APIView

REST framework提供了一个 `APIView` 类，它是Django的 `View` 类的子类。

`APIView` 类和Django原生的类视图的 `View` 类有以下不同：

- 传入的请求对象不是Django原生的 `HttpRequest` 类的实例，而是REST framework的 `Request` 类的实例。
- 视图返回的是REST framework的 `Response` 响应，而不是Django的 `HttpResponse` 。视图会管理内容协商的结果，给响应设置正确的渲染器。
- 任何 `APIException` 异常都会被捕获，并且传递给合适的响应。
- 请求对象会经过合法性检验，权限验证，或者阈值检查后，再被派发到相应的视图。

使用 `APIView` 类和使用一般的 `View` 类非常相似，通常，进入的请求会被分发到合适的处理方法比如 `.get()` ，或者 `.post` 。

比如：

```
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)
```

APIView的所有属性和方法：

- allowed_methods
- as_view
- authentication_classes
- check_object_permissions
- check_permissions
- check_throttles
- content_negotiation_class
- default_response_headers
- determine_version
- dispatch
- finalize_response
- get_authenticate_header
- get_authenticators
- get_content_negotiator
- get_exception_handler
- get_exception_handler_context
- get_format_suffix
- get_parser_context
- get_parsers
- get_permissions
- get_renderer_context
- get_renderers
- get_throttles
- get_view_description
- get_view_name
- handle_exception
- http_method_names
- http_method_not_allowed
- initial
- initialize_request
- metadata_class
- options
- parser_classes
- perform_authentication
- perform_content_negotiation
- permission_classes

- permission_denied
- raise_uncaught_exception
- renderer_classes
- schema
- settings
- setup
- throttle_classes
- throttled
- versioning_class

API视图的属性

了解即可。

- .renderer_classes
- .parser_classes
- .authentication_classes
- .throttle_classes
- .permission_classes
- .content_negotiation_class

API实例化方法属性

通常不需要重写这些方法，了解即可。

- .get_renderers(self)
- .get_parsers(self)
- .get_authenticators(self)
- .get_throttles(self)
- .get_permissions(self)
- .get_content_negotiator(self)
- .get_exception_handler(self)

API实现方法

下面这些方法会在请求被分发到具体的处理方法之前调用。了解即可。

- .check_permissions(self, request)
- .check_throttles(self, request)
- .perform_content_negotiation(self, request, force=False)

分发dispatch()

下面这些方法会被视图的 `.dispatch()` 方法直接调用。它们在调用 `.get` 、 `.post()` 、 `put()` 、 `patch()` 和 `delete()` 之类的请求处理方法之前或者之后执行任何需要执行的操作。

.initial(self, request, *args, **kwargs)

在处理方法调用之前进行任何需要的动作。 这个方法用于执行权限认证和限制，并且执行内容协商，通常不需要重写此方法。

.handle_exception(self, exc)

任何被处理请求的方法抛出的异常都会被传递给这个方法，这个方法既不返回 `Response` 的实例，也不重新抛出异常。

默会处理 `rest_framework.expceptions.APIException` 的子类异常，以及Django的 `Http404` 和 `PermissionDenied` 异常，并且返回一个适当的错误响应。

如果你需要在自己的API中自定义返回的错误响应，你可以重写这个方法。

.initialize_request(self, request, *args, **kwargs)

这个方法确保传递给视图的请求对象是 `Request` 的实例，而不是原生的Django `HttpResquest` 的实例。通常不需要重写这个方法。

.finalize_response(self, request, response, *args, **kwargs)

确保任何从处理请求的方法返回的 `Response` 对象被渲染到由内容协商决定的正确内容类型。通常不需要重写这个方法。

@api_view()

在REST framework中，也可以使用常规的基于函数的视图。DRF提供了一组简单的装饰器，用来包装你的视图函数，以确保视图函数会收到 `Request` （而不是Django原生的 `HttpRequest`)对象，并且返回 `Response` （而不是Django的 `HttpResponse` ）对象，同时允许你设置这个请求的处理方式。

@api_view()装饰器

签名: `@api_view(http_method_names=['GET'], exclude_from_schema=False)`

`api_view` 装饰器的主要参数是响应的HTTP方法的列表。 比如，你可以像这样写一个返回一些数据的非常简单的视图。

```
from rest_framework.decorators import api_view

@api_view()
def hello_world(request):
    return Response({"message": "Hello, world!"})
```

这个视图会使用settings中指定的默认的渲染器，解析器，认证类等等。

默认的情况下，只有 `GET` 请求会被接受。其他的请求方法会得到一个"405 Method Not Allowed"响应。可以像下面的示例代码一样改变默认行为：

```
@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!"})
```

API 访问策略装饰器

REST framework提供了一组可以加到视图上的装饰器来重写一些默认设置。这些装饰器必须放在 `@api_view` 装饰器的 *后* (下)面。比如，要创建一个使用限制器确保特定用户每天只能调用一次的视图，可以用 `@throttle_classes` 装饰器并给它传递一个限制器类的列表。

```
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! See you tomorrow!"})
```

这些装饰器和前文中的 `APIView` 的子类中设置的属性相对应。

可用的装饰器有：

- `@renderer_classes(...)`
- `@parser_classes(...)`
- `@authentication_classes(...)`
- `@throttle_classes(...)`
- `@permission_classes(...)`

这些装饰器都只接受一个参数，这个参数必须是类的列表或元组。

视图模式装饰器

要重写默认的基于函数的视图生成的模式，你需要使用 `@schema` 装饰器。它必须放在 `@api_view` 装饰器后面，例如：

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

class CustomAutoSchema(AutoSchema):
    def get_link(self, path, method, base_url):
        # override view introspection here...

@api_view(['GET'])
@schema(CustomAutoSchema())
def view(request):
    return Response({"message": "Hello for today! See you tomorrow!"})
```

如果给装饰器传递一个 `None` 参数值，那么会将函数排除在模式生成之外。

```
@api_view(['GET'])
@schema(None)
def view(request):
    return Response({"message": "Will not appear in schema!"})
```