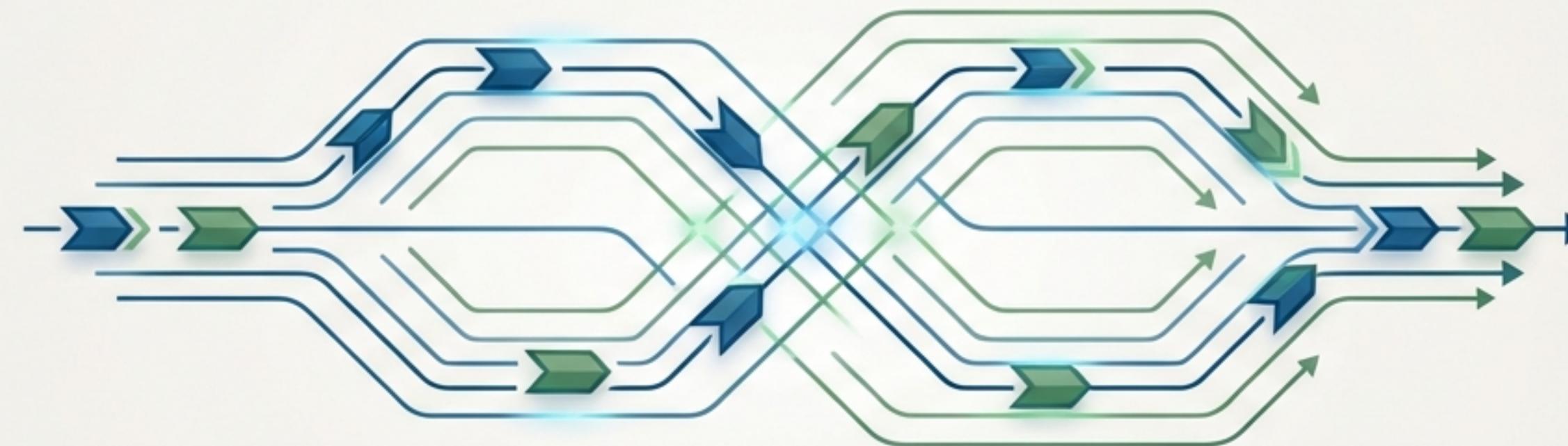


کندوکاوی عمیق در View‌ها در Django REST Framework

(Function-Based Views) و نماهای مبتنی بر تابع (APIView)



برگرفته از مستندات رسمی، برای توسعه‌دهندگانی که به دنبال درک کامل معماری View‌ها هستند.



معماری View در DRF: دو رویکرد اصلی

نماهای مبتنی بر کلاس (Class-Based Views)

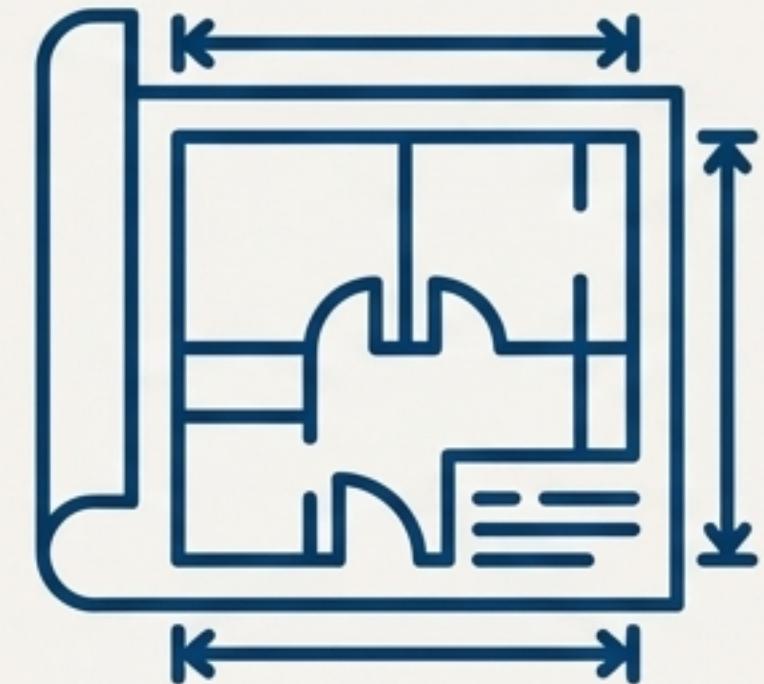
قدرت، ساختار و قابلیت استفاده مجدد با `.APIView`.

API Endpoint

نماهای مبتنی بر تابع (Function-Based Views)

садگی، وضوح و سرعت با دکوراتور `@api_view`.

این ارائه هر دو مسیر را به تفصیل کالبدشکافی می‌کند تا شما بتوانید آگاهانه بهترین ابزار را برای کار خود انتخاب کنید.



بخش اول: نماهای مبتنی بر کلاس (Class-Based Views)

کالبدشکافی قدرت و انعطاف‌پذیری `APIView`

:سنگ بنای View های مبتنی بر کلاس APIView

'زنگو' است، اما با ۴ تفاوت کلیدی که برای ساخت API ها طراحی شده اند:

انعطاف پذیر Response →

متدها می توانند نمونه ای از کلاس 'Response' را برگردانند.
فریمورک به طور خودکار Content Negotiation را انجام داده
خروجی را با Renderer مناسب (مثلًا JSON) رندر رندر
می کند.

پیشرفته Request →]

'Request'، نمونه ای از کلاس 'Handler' است، نه 'HttpRequest' زنگو.
این به معنی دسترسی آسان به داده های parse شده
(request.data) است.

اجرای خودکار Policy ها



قبل از اینکه درخواست به متدهای Handler (مانند (.get(), .post(), .put(), .patch(), .delete())) برسد، بررسی های احراز هویت (Authentication)، مجوزها (Permissions) و محدودسازی نرخ درخواست (Throttling) به طور خودکار اجرا می شوند.

مدیریت مرکز استثناهای (Exceptions)



هر استثنای از نوع 'APIException' به طور خودکار گرفته شده و به یک پاسخ خطای مناسب و ساختار یافته تبدیل می شود.

در عمل: یک مثال کاربردی `APIView`

یک View برای لیست کردن تمام کاربران سیستم که نیازمند احراز هویت با توکن است و فقط برای کاربران ادمین قابل دسترسی است.

```
from rest_framework.views import APIView
from rest_framework.response import Response
from rest_framework import authentication, permissions
from django.contrib.auth.models import User

class ListUsers(APIView):
    """
    View to list all users in the system.
    * Requires token authentication.
    * Only admin users are able to access this view.
    """
    authentication_classes = [authentication.TokenAuthentication]
    permission_classes = [permissions.IsAdminUser]

    def get(self, request, format=None):
        """
        Return a list of all users.
        """
        usernames = [user.username for user in User.objects.all()]
        return Response(usernames)
```

تعیین نحوه احراز هویت به
صورت اعلانی (Declarative)

تعیین قوانین دسترسی و
مجوزها

بازگرداندن `HttpResponse` فریمورک به جای `Response`

View: پیکربندی رفتار Policy‌های اتربیوت

اتربیوت	شرح
	کلاس‌های مسئول رندر کردن خروجی (e.g., JSONRenderer)
	کلاس‌های مسئول پارس کردن ورودی (e.g., JSONParser)
	لیست کلاس‌های احراز هویت
	لیست کلاس‌های محدودکننده نرخ درخواست
	لیست کلاس‌های بررسی مجوز دسترسی
	کلاس مسئول انتخاب Parser و Renderer مناسب

متدهای چرخه حیات: سفارشی‌سازی فرآیند پردازش درخواست

۱. مقداردهی اولیه Policy‌ها (Policy Instantiation)

این متدها برای ایجاد نمونه از کلاس‌های Policy استفاده می‌شوند. به ندرت نیاز به بازنویسی (override) دارند.

```
.get_renderers() .get_parsers() .get_authenticators() .get_throttles()  
•  
.get_permissions() .get_content_negotiator() .get_exception_handler()
```

۲. اجرای Policy‌ها (Policy Implementation)

این متدها درست قبل از اجرای متدهای Handler فراخوانی می‌شوند تا قوانین را اجرا کنند.

```
.check_permissions(request) .check_throttles(request) .perform_content_negotiation(request)
```

۳. متدهای Dispatch (هسته اصلی چرخه)

این متدها مستقیماً توسط `dispatch()` فراخوانی شده و هسته اصلی چرخه درخواست/پاسخ را تشکیل می‌دهند.

```
.initial(request, ...) .Content Negotiation  
• اجرای Policy‌ها و  
.handle_exception(exc) . نکته مهم*: برای سفارشی‌سازی پاسخ‌های خطا در API، این متدها را بازنویسی کنید.  
.initialize_request(request, ...) . تبدیل `HttpRequest` به `Request`  
.finalize_response(request, response, ...) . رندر نهایی پاسخ با Renderer انتخاب شده.
```





بخش دوم: نماهای مبتنی بر تابع (Function-Based Views)

دستیابی به سادگی و قدرت با دکوراتور `@api_view`

افزودن قابلیت‌های DRF به توابع ساده:@api_view

Signature: `@api_view(http_method_names=['GET'])`

این دکوراتور توابع شما را بسته‌بندی (wrap) می‌کند تا اطمینان حاصل شود که یک نمونه از `Request` دریافت کرده و می‌توانند یک `Response` برگردانند.

مثال ۲: پذیرش متدهای مختلف

```
@api_view(['GET', 'POST'])
def hello_world(request):
    if request.method == 'POST':
        return Response({
            "message": "Got some data!",
            "data": request.data
        })
    return Response({"message": "Hello, world!"})
```

مثال ۱: یک ویو ساده (فقط GET)

```
from rest_framework.decorators import api_view
from rest_framework.response import Response

@api_view() # By default, only GET is allowed
def hello_world(request):
    return Response({"message": "Hello, world!"})
```

دکوراتورهای Policy: معادل اتریبیوت‌های `APIView`

قانون کلیدی": برای بازنویسی تنظیمات پیش‌فرض، از دکوراتورهای Policy بعد از (زیر) دکوراتور استفاده کنید. `@api_view`

لیست دکوراتورهای موجود

- `@renderer_classes(...)`
- `@parser_classes(...)`
- `@authentication_classes(...)`
- `@throttle_classes(...)`
- `@permission_classes(...)`
- `@content_negotiation_class(...)`
- `@metadata_class(...)`
- `@versioning_class(...)`

مثال کد عملی (Throttling)

```
from rest_framework.decorators import api_view,  
throttle_classes  
from rest_framework.throttling import  
UserRateThrottle  
  
class OncePerDayUserThrottle(UserRateThrottle):  
    rate = '1/day'  
  
@api_view(['GET'])  
@throttle_classes([OncePerDayUserThrottle])  
def view(request):  
    return Response({"message": "Hello for today!"})
```

کنترل پیشرفت‌های Schema با @schema: سفارشی‌سازی

با استفاده از دکوراتور `@schema`، می‌توانید نحوه نمایش view در مستندات خودکار API (مانند Swagger/A) را کنترل کنید.

۲. حذف یک Schema View

```
@api_view(['GET'])
@schema(None)
def view(request):
    # This view will not appear in the generated schema
    return Response({"message": "Hidden from schema!"})
```

۱. استفاده از یک کلاس Schema سفارشی

```
from rest_framework.decorators import schema
from rest_framework.schemas import AutoSchema

class CustomAutoSchema(AutoSchema):
    # ... override methods here ...
```

```
@api_view(['GET'])
@schema(CustomAutoSchema())
def view(request):
    # ...
```

انتخاب ابزار مناسب: `@api_view` در برابر `APIView`

@api_view (مبنی بر تابع)	APIView (مبنی بر کلاس)	ویژگی (Feature)
یک تابع واحد با دکوراتور	HTTP method هر کلاس با متدها برای (.get,.post)	ساختار (Structure)
از طریق دکوراتورهای اضافی (@permission_classes(...))	از طریق اتربیوت‌های کلاس (.permission_classes = ...)	پیکربندی (Policy)
پایین؛ تمرکز بر روی یک وظیفه مشخص	بالا؛ ایده‌آل برای استفاده از ارث‌بری Mixins و Inheritance	قابلیت استفاده مجدد
برای endpoint‌های ساده و سرراست، بسیار واضح و مختصر است	برای منطق‌های پیچیده و چندبخشی، بسیار ساختاریافته است	خوانایی (Readability)
ابزارهای Endpoint API‌های کوچک و متوسط	API‌های بزرگ، View‌های با منطق مشترک، Endpoint‌های پیچیده	بهترین کاربرد

جمع‌بندی نهایی و گام‌های بعدی

برای مطالعه بیشتر

برای مشاهده کامل متدها و اتریوت‌های هر کلاس در DRF، منبع روشنل Classy Django REST Framework یک راهنمای تعاملی و بسیار مفید است که در مستندات رسمی نیز به آن اشاره شده است.

Classy Django REST Framework



Official DRF Views Documentation



نکات کلیدی (Key Takeaways)

- دو رویکرد قدرتمند و انعطاف‌پذیر برای ساخت View‌ها در DRF ارائه می‌دهد.
- برای ساختارهای پیچیده، قابل استفاده مجدد و نیازمند سفارشی‌سازی عمیق در چرخه حیات درخواست، ایده‌آل است.
- برای endpoint‌های ساده و سریع، وضوح و سادگی فوق العاده‌ای به ارمغان می‌آورد، در حالی که تمام قدرت DRF . Policy را حفظ می‌کند.
- انتخاب هوشمندانه بین این دو، کلید معماری API‌های کارآمد، تمیز و قابل نگهداری است.