django-rest-models Documentation

Release 1.8.5

Darius BERNARD

Apr 17, 2019

Contents

Django Rest Models is a ROI implementation that allow django to query a remote REST API with the legacy Model ORM. It works by creating a second database configuration, which will query the API instead of a legacy Relational database.

Contents:

CHAPTER 1

Quick start

1.1 Client

1.1.1 Settings

- add the router `'rest_models.router.RestModelRouter'` to the setting `DATABASE_ROUTERS`
- add your api to the `DATABASES` setting

```
DATABASES = {
    'default': {
        ...
    },
    'api': {
        'ENGINE': 'rest_models.backend',
        'NAME': 'http://127.0.0.1:8001/api/v2/',
        'USER': 'interim',
        'PASSWORD': 'interim',
        'AUTH': 'rest_models.backend.auth.OAuthToken',
        'OPTIONS': {
            'OAUTH_URL': '/oauth2/token/',
            'TIMEOUT': 10,
        }
    },
}

DATABASE_ROUTERS = [
    'rest_models.router.RestModelRouter',
]
```

1.1.2 Models

Create your models that will match your serializers on the api. The main customisation to make it a non database model is the addition of the class `APIMeta`. Thanks to the RestModelRouter, this addition will choose the database with the rest_models backend as the backend to use. see *APIMeta* for more details on all customisation.

```
class MyModel(models.Model):
    field = models.IntegerField()
    othermodel = models.ForeignKey(db_column="othermodel")
    ...

    class Meta:
        # basic django meta Stuff
        verbose_name = 'my model'

    # the only customisation that makes this model special
    class APIMeta:
        pass
```

Note: if you use any of these fields:

- ImageField
- FileField
- JSONField
- ForeignKey

you must take extra modifications on your models, see *build your client models*.

1.1.3 Usage

Use it as any normal Django Model. Just keep in mind that the backend is not a SGDB and it may not be performant on all queryset, and that some query is not possible.

You can not:

- aggregate
- annotate
- make complex filters with NOT and OR

1.2 API side

On the API side, you don't need to install this lib. But the serializers must follow these constraint :

- inherit the `DynamicModelSerializer` from `dynamic-rest`
- provide all related serializers using `DynamicRelationField` from `dynamic-rest`
- provide all backward relation in both serializers.

1.2.1 Examples

with the folowing models

```
class Menu(models.Model):
    name = models.CharField(max_length=135)
    code = models.CharField(max_length=3)

    def __str__(self):
        return self.name  # pragma: no cover


class Topping(models.Model):
    name = models.CharField(max_length=125)
    cost = models.FloatField()

    def __str__(self):
        return self.name  # pragma: no cover


class Pizza(models.Model):

    name = models.CharField(max_length=125)
    price = models.FloatField()
    from_date = models.DateField(auto_now_add=True)
    to_date = models.DateTimeField(default=auto_now_plus_5d)

    creator = models.ForeignKey(settings.AUTH_USER_MODEL, null=True)
    toppings = models.ManyToManyField(Topping, related_name='pizzas')
    menu = models.ForeignKey(Menu, null=True, related_name='pizzas')

    def __str__(self):
        return self.name  # pragma: no cover
```

1.2.2 Inheritance

- inherit the `DynamicModelSerializer` from `dynamic-rest`

Bad:

```
from rest_framework import serializers

class MenuSerializer(serializers.Serializer):
    ...
```

Good:

```
from dynamic_rest.serializers import DynamicModelSerializer

class MenuSerializer(DynamicModelSerializer):
    ...
```

1.2.3 Related serializers fields

- Provide all related serializers using `DynamicRelationField` from `dynamic-rest`

Bad:

```
class PizzaSerializer(DynamicModelSerializer):
    toppings = ToppingSerializer(many=True)
```

Good:

```
from dynamic_rest.fields.fields import DynamicRelationField

class PizzaSerializer(DynamicModelSerializer):
    toppings = DynamicRelationField(ToppingSerializer, many=True)
```

1.2.4 Backward relationship

- Provide all backward relation in both serializers.

bad:

```
class MenuSerializer(DynamicModelSerializer):
    # missing backward serializer to pizza, which have a «menu» foreignkey

    class Meta:
        model = Menu
        name = 'menu'
        fields = ('id', 'code', 'name')


class PizzaSerializer(DynamicModelSerializer):

    menu = DynamicRelationField(MenuSerializer)

    class Meta:
        model = Pizza
        name = 'pizza'
        fields = ('id', 'name', 'price', 'from_date', 'to_date', 'menu')
```

Good:

```
class MenuSerializer(DynamicModelSerializer):
    pizzas = DynamicRelationField('PizzaSerializer', many=True)  # good backward link.
↪ respecting menu.related_name

    class Meta:
        model = Menu
        name = 'menu'
        fields = ('id', 'code', 'name', 'pizzas')


class PizzaSerializer(DynamicModelSerializer):

    menu = DynamicRelationField(MenuSerializer)

    class Meta:
        model = Pizza
        name = 'pizza'
        fields = ('id', 'name', 'price', 'from_date', 'to_date', 'menu')
```

CHAPTER 2

build your client models

to build your models based on the one on the api, it pretty simple.

your api models look like this:

```
from django.db import models
from django.contrib.postgres.fields import JSONField


class Menu(models.Model):
    name = models.CharField(max_length=135)
    code = models.CharField(max_length=3)


    def __str__(self):
        return self.name  # pragma: no cover

class Pizza(models.Model):

    name = models.CharField(max_length=125)
    price = models.FloatField()
    from_date = models.DateField(auto_now_add=True)
    secret_ingredient = models.CharField(max_length=125)
    metadata = JSONField(null=True)


    menu = models.ForeignKey(Menu, null=True, related_name='pizzas')

    photo = models.ImageField(upload_to='/images/')

    def __str__(self):
        return self.name  # pragma: no cover
```

0. copy/past the model class from the api to your own models.py
1. add the APIMeta to your each model class. see See *APIMeta* section to know how you can customize it.

```
from django.db import models
from django.contrib.postgres.fields import JSONField


class Menu(models.Model):
    name = models.CharField(max_length=135)
    code = models.CharField(max_length=3)

    def __str__(self):
        return self.name  # pragma: no cover


    class APIMeta:
        pass

class Pizza(models.Model):

    name = models.CharField(max_length=125)
    price = models.FloatField()
    from_date = models.DateField(auto_now_add=True)
    secret_ingredient = models.CharField(max_length=125)
    metadata = JSONField(null=True)

    menu = models.ForeignKey(Menu, null=True, related_name='pizzas')

    photo = models.ImageField(upload_to='/images/')

    def __str__(self):
        return self.name  # pragma: no cover

    class APIMeta:
        pass
```

2. comment/remove the fields which is not exposed in the serializer of the models in your api

from:

```
class Pizza(models.Model):

    ...
    secret_ingredient = models.CharField(max_length=125)
    ...
```

to:

```
class Pizza(models.Model):

    ...
    # this sensitive data is not exposed by the api, we can't use it on the client.
    # secret_ingredient = models.CharField(max_length=125)
    ...
```

3. change all ForeignKey by adding a db_column equivalent to the field name itself.

from:

```
class Menu(models.Model):
    ...
```

(continues on next page)

(continued from previous page)

```
class Pizza(models.Model):


    menu = models.ForeignKey(Menu, null=True, related_name='pizzas')
    ...
```

to:

```
class Menu(models.Model):
    ...


class Pizza(models.Model):


    menu = models.ForeignKey(Menu, null=True, related_name='pizzas', db_column='menu')
    ...
```

4. change all ImageField/FileField to add our custom storage which handle upload to api

```
from rest_models.storage import RestApiStorage
...

class Pizza(models.Model):
    ...

    photo = models.ImageField(storage=RestApiStorage())
    ...
```

5. change all JSONField from django.contrib.postgres.fields to use our custom field.

   our custom field just handle the get_prep_value to make it compatible with our backend.

```
from rest_models.backend.utils import JSONField

...

class Pizza(models.Model):

    ...
    metadata = JSONField(null=True)
    ...
```

6. enjoy your new model, which should look like this:

```
from django.db import models
from rest_models.storage import RestApiStorage
from rest_models.backend.utils import JSONField


class Menu(models.Model):
    name = models.CharField(max_length=135)
    code = models.CharField(max_length=3)
```

(continues on next page)

(continued from previous page)

```
    def __str__(self):
        return self.name  # pragma: no cover


    class APIMeta:
        db_name = 'api'

class Pizza(models.Model):

    name = models.CharField(max_length=125)
    price = models.FloatField()
    from_date = models.DateField(auto_now_add=True)
    # this sensitive data is not exposed by the api, we can't use it on the client.
    # secret_ingredient = models.CharField(max_length=125)
    metadata = JSONField(null=True)

    menu = models.ForeignKey(Menu, null=True, related_name='pizzas', db_column='menu')

    photo = models.ImageField(storage=RestApiStorage())

    def __str__(self):
        return self.name  # pragma: no cover


    class APIMeta:
        db_name = 'api'
```

CHAPTER 3

Settings

Some custom settings are available to customise the behavior of rest_models.

3.1 DATABASES setting

Example of many settings:

```
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.sqlite3',
        'NAME': 'db.sq3',
    },
    'api': {
        'ENGINE': 'rest_models.backend',
        'NAME': 'http://localapi/api/v2',
        'USER': 'admin',
        'PASSWORD': 'admin',
        'AUTH': 'rest_models.backend.auth.BasicAuth',
    },
    'api2': {
        'ENGINE': 'rest_models.backend',
        'NAME': 'http://127.0.0.1:8001/api/v2/',
        'USER': 'pwd',
        'PASSWORD': 'pwd',
        'AUTH': 'rest_models.backend.auth.OAuthToken',
        'OPTIONS': {
            'OAUTH_URL': '/oauth2/token/',
            'TIMEOUT': 10,
            'SKIP_CHECK': True,
        },
        'PREVENT_DISTINCT': False,
    },
    'TEST_api2': {
```

(continues on next page)

(continued from previous page)

```
        'ENGINE': 'rest_models.backend',
        'NAME': 'http://localhost:8080/api/v2',
        'USER': 'userapi',
        'PASSWORD': 'passwordapi',
        'AUTH': 'rest_models.backend.auth.BasicAuth',
    },
}
```

The database setting for the engine rest_models.backends accept the folowing values :

3.1.1 NAME

This settings give the url to the api root. it must be a http/https url that can have a path.

The path must be the one that give you a 200 status code with the list of serializers.

Examples :

- `http://127.0.0.1:8001/api/v2/`
- `https://api.mysite.org/`
- `http://customhost/api/`

Special urls

The special host *localapi* can be provided to bypass the network and redirect the connection on the local running process in the same way as django do during tests (via Client). This allow to test against an api database without having to make it run before the tests, and allow the transactions to work for unit-testing. See *Testing*

Example:

- `http://localapi/api/v2`
- `http://localapi/`

This will make requests to the local API system, rather than via the network to a remote system.

3.1.2 USER

The settings `USER` may be used by the `AUTH` backend. This depends on the backend used. Can be empty if the backend doesn't need it or if there is no backend (public api)

3.1.3 PASSWORD

Like `USER` it can be used by the `AUTH` backend

3.1.4 AUTH

The path to the `ApiAuthBase` subclass that will be used to provide the authentication header to the api. Each query will __call__ this subclass with the `requests.Request` object and shall update the header to provide the data for authentications.

The following backends are shiped with rest_models:

rest_models.backend.auth.BasicAuth

Provide the Basic authentication with `USER` and `PASSWORD`

rest_models.backend.auth.OAuthToken

Provide the OAuth2 authentication. This will fetch a token using `USER` and `PASSWORD` each time it's expired, and provide the API with the header `Authorization: Bearer <token>`

This backend can use extra settings in `OPTIONS` named `OAUTH_URL` which is the endpoint to the Oauth2 token provider. By default this url is `/oauth2/token/`.

3.1.5 OPTIONS['TIMEOUT']

Provide the time for triggering a new query on the api. If a query take longer than this, it will retry 3 more times, and eventialy raise an OperationalError.

3.1.6 OPTIONS['SKIP_CHECK']

Will skip checking the api if this settings is set to true. By default, the Django check command (executed during tests and migration) will query the api to check if our models match the structure of the api. Settings this to True will prevent any query to be made to the api. This is useful for testing environments where all queries are faked and there is no api at all.

3.1.7 PREVENT_DISTINCT

This settings allow to accept request with a *distinct* clause without raising an Exception. Note that the distinct stuff will be trashed and the final query may repeat his lines. Enable it if you know what you are doing.

3.2 APIMeta

On each api models, a nested class named APIMeta must be attached to the model. This class can contain some customisation for the model.

Example:

```
class Menu(models.Model):
    name = models.CharField(max_length=135)
    code = models.CharField(max_length=3)

    class APIMeta:
        db_name = 'api'
        resource_path = 'menulol/'
        resource_name = 'menu'
        resource_name_plural = 'menus'
```

3.2.1 db_name

Provide the name of the database connection in which this model is placed. If there is only one database connection that use rest_models backend, it is optional. If there is more than one connection with this backend, all models MUST give this setting on APIMeta

3.2.2 resource_path

The value to append to the path of the api to get the endpoint of this model. In many cases, it's the «verbose_name» on the api side. or the value given in the router:

```
router = DynamicRouter()
router.register('pizza', PizzaViewSet)  # this match the verbose_name of Pizza...␣
↪default behavior will work
router.register('topping', ToppingViewSet)
router.register('menulol', MenuViewSet)  # «menulol» for path. must be specified␣
↪since menulol don't match verbose_name
```

3.2.3 resource_name

The value for the serializer.Meta.name

```
class PizzaSerializer(DynamicModelSerializer):

    class Meta:
        model = Pizza
        name = 'pizza' # resource name match the verbose_name of the model. no need␣
↪to customise resource_name
```

3.2.4 resource_name_plural

This is the plural variant of resource_name. If the resource_name is customized, you will need to customize this too. In many cases, it will resource_name + 's'

```
class PizzaSerializer(DynamicModelSerializer):

    class Meta:
        model = Pizza
        name = 'pizza' # resource name match the verbose_name of the model. no need␣
↪to customise resource_name_plural
```

CHAPTER 4

Testing

4.1 localapi

To write tests using the api database, Rest Model allow you to bypass the usage of a remote API to loopback into the local process for the api query. If the host of the api is `localapi`, Rest Model will not go to the network seeking a host named `localapi` but will query the local url engine to fetch the serializers.

If you have your API application installed as local dependecies for testing, and an url running with it, you can write tests painlessly.

This allows:

- database transactions on all requests made to the api
- faster testing since no network stack is used
- easier testing setup, since no api should be started before the tests

By default, the test database for api will use the `localapi` system. to bypass this, provide a `DATABASES` config named `TEST_{name}`. This database will be used for all queries on the api database *{name}*

```
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.sqlite3',
        'NAME': 'db.sq3',
    },
    'api': {
        'ENGINE': 'rest_models.backend',
        'NAME': 'http://localapi/api/v2',
        'USER': 'admin',
        'PASSWORD': 'admin',
        'AUTH': 'rest_models.backend.auth.BasicAuth',
    },

    'TEST_api': {  # replace the database «api» durring the tests
        'ENGINE': 'rest_models.backend',
```

(continues on next page)

(continued from previous page)

```
        'NAME': 'http://localhost:8080/api/v2',
        'USER': 'userapi',
        'PASSWORD': 'passwordapi',
        'AUTH': 'rest_models.backend.auth.BasicAuth',
        'OPTIONS': {
            'SKIP_CHECK': True,
        },
    },
}
```

4.2 Mock API

4.2.1 Overview

You can mock the api with a custom response for each given url. It won't trigger any api query, but will instead return the predefined data from each request matching the patterns.

Your test cases must inherit from either `rest_models.test.RestModelTestMixin` or `rest_models.test.RestModelTestCase`

With this, you have 2 more functionalities.

You can provide the matching «url» => «response» by giving the `rest_fixtures` like this:

```
class TestAnonymousVisit(RestModelTestMixin, TestCase):

    rest_fixtures = {
        '/oauth2/token/': [
            {'data': {'scope': 'read write', 'access_token':
↪'HJKMe81faowKipJGKZSwg05LnfJmrU',
                                    'token_type': 'Bearer', 'expires_in': 36000}}
        ],
        '/pizzas/': 'path/to/fixtures.json',
    }
```

with the file in `path/to/fixtures.json` :

```
[
  {
    "filter": {},
    "data": {
      "pizza": {
        "cost": 2.08,
        "to_date": "2016-11-20T08:46:02.016000",
        "from_date": "2016-11-15",
        "price": 10.0,
        "id": 1,
        "links": {
          "toppings": "toppings/"
        },
        "name": "suprème",
        "toppings": [
          1,
          2,
```

(continues on next page)

(continued from previous page)

```
            3,
            4,
            5
        ],
        "menu": 1
      }
    }
  }
]
```

4.2.2 Providing data

Global to the tests

If you have 2 or more api databases, you must provide a mapping *database* => *fixtures* in the static attribute `database_rest_fixtures`. If you have only one api database, the `database_rest_fixtures` is automatically mapped to the default one:

```
class TestAnonymousVisit(RestModelTestMixin, TestCase):

    database_rest_fixtures = {'api': {  # api is our first database
        '/oauth2/token/': [
            {'data': {'scope': 'read write', 'access_token':
↪'HJKMe81faowKipJGKZSwg05LnfJmrU',
                                    'token_type': 'Bearer', 'expires_in': 36000}}
        ],
    }}
```

Local to a function

You can temporary mock the data from the api by using `RestModelTestMixin.mock_api` context manager:

```
class TestAnonymousVisit(RestModelTestMixin, TestCase):

    def test_remote_name_mismatch(self):

        with self.mock_api('pizza', {'pizzas': []}, using='api'):
            self.assertEqual(len(list(Pizza.objects.all())), 0)
```

It takes 3 arguments :

- url: the url to mock
- result : the result to return for the given url
- params: the parameters that will be used to filter the usage of this mock
- using: optionally the api to mock, if there is more than one

4.2.3 Data structure

The structure of the mocked data is a list of possible results, represented by a dict with two keys :

- data: the actual data returned by the api if it was queried (`{"pizzas": [...], "menus": [...]}`)

- filter: for the given data to be used, the query must match this dict of data
- statuscode: the status code to simulate

Data

The data is a copy of the real result expected in the api.

The following is extracted from the rest api interface and is a valid `data` value

```
{
    "pizzas": [
        {
            "links": {
                "toppings": "toppings/"
            },
            "to_date": "2016-11-20T08:46:02.016000",
            "price": 10.0,
            "cost": 2.08,
            "name": "suprème",
            "from_date": "2016-11-15",
            "toppings": [
                1,
                2,
                3,
                4,
                5
            ],
            "menu": 1,
            "id": 1
        },

    ],
    "meta": {
        "per_page": 10,
        "total_pages": 1,
        "page": 1,
        "total_results": 1
    }
}
```

Filter

The filter is a dict or a list of dict that can be empty, in that case it will match all queries. It can contain one of the following relevant values - any other will make this dataset not match any query. If it's a list, any dict inside that matches the query will validate this fixture.

- params: the main filter helper. it must contains a dict with the query parameters in the get for the api
- method: the method used (get, post, put, . . .)
- json: the posted data

Params

The params filter is a dict with each item the part of the final query GET to the api.

For example:

`?filter{name}=lolilol&filter{pizza.name}=pipi =>`

```
{'params': {'filter{name}': 'lolilol', 'filter{pizza.name}': 'pipi'}}
```

JSON

The json must match the POSTed/PUT data if given. If you created a Menu with name='hey' :

```
'filter': {
    'method': 'post',
    'json': {'menu': {'name': 'hey'}}  # posted data must match this
},
```

Note: remember that all posted data must return a 201 status code

```
{  # response for post
    'filter': {
        'method': 'post',
        ...
    },
    'data': {  # this will return a fake models created response
        ...
    },
    'status_code': 201  # the mandatory statuscode to return for a post success
},
```

4.2.4 Full example

The following test case is a full example taken from the test suit. It's a good point for start.

```
class TestMockDataSample(RestModelTestCase):
    database_rest_fixtures = {'api': {  # api => response mocker for database named
↪«api»
        'menulol': [  # url menulol
            {
                'filter': {  # set of filters to match
                    'params': {  # params => requests parameters to sort[],exclude[],
↪filter{...},include[]
                        'filter{name}': ['lolilol'],  # with filter(name='lolilol')
                        'sort[]': ['-name']  # with order_by('-name')
                    }
                },
                'data': {
                    "menus": [],
                    "meta": {
                        "per_page": 10,
                        "total_pages": 1,
                        "page": 1,
                        "total_results": 0
                    }
                }
```

(continues on next page)

(continued from previous page)

```
            },
            {
                'filter': [{
                    'params': {
                        'filter{name}': ['lolilol'],  # just the filter, no sorting
                    }
                }],
                'data': {
                    "menus": [
                        {
                            "links": {
                                "pizzas": "pizzas/"
                            },
                            "id": 1,
                            "pizzas": [
                                1
                            ],
                            "name": "main menu",
                            "code": "mn"
                        }
                    ],
                    "meta": {
                        "per_page": 10,
                        "total_pages": 1,
                        "page": 1,
                        "total_results": 1
                    }
                }
            },
            {  # response for post
                'filter': {
                    'method': 'post',
                    'json': {'menu': {'name': 'hey'}}  # posted data must match this
                },
                'data': {  # this will return the fake models created response
                    "menu": {
                        "id": 1,
                        "pizzas": [],
                        "name": "hey",
                        "code": "hy"
                    }
                },
                'status_code': 201  # the mandatory status code to return for a post␣
↪success
            },
            {  # response for post
                'filter': {
                    'method': 'post',
                },
                'data': {
                    "menu": {
                        "id": 2,
                        "pizzas": [],
                        "name": "hello",
                        "code": "ho"
                    }
                },
```

(continues on next page)

(continued from previous page)

```
                'status_code': 201
            },
            {  # fallback
                'filter': {},  # no filter => fallback
                'data': {
                    "menus": [
                        {
                            "links": {
                                "pizzas": "pizzas/"
                            },
                            "id": 1,
                            "pizzas": [
                                1
                            ],
                            "name": "lolilol",
                            "code": "mn"
                        },
                        {
                            "links": {
                                "pizzas": "pizzas/"
                            },
                            "id": 2,
                            "pizzas": [
                                2
                            ],
                            "name": "lolilol",
                            "code": "ll"
                        }
                    ],
                    "meta": {
                        "per_page": 10,
                        "total_pages": 1,
                        "page": 1,
                        "total_results": 2
                    }
                }

            }
        ]
    }}

    def test_multi_results_filter(self):
        # no filter/no sort => fallback
        self.assertEqual(len(list(Menu.objects.all())), 2)
        # no matching filter => fallback
        self.assertEqual(len(list(Menu.objects.filter(code='pr'))), 2)
        # no matching filter => fallback
        self.assertEqual(len(list(Menu.objects.filter(name='pr'))), 2)
        # matching filter/no sort => don't care for missing sort and return 1rst
        self.assertEqual(len(list(Menu.objects.filter(name='lolilol'))), 1)
        # no matching sort => 2nd found
        self.assertEqual(len(list(Menu.objects.filter(name='lolilol').order_by('name
↪'))), 1)
        # no matching sort => 1st found
        self.assertEqual(len(list(Menu.objects.filter(name='lolilol').order_by('-name
↪'))), 0)
        # no matching filter => fallback
```

(continues on next page)

(continued from previous page)

```
        self.assertEqual(len(list(Menu.objects.filter(name='pr').order_by('-name'))),␣
↪2)

    def test_post_filter(self):
        # no filter/no sort => fallback
        m = Menu.objects.create(name='hey', code='!!')
        self.assertEqual(m.pk, 1)
        self.assertEqual(m.name, 'hey')
        self.assertEqual(m.code, 'hy')

        m = Menu.objects.create(name='prout', code='??')
        self.assertEqual(m.pk, 2)
        self.assertEqual(m.name, 'hello')
        self.assertEqual(m.code, 'ho')
```

CHAPTER 5

Middlewares

the Rest Models Middlewares work like the django ones. it intercept all query that will be sent to the api, and can update the params, or bypass the api and return a result.

class rest_models.backend.middlewares.**ApiMiddleware**
a base class to implemente a middleware that will interact with a api query/response

data_response(*data*, *status_code=None*)
shortcut to return a response with 200 and data

Parameters **data** – the data to insert in the response

Returns a FakeApiResponse with the given data

empty_response()
shortcut to return a response with 204 status code and no data, which will be

Returns a FakeApiResponse with no data and 204 status code

static make_response(*data*, *status_code*)
helper to make a response (returned by process_response or process_request) with given data

Parameters

- **data** – the data in the response (will be encoded in json to be compatible)
- **status_code** – the status code of the response

Returns a FakeApiResponse that contains raw data

process_request(*params*, *requestid*, *connection*)
process the request. if return other than None, it will be the result of the request

Parameters

- **params** (*dict*) –

  the params given (and modified by previous middleware) for the _make_query. the params can be updated at will by side-effect.

  params contain

  - verb: the verb to execut the request (post, get, put)
  - url: the url in which the query will be executed
  - data: the data given to the query to post,put, etc
- **requestid** (*int*) – the id of the current request done by this connection
- **connection** – the connection used for this query

Returns the response if there is no need to pursue the query.

process_response(*params*, *response*, *requestid*)
process the response from a previous query. MUST return a response (result)

Parameters

- **params** – the params finaly given to query the api. same format as for process_request
- **response** – the response, either the original one or modifier by preceding middleware
- **requestid** (*int*) – the id of the current request done by this connection

Returns

5.1 helper Middleware

CHAPTER 6

Special cases

6.1 prefetch_related

The *prefetch_related* method is a Django optimisation for querysets that causes the database to cache the related fields for all given manytomany relation in the model. This feature works in django-rest-models backend, but the remote api will return far more data than the SQL equivalent should.

To prevent performance issues in the api, django-rest-models will add a special get parameter «filter_to_prefetch» in the query which can be interpreted by the backend to filter all sub-query with the actual id.

A small trick in the api side is to override the DynamicFilterBackend with the following class, and use it in your views.

```
class PrefetchRelatedCompatibleFilterBackend(DynamicFilterBackend):
    def _build_requested_prefetches(
            self,
            prefetches,
            requirements,
            model,
            fields,
            filters
    ):
        # hack that apply the original filters on the responses from a

        prefetch = super(PrefetchRelatedCompatibleFilterBackend, self)._build_
↪requested_prefetches(
            prefetches,
            requirements,
            model,
            fields,
            filters
        )
        if self.request.GET.get('filter_to_prefetch'):
            for field in prefetch:
                in_ = filters.get('_include', {}).get('%s__in' % field)
                if in_:
```

(continues on next page)

(continued from previous page)

```
                    prefetch[field].queryset = prefetch[field].queryset.filter(pk__
↪in=in_.value)
        return prefetch
```

To use this new FilterBackend, you can write your view like this

```
class CompanyViewSet(DynamicModelViewSet):
    serializer_class = CompanySerializer
    queryset = Company.objects.all()
    filter_backends = (PrefetchRelatedCompatibleFilterBackend, DynamicSortingFilter)
```

the *filter_backends* attribute is inherited from the WithDynamicViewSetMixin mixin. replace the *DynamicFilterBackend* with your override

CHAPTER 7

Indices and tables

- genindex
- modindex
- search

Index