# Pyramid Restful Framework Documentation

*Release 1.0.0*

Daniel Poland

May 24, 2018

## User Guide

A RESTful API framework for Pyramid heavily influenced by django-rest-framework.

The goal of this project is to provide DRF's view patterns on a lighter weight web framework that grants you more fine grained and explicit control over database queries and object serialization/deserialization. This is accomplished using SQLAlchemy as an ORM and marshmallow Schemas for object serialization and deserialization.

To get the most out of this documentation, and PRF in general, it is recommended that you first familiarize yourself with the following documentation:

- Pyramid
- Django Rest Framework
- SQLAlchemy
- Marshmallow

CHAPTER 1

# Quick Start

The quickest way to get started with pyramid-restful-framework is to use pyramid-cookiecutter-restful. The cookiecutter will scaffold a project that includes Pyramid, SQLAlchemy and pyramid-restful-framework. The project uses Django like settings instead of .ini files for configuration. It includes a wsgi.py file for running the app.

If you like ini files or want to include pyramid-restful-framework in an existing project you can install the library via pip.

```
$ pip install pyramid-restful-framework
```

Be sure to add **pyramid_restful** to the **pyramid.includes** directive in your configuration file(s).

```
pyramid.includes = pyramid_restful
```

Alternatively you can use *pyramid.config.Configurator.include* in your app setup:

```
config.include('pyramid_restful')
```

# CHAPTER 2

# Configuration

Currently there are three settings you can use to configure default behavior in PRF.

- **default_pagination_class**: A string representing the path to the default pagination class to use.
- **page_size**: An integer used as the default page size for pagination.
- **default_permission_classes**: A list or tuple of strings. Each string represents the path to a permissions class to use by default with each view.

If you used pyramid-cookiecutter-restful to create your project you can simply update these values in the `settings.__init__.py` file in the `PYRAMID_APP_SETTINGS` variable:

```
PYRAMID_APP_SETTINGS = {
    'pyramid.reload_templates': PYRAMID_RELOAD_TEMPLATES,
    'pyramid.debug_authorization': PYRAMID_DEBUG_AUTHORIZATION,
    'pyramid.debug_notfound': PYRAMID_DEBUG_NOTFOUND,
    'pyramid.debug_routematch': PYRAMID_DEBUG_ROUTEMATCH,
    'pyramid.default_locale_name': 'en',
    # pyramid_restful settings
    'restful.page_size': 50,
    'restful.default_pagination_class': 'pyramid_restful_jsonapi.pagination.
↪JSONAPIPagination',
    'restful.default_permission_classes': ['exampleapp.permissions.
↪AuthenticatedAndActivePermission'],
}
```

If you are adding PRF to an existing project or your prefer using ini files for configuration you can set the values for these configurations by adding a new `restful` section to you ini file:

```
[restful]
restful.page_size = 50
restful.default_pagination_class = 'pyramid_restful_jsonapi.pagination.
↪JSONAPIPagination'
restful.default_permission_classes = 'exampleapp.permissions.
↪AuthenticatedAndActivePermission'
```

# CHAPTER 3

# Class-based Views

The `APIView` serves as the base class for all views in PRF. It replaces the function based views often used in pyramid applications. Requests are passed to the view from the router and are dispatched to a method in the view with the same name as the HTTP method from the request. If the view class does not implement a method used by the request a 405 response is returned.

For example, in the class definition below a **GET** request would routed the class's `get()` method and a **POST** request would be routed to the class's `post()` method:

```
from pyramid.response import Response

from pyramid_restful.views import APIView

from .models import User

class UserView(ApiView):
    """
    A view to list all the users and to create a new User.
    """

    def get(self, request, *args, **kwargs):
        users = request.dbsession.query(User).all()
        return Response(json_body=users)

    def post(self, request, *args, **kwargs):
        user = User(**request.json_body)
        request.dbsession.add(user)
        return Response(status=201)
```

You route `APIView` classes similar to how you route typical views in pyramid. Below is an example `routes.py` file that routes the view defined above:

```
from . import views

def includeme(config):
```

(continues on next page)

(continued from previous page)

```
    config.add_route('users', '/users/')
    config.add_view(views.UserView.as_view(), route_name='users')
```

Any URL pattern matching variables used in the route definition will be passed to the view's method as a kwarg.:

```
class UserDetailView(ApiView):
    """
    Retrieve a specific User.
    """

    def get(self, request, id, *args, **kwargs):
        user = request.dbsession.query(User).get(id)
        return Response(json_body=user)



def includeme(config):
    config.add_route('users', '/users/{id}/')
    config.add_view(views.UserDetailView.as_view(), route_name='users')
```

## 3.1 Permissions

The `permission_classes` class attribute on `ApiView` controls which permissions are applied to incoming requests. By default, `permission_classes` is set to the value of the configuration variable `default_permission_classes`. See *Configuration* and *Permissions* for more details.

CHAPTER 4

# Generic Views

The key advantage of using class-based views is that it allows you to reuse common behavior across many views. PRF supplies you with a few pre-constructed views that provide commonly used functionality.

The GenericAPIView class allows you quickly compose an API while keeping your code DRY through class configuration rather than redefining view logic every time you need it.

**Examples**

Typically when you use a generic view all you need to do is set some of the class attributes.

```
from pyramid_restful import generics

from .models import User
from .schemas import UserSchema

class UserView(generics.ListCreateAPIView):
    model = User
    schema_class = UserSchema
```

That's all it takes. This provides the same functionality as the UserView created using the APIView class in the *Class-based Views* section. It provides two methods. One for **GET** requests, which returns all the Users, and one for **POST** requests, which allows you to add a new User.

In some cases the default behavior might not meet your needs. In those cases you can override the methods on the view class.

```
from pyramid.response import Response

from pyramid_restful import generics

from .models import User
from .schemas import UserSchema

class UserView(generics.ListCreateAPIView):
    model = User
```

(continues on next page)

(continued from previous page)

```
    schema_class = UserSchema

    def list(self, request, *args, **kwargs):
        rows = self.get_query().all()
        schema = UserSchema()
        data, errors = schema.dump(rows)

        return Response(data)
```

## 4.1 API Reference

**GenericAPIView**

This class extends *APIView* adding commonly used functionality for basic list and detail views. Full fledged API views are constructed by combining `GenericAPIView` with mixin classes. A few concrete generic views are provided by PRF. For a full list of these classes see the *Generics API* docs.

**Attributes**

**Basics:**

- `model`: The SQLAlchemy model that should be used for returning objects from the view. You must set this attribute or override the `get_query()` method.
- `schema_class`: The marshmallow Schema class to be used for validating and deserializing request data and for serializing response data.
- `lookup_field`: The field on the model used to identify individual instance of an model. Defaults to `'id'`.

**Pagination:**

- `pagination_class`: The pagination class that is used to paginate list results. This defaults to the value of the `restful.default_pagination_class` configuration, if set.

**Filtering:**

- `filter_classes`: An iterable of classes that extend `BaseFilter`. Filtering is pretty primative currently in PRF. Each class in the `filter_classes` iterable is passed the query used by the viewset before the query finally executed to produce the data for a response from the view.

CHAPTER 5

# ViewSets

A ViewSet is a class-based view that allows you to combine a set of related views into a single class. The most typical usage of ViewSets is to combine CRUD operations for a particular model in a single class. ViewSets allow you define methods that handle both detail and list operations in a single class. Unlike a `APIView` class that defines methods such as `get()` or `post()`, an `APIViewSet` defines actions like `retrive()` and `create()`.

## 5.1 Example

Below we define a single `APIViewSet` that can be used to retrieve a single user or all the users in the system:

```python
from pyramid.response import Response
from pyramid.httpexceptions import HTTPNotFound

from pyramid_restful import viewsets

from myapp.models import User
from myyapp.schemas import UserSchema

class UserViewSet(viewsets.APIViewSet):
    model = User
    schema = UserSchema

    def list(self, request):
        users = request.dbsession.query(User).all()
        schema = UserSchema()
        content = schema.dump(users, many=True)[0]
        return Response(json=content)


    def retrieve(self, request, id):
        user = request.dbsession.query(User).get(id)

        if not user:
```

(continues on next page)

(continued from previous page)

```
            raise HTTPNotFound()

        schema = UserSchema()
        content = schema.dump(user)[0]
        return Response(json=content)
```

To route this view in Pyramid we bind the view to two different routes:

```
from . import views


def includeme(config):
    config.add_route('user-list', '/users/')
    config.add_view(views.UserViewSet.as_view({'get': 'list'}), route_name='user-list
↪')

    config.add_route('user-detail', '/users/{id}/')
    config.add_view(views.UserViewSet.as_view({'get': 'retrieve'}), route_name='user-
↪detail')
```

Typically you wont do this. Instead you would use the *ViewSetRouter* to configure the routes for you:

```
from pyramid_restful.routers import ViewSetRouter

from . import views


def includeme(config):
    router = ViewSetRouter(config)
    router.register('users', views.UserViewSet, 'user')
```

## 5.2 ViewSet Actions

The `ViewSetRouter` provides defaults for the standard CRUD actions, as shown below:

```
class UserViewSet(viewsets.APIViewSet):
    """
    Example empty viewset demonstrating the standard
    actions that will be handled by a router class.
    """

    def list(self, request):
        pass

    def create(self, request):
        pass

    def retrieve(self, request, id=None):
        pass

    def update(self, request, id=None):
        pass

    def partial_update(self, request, id=None):
```

(continues on next page)

(continued from previous page)

```
        pass

    def destroy(self, request, id=None):
        pass
```

## 5.3 Including extra actions for routing

You can add ad-hoc methods to ViewSets that will automatically be routed by the `ViewSetRouter` by using the `@detail_route` or `@list_route` decorators. The `@detail_route` includes `id` in it's url pattern and is used for methods that operate on a single instance of model. `@list_route` decorator is used for methods that operate on many instances of a model.

Example:

```
from pyramid.response import Response

from pyramid_restful.viewsets import ModelCRPDViewSet
from pyramid_restful.decorators import list_route, detail_route

from .models import User
from .schemas import UserSchema


class UserViewSet(ModelCRPDViewSet):
    model = User
    schema = UserSchema

    @detail_route(methods=['post'])
    def lock(request, id):
        user = request.dbsession.query(User).get(id)

        if not user:
            raise HTTPNotFound()

        user.is_locked = True
        return Response(status=204)

    @list_route(methods=['get'])
    def active(request):
        users = request.dbsession.query(User).filter(User.is_active == True).all()
        schema = UserSchema()
        content = schema.dump(users, many=True)[0]
        return Response(json=content)
```

By default the router will append the name of method to the url pattern generated. The two decorated routes above would result in the following url patterns:

```
'/users/{id}/lock'
'/users/active'
```

You can override this behavior by setting the kwarg `url_path` on the decorator.

## 5.4 Base ViewSet Classes

Generally your not going to need to write your own viewsets. Instead you will use one of the base ViewSet classes provided by PRF or use a number of mixin classes in your ViewSet to compose a class that only includes the actions you need for a particular resource.

### 5.4.1 APIViewSet

The `APIViewSet` class extends the `APIView` class and does not provide any actions by default. You will have to add the action methods explicitly to the class. You can use the standard `APIView` attributes such as `permissions`.

### 5.4.2 GenericAPIViewSet

The `GenericAPIViewSet` class extends `GenericAPIView` and does not provide any actions by default, but does include the base set of generic view behavior, such as the `get_object()` and `get_query()` methods. To use the class you will typically mixin the actions you need from the mixins module or write the action methods explicitly.

### 5.4.3 The ModelViewSets

PRF provide you with several ModelViewSet implementations. ModelViewSets are simply classes in which several action mixins are combined with `GenericAPIViewSet`. They provide all the functionality that comes with a `GenericAPIView`, such as the `filter_classes` and `permission_classes` attributes and well as the `get_query()` and `get_object()` methods. The base ModelViewSets provided by PRF along with their default actions are listed below:

- ReadOnlyModelViewSet: `list()`, `retrieve()`
- ModelCRUDViewSet: `list()`, `create()`, `retrieve()`, `update()`, `destroy()`
- ModelCRPDViewSet: `list()`, `create()`, `retrieve()`, `partial_update()`, `destroy()`
- ModelCRUPDViewSet: `list()`, `create()`, `retrieve()`, `update()`, `partial_update()`, `destroy()`

### 5.4.4 Custom ViewSets

If one of the predefined ViewSets doesn't meet your needs you can always compose your own ViewSet and override its actions.

Example:

```
from pyramid_restful import mixins
from pyramid_restful import viewsets

from .models import User
from .schema import UserSchema


class UserViewSet(mixins.CreateModelMixin,
                  mixins.RetrieveModelMixin,
                  mixins.UpdateModelMixin):

    model = User
```

(continues on next page)

(continued from previous page)

```
    schema = UserSchema

    def get_query():
        """
        Restrict user to the authenticated user.
        """

        return super(UserViewSet, self).get_query() \
            .filter(User.id == request.user.id)
```

# CHAPTER 6

# Permissions

PRF offers a single base class for writing your own permissions, `BasePermission`. There are two methods that you can override, `has_permission()` and `has_object_permission()`. The first is checked on every request to a view and the later is checked when a specific instance of an object is being accessed in a view.

In the example below the request's authenticated user must be an admin:

```python
from pyramid.response import Response

from pyramid_restful.viewsets import ModelCRPDViewSet
from pyramid_restful.permissions import BasePermission

from .models import User
from .schemas import UserSchema

class IsAdminPermission(BasePermission):
    message = 'You must be an admin.'

    def has_permission(self, request, view):
        return request.user.is_admin == True:


class UserViewSet(ModelCRPDViewSet):
    model = User
    schema = UserSchema
    permission_classes = (IsAdminPermission,)
```

If you prefer you can still use pyramid's built in authorization and permissions framework. If you are manually routing a view and using pyramid's authorization framework you would use permissions just as you would normally:

```python
# config is an instance of pyramid.config.Configurator
config.add_route('users', '/users/')
config.add_view(views.UserView.as_view(), route_name='user', permission='view')
```

If you are routing a `ViewSet` and using a `ViewSetRouter` you simply set your permission using the `permission` kwarg:

```
from pyramid.routers import ViewSetRouter

def includeme(config):
router = ViewSetRouter(config)
router.register('users', views.UserViewSet, 'coop', permission='view')
```

# CHAPTER 7

# Filters

PRF comes with very simple filter functionality. This functionality will likely be improved in the future. As outlined in the *Class-based Views* documentation you can attach several filter classes to a class that extends `GenericAPIView` by using the `filter_classes` class attribute. All filter classes must extend the `BaseFilter` class. PRF comes with a few predefined filter classes outlined below.

## 7.1 FieldFilter

The `FieldFilter` class allows you to filter your request's query by using query string parameters. The query string parameters on the request should be formatted as `filter[field_name]=val`. Comma-separated values are treated as ORs. Multiple filter query params are AND'd together.

For example given the the `ViewSet` definition below and a request with the url of `https://api.mycoolapp.com/users/?filter[account_id]=1`. The ViewSet would filter the query of users by `User.account_id` where the value was `1`.

```
class UserViewSet(ModelCRUDViewSet):
        model = User
        schema = UserSchema
        filter_classes = (FieldFilter,)
        filter_fields = (User.account_id, User.email, User.name,)
```

## 7.2 SearchFilter

The `SearchFilter` class allows you to filter your request's query by using `LIKE` statements. Comma separated values are treated as ORs. Multiple search query parameters are OR'd together. (Note: this works differently than multiple search query parameters use for FieldFilters.) The values are transformed into their all lower-case representation before the comparision is applied.

For example given the the `ViewSet` definition below and a request with the url of `https://api.mycoolapp.com/users/?search[email]=gmail,hotmail`. The ViewSet would filter the query of users with the a statement similar to: `WHERE (user.email LIKE '%gmail%' OR user.email LIKE '%hotmail%').`

```
class UserViewSet(ModelCRUDViewSet):
        model = User
        schema = UserSchema
        filter_classes = (SearchFilter,)
        filter_fields = (User.email,)
```

## 7.3 OrderFilter

The `OrderFilter` class allows you to order your request's query results by the fields specified in the query string parameters. The value of the query string parameter indicates the direction of the ordering. Either `asc` or `desc`. If multiple ordering query string parameters are used, the order in which they are used will determine the order in which they are applied for ordering.

For example given the the `ViewSet` definition below and a request with the url of `https://api.mycoolapp.com/users/?order[name]=asc&order[created_at]=desc`. The ViewSet would order the results returned in the response by the `User.name` field in ascending order, then `User.created_at` field in descending order.

```
class UserViewSet(ModelCRUDViewSet):
        model = User
        schema = UserSchema
        filter_classes = (OrderFilter,)
        filter_fields = (User.name, User.created_at,)
```

# CHAPTER 8

# Expandables

Using query string parameters, the expandables mixins allow you to dynamically and efficiently control the expansion of relationships in the objects returned from your views. Both the `ExpandableViewMixin` and `ExpandableSchemaMixin` can be used independently but are most effective when used together.

## 8.1 ExpandableViewMixin

The `ExpandableViewMixin` allows you to dynamically control the joins executed by your view for each request. Using the `expandable_fields` class attribute you can configure query joins and options based on the values passed into the `expand` query string parameter. This allows you to avoid inadvertently executing extra queries that expand relationships for each object returned from your view. See the *expandables API* documentation for more information on using the `expandable_fields` attribute.

Example:

```
from sqlalchemy.orm import subqueryload

from pyramid_restful import viewsets
from pyramid_restful.expandables import ExpandableViewMixin


class AuthorViewSet(viewsets.CRUDModelViewSet,
                    ExpandableViewMixin):
    model = Author
    schema_class = AuthorSchema
    expandable_fields = {
        'books': {'options': [subqueryload(Author.books)]
    }
```

Using the example class above, a request with a query string of `?expand=books` would results in `subqueryload(Author.books)` being passed to the `.options()` method on the SQLAlchemy query executed by the view. This effectively performs a single query for all of the books related to the authors returned by the view, which prevents performing individual quries to retrieve the books for each author returned when the authors are serialized.

## 8.2 ExpandableSchemaMixin

The `ExpandableSchemaMixin` is a mixin class for `marshmallow.Schema` classes. It supports optionally including `Nested` fields based on the value of the query string parameters. The query string parameter's key is determined by the value of the `QUERY_KEY` class attribute.

Fields that can be expanded are defined in the schema's Meta class using the `expandable_fields` attribute. The value of `expandable_fields` should be a dictionary. That dictionary's keys should match both the name of the model's relationship being expanded, and the query string parameter in from the request. The dictionary's values should be a `marshmallow.fields.Nested` definition.

Example:

```
from marshmallow import Schema, fields

from pyramid_restful.expandables import ExpandableSchemaMixin

class AuthorSchema(ExpandableSchemaMixin, schema)
    id = fields.Integer()
    name = fields.String()

    class Meta:
        expandable_fields = {
            'books': fields.Nested('BookSchema')
        }
```

# CHAPTER 9

# Pagination

PRF has built in support for pagination. You can set the default pagination class for you project using the `restful.default_pagination_class` setting. The pagination class can also be set on a per view settings using the `pagination_class` class attribute. PRF supports two styles of pagination out of the box, `PageNumberPagination` and `LinkHeaderPagination` pagination. You can find the details about these pagination classes in the *pagination* section of the API docs.

## 9.1 Custom Pagination Classes

To create you own pagination classes simply extend the `BasePagination` class and implement the `paginate_query()` and `get_paginated_response()` methods.

CHAPTER 10

# views

**class** `pyramid_restful.views.`**`APIView`**(***kwargs*)

Base for class based views. Requests are routed to a view's method with the same name as the HTTP method of the request.

**`permission_classes = []`**

An iterable of permissions classes. Defaults to `default_permission_classes` from the pyramid_restful configuration. Override this attribute to provide view specific permissions.

**`initial`**(*request*, **args*, ***kwargs*)

Runs anything that needs to occur prior to calling the method handler.

**`get_permissions`**()

Instantiates and returns the list of permissions that this view requires.

**`check_permissions`**(*request*)

Check if the request should be permitted. Raises an appropriate exception if the request is not permitted.

**Parameters** **`request`** – Pyramid Request object.

**`check_object_permissions`**(*request*, *obj*)

Check if the request should be permitted for a given object. Raises an appropriate exception if the request is not permitted.

**Parameters**

- **`request`** – Pyramid Request object.
- **`obj`** – The SQLAlchemy model instance that permissions will be evaluated against.

**`options`**(*request*, **args*, ***kwargs*)

Handles responding to requests for the OPTIONS HTTP verb.

CHAPTER 11

# generics

**class** pyramid_restful.generics.**GenericAPIView**(***kwargs*)

Provide default functionality for working with RESTFul endpoints. pagination_class can be overridden as a class attribute:

Usage:

```
class MyView(GenericAPIView):
    pagination_class = MyPager
```

**pagination_class**
alias of `pyramid_restful.pagination.pagenumber.PageNumberPagination`

**model = None**
The SQLAlchemy model class used by the view.

**schema_class = None**
The marshmallow schema class used by the view.

**filter_classes = ()**
Iterable of Filter classes to be used with the view.

**lookup_field = 'id'**
The name of the primary key field in the model used by the view.

**get_query**()
Get the list of items for this view. You may want to override this if you need to provide different query depending on the incoming request. (Eg. return a list of items that is specific to the user)

**Returns** `sqlalchemy.orm.query.Query`

**get_object**()
Returns the object the view is displaying. You may want to override this if you need to provide non-standard queryset lookups. Eg if objects are referenced using multiple keyword arguments in the url conf.

**Returns** An instance of the view's model.

**get_schema_class**()
Return the class to use for the schema. Defaults to using *self.schema_class*. You may want to override this if you need to provide different serializations depending on the incoming request.

**get_schema_context**()
Extra context provided to the schema class.

**get_schema**(**args*, ***kwargs*)
Return the schema instance that should be used for validating and deserializing input, and for serializing output.

**filter_query**(*query*)
Filter the given query using the filter classes specified on the view if any are specified.

**paginator**
The paginator instance associated with the view, or *None*.

**paginate_query**(*query*)
Return single page of results or *None* if pagination is disabled.

**get_paginated_response**(*data*)
Return a paginated style `Response` object for the given output data.

**class** `pyramid_restful.generics.`**CreateAPIView**(***kwargs*)
Concrete view for creating a model instance.

**class** `pyramid_restful.generics.`**ListAPIView**(***kwargs*)
Concrete view for listing a queryset.

**class** `pyramid_restful.generics.`**RetrieveAPIView**(***kwargs*)
Concrete view for retrieving a model instance.

**class** `pyramid_restful.generics.`**DestroyAPIView**(***kwargs*)
Concrete view for deleting a model instance.

**class** `pyramid_restful.generics.`**UpdateAPIView**(***kwargs*)
Concrete view for updating a model instance.

**class** `pyramid_restful.generics.`**ListCreateAPIView**(***kwargs*)
Concrete view for listing a queryset or creating a model instance.

**class** `pyramid_restful.generics.`**RetrieveUpdateAPIView**(***kwargs*)
Concrete view for retrieving, updating a model instance.

**class** `pyramid_restful.generics.`**RetrieveDestroyAPIView**(***kwargs*)
Concrete view for retrieving or deleting a model instance.

**class** `pyramid_restful.generics.`**RetrieveUpdateDestroyAPIView**(***kwargs*)
Concrete view for retrieving, updating or deleting a model instance.

# CHAPTER 12

# viewsets

**class** `pyramid_restful.viewsets.`**`ViewSetMixin`**

Overrides `.as_view()` so that it takes an `actions_map` keyword that performs the binding of HTTP methods to actions on the view.

For example, to create a concrete view binding the 'GET' and 'POST' methods to the 'list' and 'create' actions…

view = MyViewSet.as_view({'get': 'list', 'post': 'create'})

**classmethod as_view**(*action_map=None*, ***initkwargs*)

Allows custom request to method routing based on given `action_map` kwarg.

**class** `pyramid_restful.viewsets.`**`APIViewSet`**(***kwargs*)

Does not provide any actions by default.

**class** `pyramid_restful.viewsets.`**`GenericAPIViewSet`**(***kwargs*)

The GenericAPIView class does not provide any actions by default, but does include the base set of generic view behavior, such as the `get_object` and `get_query` methods.

**class** `pyramid_restful.viewsets.`**`ReadOnlyModelViewSet`**(***kwargs*)

A ViewSet that provides default `list()` and `retrieve()` actions.

**class** `pyramid_restful.viewsets.`**`ModelCRUDViewSet`**(***kwargs*)

A ViewSet that provides default `create()`, `retrieve()`, `update()`, `destroy()` and `list()` actions.

**class** `pyramid_restful.viewsets.`**`ModelCRPDViewSet`**(***kwargs*)

A ViewSet that provides default `create()`, `retrieve()`, `partial_update()`, `destroy()` and `list()` actions.

**class** `pyramid_restful.viewsets.`**`ModelCRUPDViewSet`**(***kwargs*)

A viewset that provides default `create()`, `retrieve()`, `partial_update()`, `'update()`, `destroy()` and `list()` actions.

# CHAPTER 13

# mixins

**class** pyramid_restful.mixins.**ListModelMixin**
List objects.

**class** pyramid_restful.mixins.**RetrieveModelMixin**
Retrieve a single object.

**class** pyramid_restful.mixins.**CreateModelMixin**
Create object from serialized data.

**perform_create**(*data*)
Hook for controlling the creation of an model instance. Override this if you need to do more with your data before saving your object than just mapping the deserialized data to a new instance of `self.model`.

**class** pyramid_restful.mixins.**UpdateModelMixin**
Update a model instance (PUT).

**perform_update**(*data*, *instance*)
Hook for controlling the update of an model instance. Override this if you need to do more with your data before updating the object than just mapping the deserialized data to the attribute of the instance.

**class** pyramid_restful.mixins.**PartialUpdateMixin**
Support for partially updating instance (PATCH).

**perform_partial_update**(*data*, *instance*)
Hook for controlling the update of an model instance. Override this if you need to do more with your data before updating the object than just mapping the deserialized data to the attribute of the instance.

**class** pyramid_restful.mixins.**DestroyModelMixin**
Destroy a model instance.

**perform_destroy**(*instance*)
Hook for controlling the deletion of an model instance. Override this if you need to do more than just delete the instance.

**class** pyramid_restful.mixins.**ActionSchemaMixin**
Allows you to use different schema depending on the action being taken by the request. Defaults to the standard schema_class if no actions are specified.

CHAPTER 14

# decorators

pyramid_restful.decorators.**detail_route**(*methods=None*, ***kwargs*)

Used to mark a method on a ViewSet that should be routed for detail requests.

Usage:

```
class UserViewSet(ModelCRUDViewSet):
    model = User
    schema = UserSchema

    @detail_route(methods=['post'], url_path='lock-user')
    def lock_user(request, id):
        ...
```

**Parameters**

- **methods** – An iterable of strings representing the HTTP (GET, POST, etc.) methods accepted by the route.
- **url_path** – Replaces the route automatically generated by the ViewSetRouter for the decorated method with the value provided.

pyramid_restful.decorators.**list_route**(*methods=None*, ***kwargs*)

Used to mark a method on a ViewSet that should be routed for list requests.

Usage:

```
class UserViewSet(ModelCRUDViewSet):
    model = User
    schema = UserSchema

    @list_route(methods=['get'], url_path='active-users')
    def active_users(request, *args, **kwargs):
        ...
```

**Parameters**

- **methods** – An iterable of strings representing the HTTP (GET, POST, etc.) methods accepted by the route.
- **url_path** – Replaces the route automatically generated by the ViewSetRouter for the decorated method with the value provided.

# CHAPTER 15

# routers

**class** pyramid_restful.routers.**ViewSetRouter**(*configurator*, *trailing_slash=True*)

Automatically adds routes and associates views to the Pyramid `Configurator` for ViewSets, including any decorated `list_routes` and `detail_routes`.

**register**(*prefix*, *viewset*, *basename*, *factory=None*, *permission=None*)

Factory and permission are likely only going to exist until I have enough time to write a permissions module for PRF.

**Parameters**

- **prefix** – the uri route prefix.
- **viewset** – The ViewSet class to route.
- **basename** – Used to name the route in pyramid.
- **factory** – Optional, root factory to be used as the context to the route.
- **permission** – Optional, permission to assign the route.

# CHAPTER 16

# permissions

**class** `pyramid_restful.permissions.`**`BasePermission`**

All permission classes should inherit from this class.

**`message = None`**

Override message to customize the message associated with the exception.

**`has_permission`**(*request*, *view*)

Checked on every request to a view. Return `True` if permission is granted else `False`.

**Parameters**

- **`request`** – The request sent to the view.
- **`view`** – The instance of the view being accessed.

**Returns** Boolean

**`has_object_permission`**(*request*, *view*, *obj*)

Checked when a request is for a specific object. Return `True` if permission is granted else `False`.

**Parameters**

- **`request`** – The request sent to the view.
- **`view`** – The instance of the view being accessed.
- **`obj`** – The object being accessed.

**Returns** Boolean

# CHAPTER 17

# filters

**class** pyramid_restful.filters.**BaseFilter**

Base interface that that all filter classes must implement.

**filter_query** (*request*, *query*, *view*)

This method must be overridden.

**Parameters**

- **request** – The request being processed.
- **query** – The query to be filtered.
- **view** – The view the filter is being applied to.

**Returns** The filtered `query`.

**class** pyramid_restful.filters.**AttributeBaseFilter**

A base class for implementing filters on SQLAlchemy model attributes. Supports filtering a comma separated list using OR statements and relationship filter using the . path to attribute. WARNING: Every relationship in a `.` path is joined.

Expects the query string parameters to be formatted as: `key[field_name]=val`.

Example: `filter[email]=test@exmaple.com`

**query_string_lookup = None**

The key to use when parsing the request's query string. The key in `key[field_name]=val`.

**view_attribute_name = None**

The name of the class attribute used in the view class that uses the filter that specifies which fields can be filtered on.

**parse_query_string** (*params*)

Override this method if you need to support query string filter keys other than those in the format of `key[field_name]=val`. Maps query string values == 'null' to `None`.

**Parameters params** – The query string parameters from `request.params`.

**Returns** Dictionary.

**filter_query** (*request*, *query*, *view*)
You may want to override this method if you want to add custom filtering to an ViewSet while still utilizing the feature of the `AttributeFilter` implementation.

**Parameters**

- **request** – The pyramid `Request` instance.
- **query** – The SQLAlchemy `Query` instance.
- **view** – An instance of the view class that the filter has been applied to.

**Returns** The filtered query.

**apply_filter** (*query*, *filter_list*)
Override this if you need to do something beside calling filter on the query.

**Parameters**

- **query** – the query that will be returned from the filter_query method.
- **filter_list** – An array of SQLAlchemy comparative statements.

**Returns** The query.

**build_comparision** (*field*, *value*)
Must be overridden. Given the model field and the value to be filtered, this should return the statement to be appended as a filter to the final query.

**class** pyramid_restful.filters.**FieldFilter**
Filters a query based on the `filter_fields` set on the view. `filter_fields` should be a list of SQLAlchemy Model columns.

Comma separated values are treated as ORs. Multiple filter[<field>] query params are AND'd together.

**Usage**:

```
class UserViewSet(ModelCRUDViewSet):
    model = User
    schema = UserSchema
    filter_classes = (FieldFilter,)
    filter_fields = (User.email, User.name,)
```

**build_comparision** (*field*, *value*)
Must be overridden. Given the model field and the value to be filtered, this should return the statement to be appended as a filter to the final query.

**class** pyramid_restful.filters.**SearchFilter**
Implements LIKE filtering based on the search[field_name]=val querystring. Comma separated values are treated as ORs. Multiple search[<fields>] are OR'd together.

**Usage**:

```
class UserViewSet(ModelCRUDViewSet):
    model = User
    schema = UserSchema
    filter_classes = (SearchFilter,)
    filter_fields = (User.email, User.name,)
```

**build_comparision** (*field*, *value*)
Must be overridden. Given the model field and the value to be filtered, this should return the statement to be appended as a filter to the final query.

**apply_filter**(*query*, *filter_list*)
Override this if you need to do something beside calling filter on the query.

**Parameters**

- **query** – the query that will be returned from the filter_query method.
- **filter_list** – An array of SQLAlchemy comparative statements.

**Returns** The query.

**class** pyramid_restful.filters.**OrderFilter**
Allow ordering of the query based on an order[field]=(asc || desc) query string.

**Usage**:

```
class UserViewSet(ModelCRUDViewSet):
    model = User
    schema = UserSchema
    filter_classes = (OrderFilter,)
    filter_fields = (User.created, User.name,)
```

**build_comparision**(*field*, *value*)
Must be overridden. Given the model field and the value to be filtered, this should return the statement to be appended as a filter to the final query.

**apply_filter**(*query*, *filter_list*)
Override this if you need to do something beside calling filter on the query.

**Parameters**

- **query** – the query that will be returned from the filter_query method.
- **filter_list** – An array of SQLAlchemy comparative statements.

**Returns** The query.

# CHAPTER 18

# expandables

**class** pyramid_restful.expandables.**ExpandableSchemaMixin**

A mixin class for `marshmallow.Schema` classes. Supports optionally expandable fields based on the value of the query string parameters. The query string parameter's key is determined by the value of the `QUERY_KEY` class attribute.

Fields that can be expanded are defined in the schema's Meta class using the `expandable_fields` attribute. The value of `expandable_fields` should be a dictionary who's keys are used to match the value of the requests's query string parameter and the value should be a `marshmallow.fields.Nested` definition.

**Usage**:

```
from marshmallow import Schema, fields

from pyramid_restful.expandables import ExpandableSchemaMixin

class UserSchema(ExpandableSchemaMixin, schema)
    id = fields.Integer()
    name = fields.String()
    email = fields.String()

    class Meta:
        expandable_fields = {
        'account': fields.Nested('AccountSchema')
    }
```

**OPTIONS_CLASS**

alias of `ExpandableOpts`

**QUERY_KEY = 'expand'**

The query string parameter name used for expansion.

**class** pyramid_restful.expandables.**ExpandableViewMixin**

Optionally used to allow more fine grained control over the query used to pull data. `expandable_fields` should be a dictionary of key = the field name that is expandable and val = a dict with the following keys.

- **join (optional)**: A table column to join() to the query.

- **outerjoin (optional)**: A table column to outerjoin() to the query.
- **options (optional)**: A list passed to the constructed queries' options method. This is where you want to include the related objects to expand on. Without a value you here you will likely end up running lots of extra queries.

Example:

```
expandable_fields = {
    'author': {'join': Book.author, 'options': [joinedload(Book.author)]
}
```

**expandable_fields = None**
A dictionary of the fields can be expanded. Its definition is described above.

**get_query**()
If you override this method do not forget to call `super()`.

CHAPTER 19

# pagination

**class** pyramid_restful.pagination.**BasePagination**

The base class each Pagination class should implement.

**paginate_query**(*query*, *request*)

**Parameters**

- **query** – SQLAlchemy `query`.
- **request** – The request from the view

**Returns** The paginated date based on the provided query and request.

**get_paginated_response**(*data*)

**Parameters** **data** – The paginated data.

**Returns** A response containing the paginated data.

**class** pyramid_restful.pagination.**PageNumberPagination**

A simple page number based style that supports page numbers as query parameters.

For example:

```
http://api.example.org/accounts/?page=4
http://api.example.org/accounts/?page=4&page_size=100
```

page_size can be overridden as class attribute:

```
class MyPager(PageNumberPagination):
    page_size = 10
```

The resulting response JSON has four attributes, count, next, previous and results. Count indicates the total number of objects before pagination. Next and previous contain URLs that can be used to retrieve the next and previous pages of date respectively. The results attribute contains the list of objects that belong to page of data.

Example:

```
{
    'count': 50,
    'next': 'app.myapp.com/api/users?page=3',
    'previous': 'app.myapp.com/api/users?page=1',
    'results': [
        {id: 4, 'email': 'user4@myapp.com', 'name': 'John Doe'},
        {id: 5, 'email': 'user5@myapp.com', 'name': 'Jan Doe'}
    ]
}
```

**paginate_query**(*query*, *request*)

**Parameters**

- **query** – SQLAlchemy `query`.
- **request** – The request from the view

**Returns** The paginated date based on the provided query and request.

**get_paginated_response**(*data*)

**Parameters** **data** – The paginated data.

**Returns** A response containing the paginated data.

**get_url_root**()

Override this if you need a different root url. For example if the app is behind a reverse proxy and you want to use the original host in the X-Forwarded-Host header.

**class** pyramid_restful.pagination.**LinkHeaderPagination**

Add a header field to responses called Link. The value of the Link header contains information about traversing the paginated resource. For more information about link header pagination checkout githhub's great explanation: https://developer.github.com/v3/guides/traversing-with-pagination/

**get_paginated_response**(*data*)

**Parameters** **data** – The paginated data.

**Returns** A response containing the paginated data.

# CHAPTER 20

# Indices and tables

- genindex
- modindex
- search

# Python Module Index

# Index