Requests

REST framework's `Request` class extends the standard `HttpRequest`, adding support for REST framework's flexible request parsing and request authentication.

Request parsing

REST framework's Request objects provide flexible request parsing that allows you to treat requests with JSON data or other media types in the same way that you would normally deal with form data.

.data

`request.data` returns the parsed content of the request body. This is similar to the standard `request.POST` and `request.FILES` attributes except that:

- It includes all parsed content, including *file and non-file* inputs.
- It supports parsing the content of HTTP methods other than `POST`, meaning that you can access the content of `PUT` and `PATCH` requests.
- It supports REST framework's flexible request parsing, rather than just supporting form data. For example you can handle incoming JSON data similarly to how you handle incoming form data.

.query_params

`request.query_params` is a more correctly named synonym for `request.GET`.

For clarity inside your code, we recommend using `request.query_params` instead of the Django's standard `request.GET`. Doing so will help keep your codebase more correct and obvious - any HTTP method type may include query parameters, not just `GET` requests.

Authentication

REST framework provides flexible, per-request authentication, that gives you the ability to:

- Use different authentication policies for different parts of your API.
- Support the use of multiple authentication policies.
- Provide both user and token information associated with the incoming request.

.user

`request.user` typically returns an instance of `django.contrib.auth.models.User`, although the behavior depends on the authentication policy being used.

If the request is unauthenticated the default value of `request.user` is an instance of `django.contrib.auth.models.AnonymousUser`.

.auth

`request.auth` returns any additional authentication context. The exact behavior of `request.auth` depends on the authentication policy being used, but it may typically be an instance of the token that the request was authenticated against.

If the request is unauthenticated, or if no additional context is present, the default value of `request.auth` is `None`.

Browser enhancements

REST framework supports a few browser enhancements such as browser-based `PUT`, `PATCH` and `DELETE` forms.

.method

`request.method` returns the **uppercase** string representation of the request's HTTP method.

Browser-based `PUT`, `PATCH` and `DELETE` forms are transparently supported.

For more information see the browser enhancements documentation.

.content_type

`request.content_type`, returns a string object representing the media type of the HTTP request's body, or an empty string if no media type was provided.

Responses

REST framework supports HTTP content negotiation by providing a `Response` class which allows you to return content that can be rendered into multiple content types, depending on the client request.

Unless you want to heavily customize REST framework for some reason, you should always use an `APIView` class or `@api_view` function for views that return `Response` objects. Doing so ensures that the view can perform content negotiation and select the appropriate renderer for the response, before it is returned from the view.

Creating responses

Response()

Signature: `Response(data, status=None, template_name=None, headers=None, content_type=None)`

Unlike regular `HttpResponse` objects, you do not instantiate `Response` objects with rendered content. Instead you pass in unrendered data, which may consist of any Python primitives.

The renderers used by the `Response` class cannot natively handle complex data types such as Django model instances, so you need to serialize the data into primitive data types before creating the `Response` object.

You can use REST framework's `Serializer` classes to perform this data serialization, or use your own custom serialization.

Arguments:

- `data`: The serialized data for the response.

- `status`: A status code for the response. Defaults to 200. See also status codes.
- `template_name`: A template name to use if `HTMLRenderer` is selected.
- `headers`: A dictionary of HTTP headers to use in the response.
- `content_type`: The content type of the response. Typically, this will be set automatically by the renderer as determined by content negotiation, but there may be some cases where you need to specify the content type explicitly.

Class-based Views

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.

```
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)
```

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 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)

Function Based Views

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'])`

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
from rest_framework.response import Response

@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!"})
```

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.

Generic views

One of the key benefits of class-based views is the way they allow you to compose bits of reusable behavior. REST framework takes advantage of this by providing a number of pre-built views that provide for commonly used patterns.

The generic views provided by REST framework allow you to quickly build API views that map closely to your database models.

If the generic views don't suit the needs of your API, you can drop down to using the regular `APIView` class, or reuse the mixins and base classes used by the generic views to compose your own set of reusable generic views.

Examples

Typically when using the generic views, you'll override the view, and set several class attributes.

```
from django.contrib.auth.models import User
from myapp.serializers import UserSerializer
from rest_framework import generics
from rest_framework.permissions import IsAdminUser


class UserList(generics.ListCreateAPIView):
    queryset = User.objects.all()
    serializer_class = UserSerializer
    permission_classes = [IsAdminUser]
```

For more complex cases you might also want to override various methods on the view class. For example.

```
class UserList(generics.ListCreateAPIView):
    queryset = User.objects.all()
    serializer_class = UserSerializer
    permission_classes = [IsAdminUser]
```

```
    def list(self, request):
        # Note the use of `get_queryset()` instead of `self.queryset`
        queryset = self.get_queryset()
        serializer = UserSerializer(queryset, many=True)
        return Response(serializer.data)
```

GenericAPIView

This class extends REST framework's `APIView` class, adding commonly required behavior for standard list and detail views.

Each of the concrete generic views provided is built by combining `GenericAPIView`, with one or more mixin classes.

Attributes

Basic settings:

The following attributes control the basic view behavior.

- `queryset` - The queryset that should be used for returning objects from this view. Typically, you must either set this attribute, or override the `get_queryset()` method. If you are overriding a view method, it is important that you call `get_queryset()` instead of accessing this property directly, as `queryset` will get evaluated once, and those results will be cached for all subsequent requests.
- `serializer_class` - The serializer class that should be used for validating and deserializing input, and for serializing output. Typically, you must either set this attribute, or override the `get_serializer_class()` method.
- `lookup_field` - The model field that should be used to for performing object lookup of individual model instances. Defaults to `'pk'`. Note that when using hyperlinked APIs you'll need to ensure that *both* the API views *and* the serializer classes set the lookup fields if you need to use a custom value.

- `lookup_url_kwarg` - The URL keyword argument that should be used for object lookup. The URL conf should include a keyword argument corresponding to this value. If unset this defaults to using the same value as `lookup_field`.

Pagination:

The following attributes are used to control pagination when used with list views.

- `pagination_class` - The pagination class that should be used when paginating list results. Defaults to the same value as the `DEFAULT_PAGINATION_CLASS` setting, which is `'rest_framework.pagination.PageNumberPagination'`. Setting `pagination_class=None` will disable pagination on this view.

Filtering:

- `filter_backends` - A list of filter backend classes that should be used for filtering the queryset. Defaults to the same value as the `DEFAULT_FILTER_BACKENDS` setting.

Methods

Base methods:

`get_queryset(self)`

Returns the queryset that should be used for list views, and that should be used as the base for lookups in detail views. Defaults to returning the queryset specified by the `queryset` attribute.

This method should always be used rather than accessing `self.queryset` directly, as `self.queryset` gets evaluated only once, and those results are cached for all subsequent requests.

May be overridden to provide dynamic behavior, such as returning a queryset, that is specific to the user making the request.

For example:

```
def get_queryset(self):
    user = self.request.user
    return user.accounts.all()
```

Note: If the serializer_class used in the generic view spans orm relations, leading to an n+1 problem, you could optimize your queryset in this method using `select_related` and

`prefetch_related`. To get more information about n+1 problem and use cases of the mentioned methods refer to related section in django documentation.

`get_object(self)`

Returns an object instance that should be used for detail views. Defaults to using the `lookup_field` parameter to filter the base queryset.

May be overridden to provide more complex behavior, such as object lookups based on more than one URL kwarg.

For example:

```
def get_object(self):
    queryset = self.get_queryset()
    filter = {}
    for field in self.multiple_lookup_fields:
        filter[field] = self.kwargs[field]

    obj = get_object_or_404(queryset, **filter)
    self.check_object_permissions(self.request, obj)
    return obj
```

Note that if your API doesn't include any object level permissions, you may optionally exclude the `self.check_object_permissions`, and simply return the object from the `get_object_or_404` lookup.

`filter_queryset(self, queryset)`

Given a queryset, filter it with whichever filter backends are in use, returning a new queryset.

For example:

```
def filter_queryset(self, queryset):
    filter_backends = [CategoryFilter]

    if 'geo_route' in self.request.query_params:
        filter_backends = [GeoRouteFilter, CategoryFilter]
    elif 'geo_point' in self.request.query_params:
        filter_backends = [GeoPointFilter, CategoryFilter]
```

```
    for backend in list(filter_backends):
        queryset = backend().filter_queryset(self.request, queryset, view=self)

    return queryset
```

`get_serializer_class(self)`

Returns the class that should be used for the serializer. Defaults to returning the `serializer_class` attribute.

May be overridden to provide dynamic behavior, such as using different serializers for read and write operations, or providing different serializers to different types of users.

For example:

```
def get_serializer_class(self):
    if self.request.user.is_staff:
        return FullAccountSerializer
    return BasicAccountSerializer
```

Save and deletion hooks:

The following methods are provided by the mixin classes, and provide easy overriding of the object save or deletion behavior.

- `perform_create(self, serializer)` - Called by `CreateModelMixin` when saving a new object instance.
- `perform_update(self, serializer)` - Called by `UpdateModelMixin` when saving an existing object instance.
- `perform_destroy(self, instance)` - Called by `DestroyModelMixin` when deleting an object instance.

These hooks are particularly useful for setting attributes that are implicit in the request, but are not part of the request data. For instance, you might set an attribute on the object based on the request user, or based on a URL keyword argument.

```
def perform_create(self, serializer):
    serializer.save(user=self.request.user)
```

These override points are also particularly useful for adding behavior that occurs before or after saving an object, such as emailing a confirmation, or logging the update.

```
def perform_update(self, serializer):
    instance = serializer.save()
    send_email_confirmation(user=self.request.user, modified=instance)
```

You can also use these hooks to provide additional validation, by raising a `ValidationError()`. This can be useful if you need some validation logic to apply at the point of database save. For example:

```
def perform_create(self, serializer):
    queryset = SignupRequest.objects.filter(user=self.request.user)
    if queryset.exists():
        raise ValidationError('You have already signed up')
    serializer.save(user=self.request.user)
```

We need to use the generic api view with mixins or use the concrete views.

There are few methods from generic api view which we can override if necessary.

The mixins classes provide the action methods like list,create,retrieve,update,destroy action methods which are used to implement how we get the data and save the data and update and destroy the data.

The concrete view will use the generic api view along with mixins to give the handler methods pre-written.

So just we need to specify the attributes in the generic api view that's it.

Else we can use the generic api view along with mixins to create our own concrete views.

Mixins

The mixin classes provide the actions that are used to provide the basic view behavior. Note that the mixin classes provide action methods rather than defining the handler methods, such as `.get()` and `.post()`, directly. This allows for more flexible composition of behavior.

The mixin classes can be imported from `rest_framework.mixins`.

ListModelMixin

Provides a `.list(request, *args, **kwargs)` method, that implements listing a queryset.

If the queryset is populated, this returns a `200 OK` response, with a serialized representation of the queryset as the body of the response. The response data may optionally be paginated.

CreateModelMixin

Provides a `.create(request, *args, **kwargs)` method, that implements creating and saving a new model instance.

If an object is created this returns a `201 Created` response, with a serialized representation of the object as the body of the response. If the representation contains a key named `url`, then the `Location` header of the response will be populated with that value.

If the request data provided for creating the object was invalid, a `400 Bad Request` response will be returned, with the error details as the body of the response.

RetrieveModelMixin

Provides a `.retrieve(request, *args, **kwargs)` method, that implements returning an existing model instance in a response.

If an object can be retrieved this returns a `200 OK` response, with a serialized representation of the object as the body of the response. Otherwise it will return a `404 Not Found`.

UpdateModelMixin

Provides a `.update(request, *args, **kwargs)` method, that implements updating and saving an existing model instance.

Also provides a `.partial_update(request, *args, **kwargs)` method, which is similar to the `update` method, except that all fields for the update will be optional. This allows support for HTTP `PATCH` requests.

If an object is updated this returns a `200 OK` response, with a serialized representation of the object as the body of the response.

If the request data provided for updating the object was invalid, a `400 Bad Request` response will be returned, with the error details as the body of the response.

DestroyModelMixin

Provides a `.destroy(request, *args, **kwargs)` method, that implements deletion of an existing model instance.

If an object is deleted this returns a `204 No Content` response, otherwise it will return a `404 Not Found`.

Concrete View Classes

The following classes are the concrete generic views. If you're using generic views this is normally the level you'll be working at unless you need heavily customized behavior.

The view classes can be imported from `rest_framework.generics`.

CreateAPIView

Used for **create-only** endpoints.

Provides a `post` method handler.

Extends: GenericAPIView, CreateModelMixin

ListAPIView

Used for **read-only** endpoints to represent a **collection of model instances**.

Provides a `get` method handler.

Extends: GenericAPIView, ListModelMixin

RetrieveAPIView

Used for **read-only** endpoints to represent a **single model instance**.

Provides a `get` method handler.

Extends: GenericAPIView, RetrieveModelMixin

DestroyAPIView

Used for **delete-only** endpoints for a **single model instance**.

Provides a `delete` method handler.

Extends: GenericAPIView, DestroyModelMixin

UpdateAPIView

Used for **update-only** endpoints for a **single model instance**.

Provides `put` and `patch` method handlers.

Extends: GenericAPIView, UpdateModelMixin

ListCreateAPIView

Used for **read-write** endpoints to represent a **collection of model instances**.

Provides `get` and `post` method handlers.

Extends: GenericAPIView, ListModelMixin, CreateModelMixin

RetrieveUpdateAPIView

Used for **read or update** endpoints to represent a **single model instance**.

Provides `get`, `put` and `patch` method handlers.

Extends: GenericAPIView, RetrieveModelMixin, UpdateModelMixin

RetrieveDestroyAPIView

Used for **read or delete** endpoints to represent a **single model instance**.

Provides `get` and `delete` method handlers.

Extends: GenericAPIView, RetrieveModelMixin, DestroyModelMixin

RetrieveUpdateDestroyAPIView

Used for **read-write-delete** endpoints to represent a **single model instance**.

Provides `get`, `put`, `patch` and `delete` method handlers.

Extends: GenericAPIView, RetrieveModelMixin, UpdateModelMixin, DestroyModelMixin

Customizing the generic views

Often you'll want to use the existing generic views, but use some slightly customized behavior. If you find yourself reusing some bit of customized behavior in multiple places, you might want to refactor the behavior into a common class that you can then just apply to any view or viewset as needed.

Creating custom mixins

For example, if you need to lookup objects based on multiple fields in the URL conf, you could create a mixin class like the following:

```
class MultipleFieldLookupMixin:
    """
    Apply this mixin to any view or viewset to get multiple field filtering
    based on a `lookup_fields` attribute, instead of the default single field filtering.
    """
    def get_object(self):
        queryset = self.get_queryset()             # Get the base queryset
        queryset = self.filter_queryset(queryset)  # Apply any filter backends
        filter = {}
        for field in self.lookup_fields:
            if self.kwargs[field]: # Ignore empty fields.
                filter[field] = self.kwargs[field]
        obj = get_object_or_404(queryset, **filter)  # Lookup the object
        self.check_object_permissions(self.request, obj)
        return obj
```

You can then simply apply this mixin to a view or viewset anytime you need to apply the custom behavior.

```
class RetrieveUserView(MultipleFieldLookupMixin, generics.RetrieveAPIView):
    queryset = User.objects.all()
    serializer_class = UserSerializer
    lookup_fields = ['account', 'username']
```

Using custom mixins is a good option if you have custom behavior that needs to be used.

Creating custom base classes

If you are using a mixin across multiple views, you can take this a step further and create your own set of base views that can then be used throughout your project. For example:

```
class BaseRetrieveView(MultipleFieldLookupMixin,
                       generics.RetrieveAPIView):
    pass


class BaseRetrieveUpdateDestroyView(MultipleFieldLookupMixin,
                                    generics.RetrieveUpdateDestroyAPIView):
    pass
```

Using custom base classes is a good option if you have custom behavior that consistently needs to be repeated across a large number of views throughout your project.

Serializers

Serializers allow complex data such as querysets and model instances to be converted to native Python datatypes that can then be easily rendered into `JSON`, `XML` or other content types. Serializers also provide deserialization, allowing parsed data to be converted back into complex types, after first validating the incoming data.

The serializers in REST framework work very similarly to Django's `Form` and `ModelForm` classes. We provide a `Serializer` class which gives you a powerful, generic way to control the output of your responses, as well as a `ModelSerializer` class which provides a useful shortcut for creating serializers that deal with model instances and querysets.

For more information please visit this link.

https://www.django-rest-framework.org/api-guide/serializers/