generics.py mixins.py

Generic views

> *Django's generic views... were developed as a shortcut for common usage patterns... They take certain common idioms and patterns found in view development and abstract them so that you can quickly write common views of data without having to repeat yourself.*
>
> — *Django Documentation*

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)
```

For very simple cases you might want to pass through any class attributes using the `.as_view()` method. For example, your URLconf might include something like the following entry:

```
path('users/', ListCreateAPIView.as_view(queryset=User.objects.all(), serializer_class=UserS
```

API Reference

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 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)
```

Other methods:

You won't typically need to override the following methods, although you might need to call into them if you're writing custom views using `GenericAPIView`.

- `get_serializer_context(self)` - Returns a dictionary containing any extra context that should be supplied to the serializer. Defaults to including `'request'`, `'view'` and `'format'` keys.
- `get_serializer(self, instance=None, data=None, many=False, partial=False)` - Returns a serializer instance.
- `get_paginated_response(self, data)` - Returns a paginated style `Response` object.
- `paginate_queryset(self, queryset)` - Paginate a queryset if required, either returning a page object, or `None` if pagination is not configured for this view.
- `filter_queryset(self, queryset)` - Given a queryset, filter it with whichever filter backends are in use, returning a new queryset.

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.get(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.

PUT as create

Prior to version 3.0 the REST framework mixins treated `PUT` as either an update or a create operation, depending on if the object already existed or not.

Allowing `PUT` as create operations is problematic, as it necessarily exposes information about the existence or non-existence of objects. It's also not obvious that transparently allowing re-creating of previously deleted instances is necessarily a better default behavior than simply returning `404` responses.

Both styles "`PUT` as 404" and "`PUT` as create" can be valid in different circumstances, but from version 3.0 onwards we now use 404 behavior as the default, due to it being simpler and more obvious.

If you need to generic PUT-as-create behavior you may want to include something like this `AllowPUTAsCreateMixin` class as a mixin to your views.

Third party packages

The following third party packages provide additional generic view implementations.

Django Rest Multiple Models

Django Rest Multiple Models provides a generic view (and mixin) for sending multiple serialized models and/or querysets via a single API request.
