views.py decorators.py

Class-based Views

> *Django's class-based views are a welcome departure from the old-style views.*
>
> *— Reinout van Rees*

REST framework provides an `APIView` class, which subclasses Django's `View` class.

`APIView` classes are different from regular `View` classes in the following ways:

- Requests passed to the handler methods will be REST framework's `Request` instances, not Django's `HttpRequest` instances.
- Handler methods may return REST framework's `Response`, instead of Django's `HttpResponse`. The view will manage content negotiation and setting the correct renderer on the response.
- Any `APIException` exceptions will be caught and mediated into appropriate responses.
- Incoming requests will be authenticated and appropriate permission and/or throttle checks will be run before dispatching the request to the handler method.

Using the `APIView` class is pretty much the same as using a regular `View` class, as usual, the incoming request is dispatched to an appropriate handler method such as `.get()` or `.post()`. Additionally, a number of attributes may be set on the class that control various aspects of the API policy.

For example:

```
from rest_framework.views import APIView
from rest_framework.response import Response
from rest_framework import authentication, permissions

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)
```

API policy attributes

The following attributes control the pluggable aspects of API views.

.renderer_classes

.parser_classes

.authentication_classes

.throttle_classes

.permission_classes

.content_negotiation_class

API policy instantiation methods

The following methods are used by REST framework to instantiate the various pluggable API policies. You won't typically need to override these methods.

.get_renderers(self)

.get_parsers(self)

.get_authenticators(self)

.get_throttles(self)

.get_permissions(self)

.get_content_negotiator(self)

.get_exception_handler(self)

API policy implementation methods

The following methods are called before dispatching to the handler method.

.check_permissions(self, request)

.check_throttles(self, request)

.perform_content_negotiation(self, request, force=False)

Dispatch methods

The following methods are called directly by the view's `.dispatch()` method. These perform any actions that need to occur before or after calling the handler methods such as `.get()`, `.post()`, `put()`, `patch()` and `.delete()`.

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

Performs any actions that need to occur before the handler method gets called. This method is used to enforce permissions and throttling, and perform content negotiation.

You won't typically need to override this method.

.handle_exception(self, exc)

Any exception thrown by the handler method will be passed to this method, which either returns a `Response` instance, or re-raises the exception.

The default implementation handles any subclass of `rest_framework.exceptions.APIException`, as well as Django's `Http404` and `PermissionDenied` exceptions, and returns an appropriate error response.

If you need to customize the error responses your API returns you should subclass this method.

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

Ensures that the request object that is passed to the handler method is an instance of `Request`, rather than the usual Django `HttpRequest`.

You won't typically need to override this method.

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

Ensures that any `Response` object returned from the handler method will be rendered into the correct content type, as determined by the content negotiation.

You won't typically need to override this method.

Function Based Views

> *Saying [that class-based views] is always the superior solution is a mistake.*
>
> — *Nick Coghlan*

REST framework also allows you to work with regular function based views. It provides a set of simple decorators that wrap your function based views to ensure they receive an instance of `Request` (rather than the usual Django `HttpRequest`) and allows them to return a `Response` (instead of a Django `HttpResponse`), and allow you to configure how the request is processed.

@api_view()

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

The core of this functionality is the `api_view` decorator, which takes a list of HTTP methods that your view should respond to. For example, this is how you would write a very simple view that just manually returns some data:

```
from rest_framework.decorators import api_view

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

This view will use the default renderers, parsers, authentication classes etc specified in the settings.

By default only `GET` methods will be accepted. Other methods will respond with "405 Method Not Allowed". To alter this behaviour, specify which methods the view allows, like so:

```
@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!"})
```

You can also mark an API view as being omitted from any auto-generated schema, using the `exclude_from_schema` argument.:

```
@api_view(['GET'], exclude_from_schema=True)
def api_docs(request):
    ...
```

API policy decorators

To override the default settings, REST framework provides a set of additional decorators which can be added to your views. These must come *after* (below) the `@api_view` decorator. For example, to create a view that uses a throttle to ensure it can only be called once per day by a particular user, use the `@throttle_classes` decorator, passing a list of throttle classes:

```
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!"})
```

These decorators correspond to the attributes set on `APIView` subclasses, described above.

The available decorators are:

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

Each of these decorators takes a single argument which must be a list or tuple of classes.
