django-rest-easy Documentation

Release 0.1

SMARTPAGER SYSTEMS INC.

Jun 13, 2019

Contents

CHAPTER 1

Introduction

Django-rest-easy is an extension to Django Rest Framework providing QOL improvements to serializers and views that introduce a more coherent workflow for creating REST APIs:

- Versioning and referencing serializers by model and schema, along with autoimport, so your serializers will be available anywhere, as long as you know the model and schema.
- A *rest_easy.fields.StaticField* for adding static data (independent of instance) to serializers.
- Creating views and viewsets using model and schema (it will automatically obtain serializer and queryset, although you can override both with usual DRF class-level parameters).
- A serializer override for a particular DRF verb, like create or update: no manual get_serialize_class override, no splitting ViewSets into multiple views.
- Scoping views' querysets and viewsets by url kwargs or request object parameters. Fore example, when you want to limit messages to a particular thread or threads to currently logged in user.
- Adding your own base classes to *GenericView* and your own mixins to all resulting generic view classes, like *ListCreateAPIView*.
- Chaining views' *perform_update* and *perform_create*: they by default pass **kwargs to *serializer.save()* now.
- A helper mixin that enables serializing Django model instances with just an instance method call.
- A helper methods that find serializer class and deserialize a blob of data, since oftentimes you will not know what exact data you will receive in a particular endpoint, especially when dealing with complex integrations.

All of the above are possible in pure DRF, but usually introduce a lot of boilerplate or aren't very easy or straightforward to code Therefore, at Telmediq we decided to open source the package that helps make our API code cleaner and more concise.

CHAPTER 2

Installation

Django-rest-easy is available on PyPI. The simplest way to install it is by running *pip install django-rest-easy*. Afterwards you need to add rest_easy to Django's *INSTALLED_APPS*:

```
INSTALLED_APPS = (
    # ...
    'rest_framework',
    'rest_easy',
    # ...
)
```

To make your serializers registered and working well with django-rest-easy's views, make sure they are autoimported. You can do that either by importing them in *app.serializers* module or modifying *REST_EASY_AUTOIMPORT_SERIALIZERS_FROM* setting to include your serializer location. For example, if you place your serializers in *app.api.serializers*, you should add the following to your settings file:

```
REST_EASY_AUTOIMPORT_SERIALIZERS_FROM = ['api.serializers']
```

Also, change your serializers to inherit from *rest_easy.serializers.Serializer* or *rest_easy.serializers.ModelSerializer* instead of default DRF serializers. Same goes for views - you should be using this:

```
from rest_easy.views import *
```

Instead of

```
from rest_framework.generics import *
```

Additionally, the following settings can alter the behaviour of the package:

- REST_EASY_AUTOIMPORT_SERIALIZERS_FROM - specify modules or packages that rest-easy will try to import serializers from when AppConfig is ready. The import is app-based, so it will search for serializers in all installed apps. By default *['serializers']*
- REST_EASY_VIEW_BASES - the mixins that should go into all views near the end of the mro (method resolution order). They will be placed before all DRF and django-rest-easy's bases, and after all generic mixins

from DRF.

- REST_EASY_GENERIC_VIEW_MIXINS - the mixins that should go into all generic views at the beginning of the mro (that means CreateAPIView, ListAPIView, RetrieveAPIView, DestroyAPIView, UpdateAPIView, ListCreateAPIView, RetrieveUpdateAPIView, RetrieveDestroyAPIView, RetrieveUpdateDestroyAPIView, ReadOnlyModelViewSet, ModelViewSet).
- REST_EASY_SERIALIZER_CONFLICT_POLICY - either 'allow' or 'raise'. What should happen when you redeclare a serializer with same model and schema - either the new one will be used or an error will be raised. By default 'allow' to not break applications with weird imports.

Because you usually won't be able to import the bases directly in settings, they should be given using class location strings (as is often the case in Django):

```
REST_EASY_VIEW_BASES = ['myapp.mixins.GlobalBase']
REST_EASY_GENERIC_VIEW_MIXINS = ['myapp.mixins.SuperMixin', 'myotherapp.mixins.
↪WhatIsItMixin']
```

They will be prepended to base class lists preserving their order. Please make sure that you are not importing django-rest-easy views before the mixins are ready to import (so before *AppConfig.ready* is called, for good measure).

CHAPTER 3

Basic usage

A minimal example to showcase what you can do would be:

```
from django.conf.urls import include, url
from rest_framework.routers import DefaultRouter

from rest_easy.serializers import ModelSerializer
from rest_easy.views import ModelViewSet
from rest_easy.scopes import UrlKwargScopeQuerySet
from rest_easy.tests.models import Account, User

class UserSerializer(ModelSerializer):
    class Meta:
        model = User
        schema = 'default'
        fields = '__all__'

class UserViewSet(ModelViewSet):
    model = User
    schema = 'default'
    lookup_url_kwarg = 'pk'
    scope = UrlKwargScopeQuerySet(Account)

router = DefaultRouter()
router.register(r'accounts/(?P<account_pk>\d+)/users', UserViewSet)

urlpatterns = [url(r'^', include(router.urls))]
```

CHAPTER 4

Detailed usage

4.1 Serializers

Django-rest-easy serializer bases (*rest_easy.serializers.Serializer* and *rest_easy.serializers.ModelSerializer*) are registered on creation and provide some consistency constraints: each serializer needs to have model and schema set in its Meta. Schema needs to be a string, while model should be a Django model subclass or explicit *None*. Both of those properties are required to be able to register the serializer properly. Both are also appended to serializer's fields as *rest_easy.fields.StaticField*. They will be auto-included in *Meta.fields* when necessary (ie. fields is not *__all__*):

```
class UserSerializer(ModelSerializer):
    class Meta:
        model = User
        schema = 'default'
        fields = '__all__'
```

Serializers can be obtained easily from *rest_easy.registers.SerializerRegister* (or, already instantiated, *rest_easy.registers.serializer_register*) like so:

```
from rest_easy.registers import serializer_register

serializer = serializer_register.get('myapp.mymodel', 'default-schema')
# or
from myapp.models import MyModel
serializer = serializer_register.get(MyModel, 'default-schema')
# or
serializer = serializer_register.get(None, 'modelless-schema')
```

This feature is leveraged heavily by django-rest-easy's views. Please remember that serializers need to be imported in order to be registered - it's best achieved by using the auto-import functionality described in the installation section.

As for the *rest_easy.fields.StaticField*, it can be used as such:

```
class UserSerializer(ModelSerializer):
    class Meta:
        model = User
        schema = 'default'
        fields = '__all__'
    static_data = StaticField(value='static_value')
```

4.2 Views

Views and viewsets provide a few additional features, allowing you to not specify *queryset* and *serializer_class* properties by default. If they are specified, though, they take priority over any logic provided by django-rest-easy.

- Providing *serializer_class* will disable per-verb custom serializers. It will make the view act basically as regular DRF view.
- *queryset* property doesn't disable any functionality. By default it is set to *model.objects.all()*, where model is provided as a class property, but it can be overridden at will without messing with django-rest-easy's functionality.

Overall using serializer_class on django-rest-easy views is not recommended.

A view example showing available features:

```
class UserViewSet(ModelViewSet):
    model = User
    schema = 'default'
    serializer_schema_for_verb = {'update': 'schema-mutate', 'create': 'schema-mutate
↪'}
    lookup_url_kwarg = 'pk'
    scope = UrlKwargScopeQuerySet(Account)

    def perform_update(self, serializer, **kwargs):
        kwargs['account'] = self.get_account()
        return super(UserViewSet, self).perform_update(serializer, **kwargs)

    def perform_create(self, serializer, **kwargs):
        kwargs['account'] = self.get_account()
        return super(UserViewSet, self).perform_create(serializer, **kwargs)
```

We're setting *User* as model, so the inferred queryest will be *User.objects.all()*. When a request comes in, a proper serializer will be selected:

- If the DRF dispatcher will call update or create methods, we will use serializer obtained by calling *serializer_register.get(User, 'schema-mutate')*.
- Otherwise the default schema will be used, so *serializer_register.get(User, 'default')*.

Additionally we're scoping the Users by account. In short, that means (by default - more on that in the section below) that our base queryset is modified with:

```
queryset = queryset.filter(account=Account.objects.get(pk=self.kwargs.get('account_pk
↪')))
```

Also, helper methods are provided for each scope that doesn't disable it:

```
def get_account(self):
    return Account.objects.get(pk=self.kwargs.get('account_pk'))
```

Technically, they are implemented with *__getattr__*, but each scope which doesn't have get_object_handle set to None will provide a get_X method (like *get_account* above) to obtain the object used for filtering. The object is kept cached on the view instance, so it can be reused during request handling without additional database queries. If the get_X method would be shadowed by something else, all scoped object are available via *view.get_scoped_object*:

```
def perform_create(self, serializer, **kwargs):
    kwargs['account'] = self.get_scoped_object('account')
    return super(UserViewSet, self).perform_create(serializer, **kwargs)
```

This follows standard Django convention of naming foreign keys by *RelatedModel._meta.model_name* (same as scoped object access on view), using pk as primary key and modelname_pk as url kwarg. All of those parameters are configurable (see Scopes section below).

For more complex cases, you can provide a list of scopes instead of a single scope. All of them will be applied to the queryset.

Now let's say all your models need to remember who modified them recently. You don't really want to pass the logged in user to serializer in each view, and using threadlocals or globals isn't a good idea for this type of task. The solution to this problem would be a common view mixin. Let's say we place this in *myapp.mixins.py*:

```
class InjectUserMixin(object):
    def perform_update(self, serializer, **kwargs):
        kwargs['user'] = self.request.user
        return super(UserViewSet, self).perform_update(serializer, **kwargs)

    def perform_create(self, serializer, **kwargs):
        kwargs['user'] = self.request.user
        return super(UserViewSet, self).perform_create(serializer, **kwargs)
```

And set *REST_EASY_GENERIC_VIEW_MIXINS* in your Django settings to:

```
REST_EASY_GENERIC_VIEW_MIXINS = ['myapp.mixins.InjectUserMixin']
```

Now all serializers will receive user as a parameter when calling *save()* from a update or create view.

4.3 Scopes

Scopes are used to apply additional filters to views' querysets based on data obtainable form kwargs (*rest_easy.scopes.UrlKwargScopeQuerySet*) and request (*rest_easy.scopes.RequestAttrScopeQuerySet*). They should be used remove the boilerplate and bloat coming from filtering inside get_queryset or in dedicated mixins by providing a configurable wrapper for the filtering logic.

There is also a base *rest_easy.scopes.ScopeQuerySet* that you can inherit from to provide your own logic. When called, the ScopeQuerySet instance receives whole view object as a parameter, so it has access to everything that happens during the request as well as in application as a whole.

Scopes can be chained (that is you can filter scope's queryset using another scope, just as it was a view; this supports lists of scopes as well). An example would be:

```
class MessageViewSet(ModelViewSet):
    model = Message
    schema = 'default'
    lookup_url_kwarg = 'pk'
    scope = UrlKwargScopeQuerySet(Thread, parent=UrlKwargScopeQuerySet(Account))
```

4.3.1 ScopeQuerySet

When instantiating it, it accepts the following parameters (*{value}* is the filtering value obtained by concrete Scope implementation):

- qs_or_obj: a queryset or model (in that case, the queryset would be *model.objects.all()*) that the scope works on. This can also be *None* in special cases (for example, when using `rest_easy.scopes.RequestAttrScopeQuerySet` with *is_object=True*). For example, assuming you have a model Message that has foreign key to Thread, when scoping a *MessageViewSet* you would use *scope = ScopeQuerySet(Thread)*.
- parent_field: the field qs_or_obj should be filtered by. By default it is pk. Following the example, the scope above would find the Thread object by *Thread.objects.all().filter(pk={value})*.
- raise_404: If the instance we're scoping by isn't found (in the example, Thread with pk={value}), whether a 404 exception should be raised or should we continue as usual. By default False
- allow_none: If the instance we're scoping by isn't found and 404 is not raised, whether to allow filtering child queryset with None (*allow_none=True*) or not - in this case we will filter with model.objects.none() and guarantee no results (*allow_none=False*). False by default.
- get_object_handle: the name under which the object used for filtering (either None or result of applying {value} filter to queryset) will be available on the view. By default this is inferred to model_name. Can be set to None to disable access. It can be accessed from view as view.get_{get_object_handle}, so when using the above example, view.get_thread(). If the get_x method would be shadowed by something else, there is an option to call view.get_scoped_object(get_object_handle), so for example view.get_scoped_object(thread).
- parent: parent scope. If present, qs_or_obj will be filtered by the scope or scopes passed as this parameter, just as if this was a view.

4.3.2 UrlKwargScopeQuerySet

It obtains filtering value from *view.kwargs*. It takes one additional keyword argument:

- url_kwarg: what is the name of kwarg (as given in url config) which has the value to filter by. By default it is configured to be model_name_pk (model name is obtained from qs_or_obj).

Example:

```
scope = UrlKwargScopeQuerySet(Message.objects.active(), parent_field='uuid', url_
↪kwarg='message_uuid', raise_404=True)
queryset = scope.child_queryset(queryset, view)
# is equal to roughly:
queryset = queryset.filter(message=Message.objects.active().get(uuid=view.kwargs.get(
↪'message_uuid'))
```

4.3.3 RequestAttrScopeQuerySet

It obtains the filtering value from *view.request*. It takes two additional keyword arguments:

- request_attr: the attribute in *view.request* that contains the filtering value or the object itself.
- is_object: whether the request attribute contains object (True) or filtering value (False). By default True.

Example with *is_object=True*:

```
scope = RequestAttrScopeQuerySet(User, request_attr='user')
queryset = scope.child_queryset(queryset, view)
# is roughly equal to:
queryset = queryset.filter(user=view.request.user)
```

Example with *is_object=False*:

```
scope = RequestAttrScopeQuerySet(User, request_attr='user', is_object=False)
queryset = scope.child_queryset(queryset, view)
# is roughly equal to:
queryset = queryset.filter(user=User.objects.get(pk=view.request.user))
```

4.4 Helpers

There are following helpers available in *rest_easy.models*:

- *rest_easy.models.SerializableMixin* - it's supposed to be used on models. It provides *rest_easy.models.SerializableMixin.get_serializer()* method for obtaining model serializer given a schema and *rest_easy.models.SerializableMixin.serialize()* to serialize data (given schema or None, in which case the default schema is used. It can be set on a model, initially it's just *'default'*).
- *rest_easy.models.get_serializer()* - looking at a blob of data, it obtains the serializer from register based on *data['model']* and *data['schema']*.
- *rest_easy.models.deserialize_data()* - deserializes a blob of data if appropriate serializer is found.

CHAPTER 5

API docs

Django-rest-easy provides base classes for API views and serializers.

To leverage the QOL features of django-rest-easy, you should use the followint base classes for your serializers:

- *rest_easy.serializers.Serializer*
- *rest_easy.serializers.ModelSerializer*

And if it's model-based, it should use one of the base views provided in the *rest_easy.views* module - preferably *rest_easy.views.ModelViewSet* or *rest_easy.views.ReadOnlyModelViewSet*.

As a baseline, all responses using django-rest-easy extension will contain top-level model and schema fields.

Guidelines regarding schemas are as usual: they have to be 100% backwards compatible. In the case of breaking changes, a serializer with new schema should be created, and the old one slowly faded away - and removed only when no applications use it - or when it's decided that the feature can't be supported anymore.

An alternative to multi-version fadeout is single-version fadeout, where the change is implemented as a set of acceptable changes (that is, you can remove the old field only when all clients stop using it - even if it means sending duplicate data for quite some time).

The classes from this module don't disable any behaviour inherent to Django Rest Framework - anything that is possible there will be possible with the django-rest-easy base classes.

Django Rest Easy uses following settings:

- REST_EASY_AUTOIMPORT_SERIALIZERS_FROM - for autoimporting serializers.
- REST_EASY_VIEW_BASES - for prepending bases to all views declared in django-rest-easy. They will end up before all base views, either DRF's or django-rest-easy's, but after generic mixins in the final generic view mro. So in *rest_easy.views.GenericAPIView* and `rest_easy.views.GenericAPIViewSet` they will be at the very beginning of the mro, but everything declared in generic mixins, like DRF's CreateMixin, will override that.
- REST_EASY_GENERIC_VIEW_MIXINS - for prepending bases to generic views. They will end up at the beginning of mro of all generic views available in django-rest-easy. This can be used to make views add parameters when doing perform_update() or perform_create().

- REST_EASY_SERIALIZER_CONFLICT_POLICY - what happens when serializer with same model and schema is redefined. Defaults to 'allow', can also be 'raise' - in the former case the new serializer will replace the old one. Allow is used to make sure that any import craziness is not creating issues by default.

class rest_easy.**ApiConfig**(*app_name*, *app_module*)
Bases: django.apps.config.AppConfig

AppConfig autoimporting serializers.

It scans all installed applications for modules specified in settings.REST_EASY_AUTOIMPORT_SERIALIZERS_FROM parameter, trying to import them so that all residing serializers using *rest_easy.serializers.SerializerCreator* metaclass will be added to *rest_easy.registers.SerializerRegister*.

In the case of a module not being present in app's context, the import is skipped. In the case of a module existing but failing to import, an exception will be raised.

autodiscover()
Auto-discover serializers in installed apps, fail silently when not present, re-raise exception when present and import fails. Borrowed form django.contrib.admin with added nested presence check.

default_paths = ['serializers', 'api.serializers']

label = 'rest_easy'

name = 'rest_easy'

paths
Get import paths - from settings or defaults.

ready()
Override this method in subclasses to run code when Django starts.

5.1 Exceptions

This module contains exceptions native to django-rest-easy.

exception rest_easy.exceptions.**RestEasyException**
Bases: exceptions.Exception

Default django-rest-easy exception class.

5.2 Fields

This module contains fields necessary for the django-rest-easy module.

class rest_easy.fields.**StaticField**(*value*, ***kwargs*)
Bases: rest_framework.fields.Field

A field that always provides the same value as output.

The output value is set on initialization, ie:

```
from rest_easy.serializers import Serializer

class MySerializer(Serializer):
    static = StaticField('This will always be the value.')
```

to_representation(*value*)
Return the static value.

5.3 Models

This module provides useful model mixins and global functions.

Its contents can be used to serialize a model or find proper serializer/deserialize data via a registered serializer.

class rest_easy.models.**SerializableMixin**
Bases: object

This mixin provides serializing functionality to Django models.

The serializing is achieved thanks to serializers registered in *rest_easy.registers.SerializerRegister*. A proper serializer based on model and provided schema is obtained from the register and the serialization process is delegated to it.

Usage:

`python from users.models import User serializer = User.get_serializer(User.default_schema) ` Or:

`python data = User.objects.all()[0].serialize() `

default_schema = u'default'

classmethod get_serializer(*schema*)
Get correct serializer for this model and given schema,

Utilizes *rest_easy.registers.SerializerRegister* to obtain correct serializer class. :param schema: schema to be used for serialization. :return: serializer class.

serialize(*schema=None*)
Serialize the model using given or default schema. :param schema: schema to be used for serialization or self.default_schema :return: serialized data (a dict).

rest_easy.models.**get_serializer**(*data*)
Get correct serializer for dict-like data.

This introspects model and schema fields of the data and passes them to *rest_easy.registers.SerializerRegister*. :param data: dict-like object. :return: serializer class.

rest_easy.models.**deserialize_data**(*data*)
Deserialize dict-like data.

This function will obtain correct serializer from *rest_easy.registers.SerializerRegister* using *rest_easy.models.get_serializer()*. :param data: dict-like object or json string. :return: Deserialized, validated data.

5.4 Patterns

This class defines generic bases for a few design / architectural patterns required by django-rest-easy, namely singleton and register.

class rest_easy.patterns.**SingletonCreator**
Bases: type

This metaclass wraps __init__ method of created class with singleton_decorator. This ensures that it's impossible to mess up the instance for example by calling __init__ with getattr.

static singleton_decorator(*func*)
We embed given function into checking if the first (zeroth) parameter of its call shall be initialised. :param func: instantiating function (usually __init__). :returns: embedded function function.

class rest_easy.patterns.**SingletonBase**
Bases: object

This class implements the singleton pattern using a metaclass and overriding default __new__ magic method's behaviour. It works together with SingletonCreator metaclass to create a Singleton base class. sl_init property is reserved, you can't use it in inheriting classes.

class rest_easy.patterns.**Singleton**
Bases: *rest_easy.patterns.SingletonBase*

This is a Singleton you can inherit from. It reserves sl_init instance attribute to work properly.

class rest_easy.patterns.**BaseRegister**(***kwargs*)
Bases: *rest_easy.patterns.Singleton*

This class is a base register-type class. You should inherit from it to create particular registers.

conflict_policy is a setting deciding what to do in case of name collision (registering another entity with the same name). It should be one of:

- allow - replace old entry with new entry, return True,
- deny - leave old entry, return False,
- raise - raise RestEasyException.

Default policy is raise.

As this is a singleton, instantiating a particular children class in any place will yield the exact same data as the register instance used in RegisteredCreator().

conflict_policy = 'allow'

entries()
Return an iterator over all registered entries.

classmethod get_conflict_policy()
Obtain conflict policy from django settings or use default.

Allowed settings are 'raise' and 'allow'. Default is 'raise'.

lookup(*name*)
I like to know if an entry is in the register, don't you? :param name: name to check. :returns: True if entry with given name is in the register, False otherwise.

register(*name*, *ref*)
Register an entry, shall we? :param name: entry name. :param ref: entry value (probably class). :returns: True if model was added just now, False if it was already in the register.

class rest_easy.patterns.**RegisteredCreator**
Bases: type

This metaclass integrates classes with a BaseRegister subclass.

It skips processing base/abstract classes, which have __abstract__ property evaluating to True.

static get_fields_from_base(*base*)
Obtains all fields from the base class. :param base: base class. :return: generator of (name, value) tuples.

classmethod get_missing_fields(*required_fields*, *fields*)
Lists required fields that are missing.

Supports two formats of input of required fields: either a simple set {'a', 'b'} or a dict with several options:

```
{
    'nested': {
        'presence_check_only': None,
        'functional_check': lambda value: isinstance(value, Model)
    },
    'flat_presence_check': None,
    'flat_functional_check': lambda value: isinstance(value, Model)
}
```

Functional checks need to return true for field not to be marked as missing. Dict-format also supports both dict and attribute based accesses for fields (fields['a'] and fields.a).

Parameters

- **required_fields** – set or dict of required fields.
- **fields** – dict or object of actual fields.

Returns List of missing fields.

static get_name(*name*, *bases*, *attrs*)
Get name to be used for class registration.

inherit_fields = False

classmethod post_register(*cls*, *name*, *bases*, *attrs*)
Post-register hook. :param cls: created class. :param name: class name. :param bases: class bases. :param attrs: class attributes. :return: None.

classmethod pre_register(*name*, *bases*, *attrs*)
Pre-register hook. :param name: class name. :param bases: class bases. :param attrs: class attributes. :return: Modified tuple (name, bases, attrs)

classmethod process_required_field(*missing*, *fields*, *name*, *value*)
Processes a single required field to check if it applies to constraints.

register = None

required_fields = set([])

5.5 Registers

This module contains the serializer register.

The serializer register is where all serializers created using *rest_easy.serializers.SerializerCreator* are registered and where they can be obtained from based on model and schema. Remember that no other serializers will be kept here - and they will not be obtainable in such a way.

class rest_easy.registers.**SerializerRegister**(***kwargs*)
Bases: *rest_easy.patterns.BaseRegister*

Obtains serializer registration name based on model and schema.

get(*model*, *schema*)
Shortcut to get serializer having model and schema.

static get_name(*model*, *schema*)
Constructs serializer registration name using model's app label, model name and schema. :param model: a Django model, a ct-like app-model string (app_label.modelname) or explicit None. :param schema: schema to be used. :return: constructed serializer registration name.

5.6 Scopes

This module provides scopes usable with django-rest-easy's generic views.

See *rest_easy.views* for detailed explanation.

class rest_easy.scopes.**ScopeQuerySet**(*qs_or_obj*, *parent_field=u'pk'*, *related_field=None*, *raise_404=False*, *allow_none=False*, *get_object_handle=u''*, *parent=None*)

Bases: object

This class provides a scope-by-parent-element functionality to views and their querysets.

It works by selecting a proper parent model instance and filtering view's queryset with it automatically.

child_queryset(*queryset*, *view*)

Performs filtering of the view queryset. :param queryset: view queryset instance. :param view: view object. :return: filtered queryset.

contribute_to_class(*view*)

Put self.get_object_handle into view's available handles dict to allow easy access to the scope's get_object() method in case the object needs to be reused (ie. in child object creation). :param view: View the scope is added to.

get_object(*view*)

Caching wrapper around _get_object. :param view: DRF view instance. :return: object (instance of init's qs_or_obj model except shadowed by subclass).

get_queryset(*view*)

Obtains parent queryset (init's qs_or_obj) along with any chaining (init's parent) required. :param view: DRF view instance. :return: queryset instance.

get_value(*view*)

Get value used to filter qs_or_objs's field specified for filtering (parent_field in init). :param view: DRF view instance - as it provides access to both request and kwargs. :return: value to filter by.

class rest_easy.scopes.**UrlKwargScopeQuerySet**(**args*, ***kwargs*)

Bases: *rest_easy.scopes.ScopeQuerySet*

ScopeQuerySet that obtains parent object from url kwargs.

get_value(*view*)

Obtains value from url kwargs. :param view: DRF view instance. :return: Value determining parent object.

class rest_easy.scopes.**RequestAttrScopeQuerySet**(**args*, ***kwargs*)

Bases: *rest_easy.scopes.ScopeQuerySet*

ScopeQuerySet that obtains parent object from view's request property.

It can work two-fold:

- the request's property contains full object: in this case no filtering of parent's queryset is required. When using such approach, is_object must be set to True, and qs_or_obj can be None. Chaining will be disabled since it is inherent to filtering process.
- the request's property contains object's id, uuid, or other unique property. In that case is_object needs to be explicitly set to False, and qs_or_obj needs to be a Django model or queryset. Chaining will be performed as usually.

get_value(*view*)

Obtains value from url kwargs. :param view: DRF view instance. :return: Value determining parent object.

5.7 Serializers

This module contains base serializers to be used with django-rest-easy.

Crucial point of creating a good API is format consistency. If you've been lacking that so far, can't afford it anymore or want to make your life easier, you can enforce a common message format and a common serialization format. Enter the following SerializerCreator - it will make sure that everything serializers output will contain schema and model fields. This affects both regular and model serializers.

Additional benefit of using such metaclass is serializer registration - we can easily obtain serializers based on model (or None for non-model serializers) and schema from anywhere in the application. That's useful in several cases:

- model serialization
- remote data deserialization (no changes to (de)serialization logic required when we add a new schema)
- simpler views and viewsets

This doesn't disable any DRF's serializers functionality.

class rest_easy.serializers.**ModelSerializer**(*instance=None*, *data=<class rest_framework.fields.empty>*, ***kwargs*)

Bases: rest_framework.serializers.ModelSerializer

Registered version of DRF's ModelSerializer.

class rest_easy.serializers.**Serializer**(*instance=None*, *data=<class rest_framework.fields.empty>*, ***kwargs*)

Bases: rest_framework.serializers.Serializer

Registered version of DRF's Serializer.

class rest_easy.serializers.**RegisterableSerializerMixin**

Bases: object

A mixin to be used if you want to inherit functionality from non-standard DRF serializer.

class rest_easy.serializers.**SerializerCreator**

Bases: *rest_easy.patterns.RegisteredCreator*, rest_framework.serializers.SerializerMetaclass

This metaclass creates serializer classes to be used with django-rest-easy.

We need to employ multiple inheritance here (if the behaviour ever needs to be overridden, you can just use both base classes to implement your own functionality) to preserve DRF's behaviour regarding serializer fields as well as registration and required fields checking from our own metaclass.

Remember that all __new__ methods from base classes get called.

static get_fields_from_base(*base*)

Alteration of original fields inheritance.

It skips all serializer fields, since SerializerMetaclass deals with that already. :param base: a base class. :return: generator of (name, value) tuples of fields from base.

static get_name(*name*, *bases*, *attrs*)

Alteration of original get_name.

This, instead of returing class's name, obtains correct serializer registration name from *rest_easy.registers.SerializerRegister* and uses it as slug for registration purposes. :param name: class name. :param bases: class bases. :param attrs: class attributes. :return: registered serializer name.

inherit_fields = False

classmethod pre_register(*name*, *bases*, *attrs*)
Pre-register hook adding required fields

This is the place to add required fields if they haven't been declared explicitly. We're adding model and schema fields here. :param name: class name. :param bases: class bases. :param attrs: class attributes. :return: tuple of altered name, bases, attrs.

register = <rest_easy.registers.SerializerRegister object>

required_fields = {u'Meta': {u'model': <function <lambda>>, u'schema': <function <la

5.8 Views

This module provides redefined DRF's generic views and viewsets leveraging serializer registration.

One of the main issues with creating traditional DRF APIs is a lot of bloat (and we're writing Python, not Java or C#, to avoid bloat) that's completely unnecessary in a structured Django project. Therefore, this module aims to provide a better and simpler way to write simple API endpoints - without limiting the ability to create more complex views. The particular means to that end are:

- *rest_easy.scopes.ScopeQuerySet* and its subclasses (*rest_easy.scopes.UrlKwargScopeQuerySet* and *rest_easy.scopes.RequestAttrScopeQuerySet*) provide a simple way to scope views and viewsets. by resource (ie. limiting results to single account, or /resource/<resource_pk>/inner_resource/<inner_resource_pk>/)
- generic views leveraging the above, as well as model-and-schema specification instead of queryset, serializer and helper methods - all generic views that were available in DRF as well as GenericAPIView are redefined to support this.
- Generic *rest_easy.views.ModelViewSet* which allows for very simple definition of resource endpoint.

To make the new views work, all that's required is a serializer:

```
from users.models import User
from accounts.models import Account
from rest_easy.serializers import ModelSerializer
class UserSerializer(ModelSerializer):
    class Meta:
        model = User
        fields = '__all__'
        schema = 'default'

class UserViewSet(ModelViewSet):
    model = User
    scope = UrlKwargScopeQuerySet(Account)
```

and in urls.py:

```
from django.conf.urls import url, include
from rest_framework.routers import DefaultRouter
router = DefaultRouter()
router.register(r'accounts/(?P<account_pk>[0-9]+)/users', UserViewSet)
urlpatterns = [url(r'^', include(router.urls))]
```

The above will provide the users scoped by account primary key as resources: with list, retrieve, create, update and partial update methods, as well as standard HEAD and OPTIONS autogenerated responses.

You can easily add custom paths to viewsets when needed - it's described in DRF documentation.

class rest_easy.views.**GenericAPIView**(***kwargs*)
Bases: rest_easy.views.GenericAPIViewBase

Base view with compat metaclass.

rest_easy_available_object_handles = {}

class rest_easy.views.**CreateAPIView**(***kwargs*)
Bases: rest_easy.views.ChainingCreateUpdateMixin, rest_framework.mixins.CreateModelMixin, *rest_easy.views.GenericAPIView*

Concrete view for retrieving or deleting a model instance.

post(*request*, **args*, ***kwargs*)
Shortcut method.

rest_easy_available_object_handles = {}

class rest_easy.views.**ListAPIView**(***kwargs*)
Bases: rest_framework.mixins.ListModelMixin, *rest_easy.views.GenericAPIView*

Concrete view for listing a queryset.

get(*request*, **args*, ***kwargs*)
Shortcut method.

rest_easy_available_object_handles = {}

class rest_easy.views.**RetrieveAPIView**(***kwargs*)
Bases: rest_framework.mixins.RetrieveModelMixin, *rest_easy.views.GenericAPIView*

Concrete view for retrieving a model instance.

get(*request*, **args*, ***kwargs*)
Shortcut method.

rest_easy_available_object_handles = {}

class rest_easy.views.**DestroyAPIView**(***kwargs*)
Bases: rest_framework.mixins.DestroyModelMixin, *rest_easy.views.GenericAPIView*

Concrete view for deleting a model instance.

delete(*request*, **args*, ***kwargs*)
Shortcut method.

rest_easy_available_object_handles = {}

class rest_easy.views.**UpdateAPIView**(***kwargs*)
Bases: rest_easy.views.ChainingCreateUpdateMixin, rest_framework.mixins.UpdateModelMixin, *rest_easy.views.GenericAPIView*

Concrete view for updating a model instance.

patch(*request*, **args*, ***kwargs*)
Shortcut method.

put(*request*, **args*, ***kwargs*)
Shortcut method.

rest_easy_available_object_handles = {}

class rest_easy.views.**ListCreateAPIView**(***kwargs*)
Bases: rest_easy.views.ChainingCreateUpdateMixin, rest_framework.mixins.ListModelMixin, rest_framework.mixins.CreateModelMixin, *rest_easy.views.GenericAPIView*

Concrete view for listing a queryset or creating a model instance.

get(*request*, **args*, ***kwargs*)
Shortcut method.

post(*request*, **args*, ***kwargs*)
Shortcut method.

rest_easy_available_object_handles = {}

class rest_easy.views.**RetrieveUpdateAPIView**(***kwargs*)
Bases: rest_easy.views.ChainingCreateUpdateMixin, rest_framework.mixins.RetrieveModelMixin, rest_framework.mixins.UpdateModelMixin, *rest_easy.views.GenericAPIView*

Concrete view for retrieving, updating a model instance.

get(*request*, **args*, ***kwargs*)
Shortcut method.

patch(*request*, **args*, ***kwargs*)
Shortcut method.

put(*request*, **args*, ***kwargs*)
Shortcut method.

rest_easy_available_object_handles = {}

class rest_easy.views.**RetrieveDestroyAPIView**(***kwargs*)
Bases: rest_framework.mixins.RetrieveModelMixin, rest_framework.mixins.DestroyModelMixin, *rest_easy.views.GenericAPIView*

Concrete view for retrieving or deleting a model instance.

delete(*request*, **args*, ***kwargs*)
Shortcut method.

get(*request*, **args*, ***kwargs*)
Shortcut method.

rest_easy_available_object_handles = {}

class rest_easy.views.**RetrieveUpdateDestroyAPIView**(***kwargs*)
Bases: rest_easy.views.ChainingCreateUpdateMixin, rest_framework.mixins.RetrieveModelMixin, rest_framework.mixins.UpdateModelMixin, rest_framework.mixins.DestroyModelMixin, *rest_easy.views.GenericAPIView*

Concrete view for retrieving, updating or deleting a model instance.

delete(*request*, **args*, ***kwargs*)
Shortcut method.

get(*request*, **args*, ***kwargs*)
Shortcut method.

patch(*request*, **args*, ***kwargs*)
Shortcut method.

put(*request*, **args*, ***kwargs*)
Shortcut method.

rest_easy_available_object_handles = {}

class rest_easy.views.**ReadOnlyModelViewSet**(***kwargs*)
Bases: rest_framework.mixins.RetrieveModelMixin, rest_framework.mixins.ListModelMixin, rest_easy.views.GenericViewSet

A viewset that provides default *list()* and *retrieve()* actions.

rest_easy_available_object_handles = {}

class rest_easy.views.**ModelViewSet**(***kwargs*)
Bases: rest_easy.views.ChainingCreateUpdateMixin, rest_framework.mixins.CreateModelMixin, rest_framework.mixins.RetrieveModelMixin, rest_framework.mixins.UpdateModelMixin, rest_framework.mixins.DestroyModelMixin, rest_framework.mixins.ListModelMixin, rest_easy.views.GenericViewSet

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

rest_easy_available_object_handles = {}

Python Module Index

r

Index

I

L

M

N

P

R

S

T

U