django easy rest Documentation

jonatan shimon

Sep 14, 2018

Contents

Django easy rest is a rest framework with the focus of making development faster, it is built on top of the awesome django rest framework (drf).

the main purpose of this framework is to give a speed boost to developers in rest api fields.

It features a rich and useful debugger, scripts, classes, mixins and brings django context to your javascripts

and does much more for you fore more read the docs:

CHAPTER 1

List of contents

1.1 setup

1.1.1 Installation

To install the python api package use pip

```
pip install vcore_api
```

1.1.2 django setup

add easy rest to your installed apps in settings.py:

```
INSTALLED_APPS = [
    ...
    'easy_rest',
]
```

for advanced feature the easy rest root should be included in your app urls as follows:

in your urls.py

```
url(r'^easy_rest/', include('easy_rest.urls')),
```

if you change the root url (for instance the url is myproject/api/something) you need to override it in settings.py

```
EASY_REST_ROOT_URL = "myproject/api/something"
```

1.1.3 Base html file

```
{% load easy_rest %}
<head>
    {% load_rest_scripts %}
</head>
```

1.2 The main rest view

The main view of the api is the RestApiView, with mixins combined this view is very powerful, there are 3 main optional parameters to initialize:

- function_field_name, default "action", this is the api function in the class see example below
- api_allowed_post_methods, default ["__all__"], this field is for security reasons and defines the avilable methods for post in the class
- api_allowed_get_methods, default ["__all__"], this field is for security reasons and defines the avilable methods for get in the class

1.2.1 Example of the api

python:

```
from easy_rest.views import RestApiView

class UsersApi(RestApiView):
    api_allowed_post_methods = [
        "authenticate",
    ]

    def authenticate(self, api_data):
        """
        Some authentication code here
        """

        return {
            "data": "user authenticated successfully"
        }
```

Sync request demo

```
let api = new RequestHandler("/<url_to_api>/");

// now lets create a sync post to authenticate a user

let api_data = api.PostSync({"action": "authenticate"});

console.log(api_data.data);
```

Async request demo

```
let api = new RequestHandler("/<url_to_api>/");

```

(continues on next page)

(continued from previous page)

```
// now lets create a sync post to authenticate a user

api.PostASync({"action": "authenticate"}, function(api_data) {
    console.log(api_data.data);
});
```

1.2.2 The function field name

this field controls the name of the function field

```
let api = new RequestHandler("/<url_to_api>/");

// if function_field_name = "action", the request is
api.PostSync({"action": "authenticate"});

// if function_field_name = "something_else", the request is
api.PostSync({"something_else": "authenticate"});
```

1.3 Function arguments unpacker

This mixin unpack the request arguments into the function

```
from easy_rest.views import RestApiView

class UsersApi(RestApiView):
    api_allowed_post_methods = [
        "larger_then",
    ]

    def larger_then(self, first_number, second_number):
        return {
            "output": int(first_number) > int(second_number)
        }
```

```
let api = new RequestHandler("/api/math");

api_data = api.PostSync({"action": "larger_then", "first_number": 1, "second_number":␣
↪2});

console.log(api_data.data.output);
```

The framework will unpack first_number and second_number into the argument field, the order here doesn't mater it matches the argument strings.

1.4 Javascript context

This mixin allows you to access the view context using javascript

```
from easy_rest.mixins import TemplateContextFetcherMixin
from django.views import generic

class ActiveTemplate(JavascriptContextMixin, generic.TemplateView):
    template_name = 'demo_app/home.html'


    def get_context_data(self, **kwargs):
        ctx = super(WelcomePage, self).get_context_data(**kwargs)
        ctx['message'] = "This is javascript context mixin"
        return ctx
```

```
let consts = window.restConsts;

// now to access the context, do the following:
console.log(consts.context);
console.log(consts.context.message);
```

1.4.1 Requirements

Requirements for the javascript context

```
<html>
    {% load easy_rest %}
    <head>
        {% load_rest_scripts %}
    </head>
</html>
```

1.5 Form post mixin

This mixin is for django generic class based views, it post the django forms using javascript, (no refresh is needed).

```
from easy_rest.mixins import FormPostMixin
from django.views.generic import UpdateView

class UpdateViewApi(FormPostMixin, UpdateView):
    fields = ['first_name', 'last_name']
    template_name = 'app/test_form_post.html'
    model = User
    success_message = 'model has been changed {}'.format(datetime.now())

    def get_object(self, queryset=None):
        return User.objects.get(pk=1)
```

```
<html lang="en">
    {% load easy_rest %}

    <head>
        {% load_rest_all %}
    </head>
    <body>
```

(continues on next page)

(continued from previous page)

```
        {% include "easy_rest/easy_rest_form.html" with form=form %}
    </body>
</html>
```

You can also add an override save function for the form

```
from easy_rest.mixins import FormPostMixin
from django.views.generic import UpdateView

class UpdateViewApi(FormPostMixin, UpdateView):
    fields = ['first_name', 'last_name']
    template_name = 'app/test_form_post.html'
    model = User
    success_message = 'model has been changed {}'.format(datetime.now())

    def get_object(self, queryset=None):
        return User.objects.get(pk=1)

@staticmethod
def form_save_function(form):
    form.save()
    print("form saved")
```

1.5.1 Requirements

load all the rest scripts and styles using

```
<html>
    {% load easy_rest %}
    <head>
        {% load_rest_all %}
    </head>
</html>
```

1.6 Live context

this mixin add a functionality to declare live context section, this section will render automatically live from the server

```
from easy_rest.mixins import TemplateContextFetcherMixin
from django.views import generic

class ActiveTemplate(TemplateContextFetcherMixin, generic.TemplateView):
    template_name = 'app/live_ctx_demo.html'

    def get_context_data(self, **kwargs):
        return {"time": str(datetime.now()), "random_int": randint(0, 100)}
```

```
<html>
    {% load easy_rest %}
    <head>
        {% load_rest_scripts %}
    </head>
```

(continues on next page)

(continued from previous page)

```
    <body>
        {% livecontext %}
           <h1>Live time from server {time}</h1>

           <h1>Random int from server {random_int}</h1>

        {% endlivecontext %}
    </body>
</html>
```

If you want your context from another view (live context tag takes a url)

```
<html>
    {% load easy_rest %}
    <head>
        {% load_rest_scripts %}
    </head>
    <body>
        {% livecontext "/url/to/other/template_view" %}
           <h1>Live time from server {time}</h1>

           <h1>Random int from server {random_int}</h1>

        {% endlivecontext %}
    </body>
</html>
```

1.7 Model unpacker mixin

Unpacks the request model into the handling function

```
from easy_rest.views import RestApiView

class UsersApi(RestApiView):
    api_allowed_get_methods = [
        "get_username",
    ]

    def user_name(self, user):
        return {
            "user_name": user.username
        }
```

```
let api = new RequestHandler("/api/math");

api_data = api.PostSync({"action":"get_username", "with-model": {"field":"auth.User",
↪"query":{"pk":1}}});

console.log(api_data.data.user_name);
```

1.7.1 Security

Keep security in mind when using this feature.

1.8 Decorative keys mixin

This mixin is really simple, it make your function_field_name value everything decorative, if the function name is authenticate_user then the request will look like:

```
let api = new RequestHandler("/<url_to_api>/");

api.PostSync({"action": "authenticate_user"});
```

With decorative keys the request can be:

```
let api = new RequestHandler("/<url_to_api>/");

api.PostSync({"action": "authenticate_user"});
api.PostSync({"action": "authenticate user"});
api.PostSync({"action": "authenticate-user"});
api.PostSync({"action": "authenticate:user"});
```

1.9 Automated testing

In rest api the functionality we often need to test is the request and response, the framework contains an automated test mixin

1.9.1 Generate a test

just add the PostRecordTestGenerator to your view and in the init function init the test

```
from easy_rest.test_framework.recorder.post_record_mixins import␣
↪PostRecordTestGenerator
class ApiTest(PostRecordTestGenerator, RestApiView):

    def __init__(self, *args, **kwargs):
        super(ApiTest, self).__init__(*args, **kwargs)
        self.init_test(app_name='demo_app')

    def echo(self, data):
        return {"echo": data}
```

then run some requests for example:

```
let api = new RequestHandler("/some_url");

api.PostSync({});
api.PostSync({"action":"echo", "data":"hello"});
```

The framework will generate tests for you

1.9.2 Generated test example

```
from django.test import TestCase
from demo_app.views import ApiTest
from django.test import RequestFactory
from django.contrib.auth.models import AnonymousUser, User
from easy_rest.test_framework.resolvers.resolve import register_unittest
from django.test.utils import override_settings

register_unittest()


def resolve_user(pk):
    try:
        return User.objects.get(pk=pk)
    except Exception:
        return AnonymousUser()


class TestApiTest(TestCase):
    @override_settings(DEBUG=True)
    def test_echo(self):
        request = RequestFactory()
        request.data = {'action': 'echo', 'data': 'asdf'}
        request.user = resolve_user(None)
        result = {'debug': {'api-attributes': {'api-allowed-methods': ['__all__']},
                            'processed-data': {'action': 'echo', 'data': 'asdf'}},
                  'debug-mode': ['enabled', 'to disable go to settings.py and change␣
↪DEBUG=True to false'],
                  'data': {'echo': 'asdf'}}
        if type(result) is dict:
            return self.assertDictEqual(result, self.test.post(request).data)
        return self.assertEqual(result, self.test.post(request).data)

    def __init__(self, *args, **kwargs):
        super(TestApiTest, self).__init__(*args, **kwargs)
        self.test = ApiTest()

    @override_settings(DEBUG=True)
    def test_easy_rest_2017_08_26_12_38_31_143966_test(self):
        request = RequestFactory()
        request.data = {}
        request.user = resolve_user(None)
        result = {'error': 'no action in data',
                  'debug': {'api-attributes': {'api-allowed-methods': ['__all__']},
↪'processed-data': {}},
                  'debug-mode': ['enabled', 'to disable go to settings.py and change␣
↪DEBUG=True to false']}
        if type(result) is dict:
            return self.assertDictEqual(result, self.test.post(request).data)
        return self.assertEqual(result, self.test.post(request).data)
```

1.10 Debugger

The debugger will automatically catch exception on debug mode, and will create a debug url.

then it will redirect the client to the url and produce an error message.

it will also build the javascript code that caused this exception

1.10.1 Screenshot

click the image to expand