django-yaat Documentation

Release 1.4.0

Slapec

April 13, 2016

Contents

Django-yaat helps you creating django-restify-framework API endpoints for the yaat AngularJS module by providing a simple resource called `YaatModelResource`.

This class handles column hiding, reordering and data sorting. For most cases where a single Django model is used this class works out of box however there is some place in it for advanced usage.

Contents:

CHAPTER 1

Installing

1. **The module is not on PyPI yet** but once it is published you can install it with `pip` like everything else:

```
pip install django-yaat
```

2. Then add the `'yaat'` module to the `INSTALLED_APPS`.
3. Migrate the models:

```
python manage.py migrate yaat
```

CHAPTER 2

Getting started

2.1 Defining the resource

First of all, because the class is called Yaat*Model*Resource you should have some model to show. So lets define one.

```
from django.db import models


class Item(models.Model):
    name = models.CharField(max_length=64)
    quantity = models.PositiveIntegerField()
    price = models.PositiveIntegerField()
```

Also import `YaatModelResource` and subclass it. The *Item* class (and some other attributes) is listed in a class called `Meta` **inside** the resource class. You're might already be familiar with this method because Django uses it too.

```
from yaat.resource import YaatModelResource

class ModelExampleResource(YaatModelResource):
    class Meta:
        resource_name = 'model-example'
        model = Item
        columns = ('name', 'quantity', 'price')
```

`YaatModelResource` is very similary to `restify.resource.ModelResource` except that it defines some additional meta attributes. In the above example `resource_name` and `model` are inherited, but `columns` is yaat-only.

Column list is always required (like Django forms require to specify either `fields` or `exclude`). Here you can list any names that are fields of the `Item` model (listed in `Item._meta.fields`) or you can add `Column` objects too.

That's it, the resource is ready. Now you have to register it as a restify-framework API endpoint.

Note: It is a good idea to create a Python module named `api` inside the Django application which has restify-framework resources. Put resources in `api/resources.py` and custom serializers in `api/serializers.py`.

2.2 Registering the API endpoint

You're not required but it is a good practice to put all API endpoints into a separate file somewhere near `ROOT_URLCONF`.

```
from restify.api import Api

api = Api(api_name='example')
api.register(regex='model_example/$', resource=ModelExampleResource)
```

So here you've registered the resource as any other restify endpoint. Then include the URLs of the API in an urlconf.

```
from django.conf.urls import include, url

urlpatterns = [
    url(r'^api/', include(api.api.urls, namespace='api'))
]
```

And there it is, the endpoint is ready to receive `POST`s.

2.3 Connecting the directive to the endpoint

This is as easy as the other steps above. In a Django template you can get the URL of every resource under the `api` namespace. This is why the `resource_name` property is required.

```
<yat api="{% url 'api:model-example' %}"></yat>
```

The `<yat>` directive handles everything else for you. If you want to customize that too, head to the yaat repository.

CHAPTER 3

Limiting the resource

By default the resource accepts any positive integer as a limit. If you have hundreds of rows you should adjust the limit value and limit choices in your resource to avoid overloading your backend.

To add a single value as a limit add the `limit` property to the meta class:

```
class Limited(YaatModelResource):
    class Meta:
        resource_name = 'limited'
        model = Item
        limit = 3
        columns = ('name', 'quantity', 'price')
```

Here the resource replies with 3 rows every time it is queried. The value of `limit` POSTed by yaat is completely ignored.

3.1 Limit choices

There is also space in the resource if you're planning to create a table where the user can change the row limit. Simply add the `limit_choices` to the meta class. It should be a tuple or a list of single values (not like in Django where you must provide value pairs).

```
class LimitedChoices(YaatModelResource):
    class Meta:
        resource_name = 'limited'
        model = Item
        limit = 3
        limit_choices = [3, 6, 9]
        columns = ('name', 'quantity', 'price')
```

Note: Yaat detects changing of the `$limit` model but it doesn't have any feature to change it on the UI. So there is no example of this resource, but I promise it works :D!

CHAPTER 4

Working examples

You can find working examples in the bundled Django example project in django-yaat's repository.

CHAPTER 5

Advanced resources

5.1 Custom columns

In the meta class of the resource you can list either strings or `Column` objects under the `columns` property. String keys must always match a field of the model. `Column` keys however are free to have any values. `Column` objects have the property `key` which is going to be used for model property or method lookup. This is very useful to send values of related models or computed values in the `POST`.

Let's say we have the following models:

```
class Owner(models.Model):
    name = models.CharField(max_length=64)


class SmartItem(models.Model):
    owner = models.ForeignKey(Owner)
    name = models.CharField(max_length=64)
    quantity = models.PositiveIntegerField()
    price = models.PositiveIntegerField()

    @property
    def get_total_price(self):
        return self.quantity * self.price

    @property
    def get_owner(self):
        return self.owner.name
```

Here we've created a simple relation between `SmartItem` and `Owner` so I can show you how to send related values too but it is super easy.

So there are 2 property methods `get_total_price` and `get_owner`. To create a resource for this define the following resource:

```
from yaat.models import Column

class ModelComputedExampleResource(YaatModelResource):
    class Meta:
        resource_name = 'model-computed-example'
        model = SmartItem
        columns = (
            Column(key='get_owner', value='Owner'),
            'name', 'quantity', 'price',
```

```
            Column(key='get_total_price', value='Total price')
        )
```

First you have to import the `Column` model. `Column` objects are going to be mapped to properties or methods of the model of the resource by their key (`key` argument). So in this example when the resource iterates over the queryset it will get `get_owner` **property** of the model first, then `name`, `quantity`, `prce` **fields** of the model, then `get_total_price` **property** of the model at the end. Those properties and methods that are invoked by the `Columns` are called *handlers*.

We call these columns *virtual* meaning that you can't order by their values out of box (because the ORM can't handle it). You are allowed to create *non-virtual* columns too but then you must implement the sorting method of those objects.

However the `Column` class is a Django model but it's never stored anywhere unless you mark the resource to be *stateful*.

5.2 Passing values to handlers

Sometimes it is useful to pass an object to a column handler e.g. when you want the model to access the `User` instance (`request.user`)

Let's modify the method `SmartItem.get_total_price` from the previous example.

```
def get_total_price(self, currency):
    return self.quantity * self.price * currency
```

It's quite trivial but let's say that you want to calculate the `Item`'s total price based on the logged in user's currency settings. Note that the method is no longer decorated: you can't pass values to properties.

To pass the value to the handler you have to override the `get_rows` method of the resource class `ModelComputedExampleResource`:

```
def get_rows(self, *args):
    return super().get_rows(*args, currency=request.user.currency)
```

Here you simply invoke the method from the parent class, pass all arguments and your own value as a keyword argument. Every handler method receives every passed keyword argument meaning that you have to decide in the handler itself which arguments you need. Use the `**kwargs` argument in this cases.

Note: In the internal implementation when `get_rows` gets the model attribute it checks if it is a callable. If it's true then it is invoked with all keyword arguments of `get_rows`. Otherwise no other processing is made and the value is stored in the row.

5.3 Modifying the row dict

In every page each row is described with the following dict structure:

```
{'id': obj.pk, 'values': cells}
```

Here `obj.pk` is the primary key of the object and `cells` is a `list` containing the cells of the given row. This is what yaat expects from you.

If you want to modify this dict (to define a row property) you can do that by overriding the `row_hook()` in your resource like this:

```
from yaat.resource import YaatModelResource

class ModelExampleResource(YaatModelResource):
    class Meta:
        resource_name = 'model-example'
        model = Item
        columns = ('name', 'quantity', 'price')

    def row_hook(self, row, obj):
        row = super().row_hook(row)
        row['is_active'] = obj.is_active
        return row
```

If you need more complex modification you have to override `get_rows()`.

Warning: Do not modify or remove the `'id'` and `'values'` keys or the the rendered table will not work at all.

5.4 Modifying the table

Modifying the whole table is not possible in the resource itself but in a custom serializer class.

By default `YaatModelResource` classes use the `restify.serializers.DjangoSerializer` serializer.

You should subclass that, override the `flatten()` method, and there you can access the `dict` describing the required table page. Please don't remove or modify existing keys and values because that may make the rendering fail on the client.

However it's a good place to add properties to the whole table e. g. server time. You can access non-yaat data described in yaat docs.

5.5 Stateful columns

It is possible to store column states in a persistent storage so you get back the same table when you reload the page. Only column-related things are stored (order, ordering and if it's hidden). Current page and limit are not.

To make a resource columns stateful simply add the `stateful` to its meta class:

```
class StatefulColumns(YaatModelResource):
    class Meta:
        stateful = True
```

That's it. Any change is going to be saved in your database.

5.5.1 Customizing the column foreign key

Yaat's Column model has a foreign key to `settings.AUTH_USER_MODEL` by default. This is what you need in 99.9% of cases. However sometimes you may want the columns to be accessible from a different model (like from a related model of the User class).

To adjust this set the `settings.YAAT_FOREIGN_KEY` key to a string. It is expected to be a dotted pair of the Django app and the Django model just like for `AUTH_USER_MODEL`. See the docs.

After changing the foreign key you also have to set the `settings.YAAT_REQUEST_ATTR` setting because subclasses of `YaatModelResource` depend on `request.user` which is likely not an instance of the new foreign key class anymore. This value is expected to be a single string. The attribute with the same name must exist in the `request` object.

Real world example

Let's say we have 2 models, `Customer` and `User` in an N:M relation through an other model, `Membership`, similar to the Django example example.

Here the same user should have different column lists depending on which of its membership is active. This means that `columns` should be a property of `Membership` instances. To achieve this set the setting:

```
YAAT_FOREIGN_KEY = 'myapp.Membership'
```

(Assume `Membership` model is in the `myapp` Django application)

Since `Column.user` is expected to point to a `Membership` instance, and `request.user` is still a `User`, you have to add the active `Membership` object to each request. It's the easiest using a middleware. Say the `Membership` is accessible through `request.member` then set the setting to this:

```
YAAT_REQUEST_ATTR = 'member'
```

Note: Keep in mind that the name of the property `Column.user` stays the same if you override `YAAT_FOREIGN_KEY` but it points to a different type of object then.

Warning: Changing `YAAT_FOREIGN_KEY` has a huge impact just like changing `AUTH_USER_MODEL`. Be sure to set this value before applying your migrations the very first time. If you set this value later the real foreign key in your database will still point to the old table.

5.6 Stateful table pages

Django-yaat can store yaat's last `limit` and `offset` values in the authenticated user's session so you can send the exact same page every time the user arrives to the same table. This is useful for cases when the user navigates away and back to the same table often.

Simply add the `stateful_init` to the meta class of the resource:

```
class StatefulInit(YaatModelResource):
    class Meta:
        stateful_init = True
```

You can combine this with `stateful` of course.

5.7 Utility methods

There are a few utility methods that may help you in some rare cases.

class `YaatModelResource`

classmethod **invalidate_column_cache**(*user*)

This method forces to invalidate the given user's `Column`. This can help you if you add a new `Column` object on the fly and you want to show it immediately.

Argument `user` is expected to be an instance of `AUTH_USER_MODEL` class or `YAAT_FOREIGN_KEY` class if that's specified.

class **YaatValidatorForm**

invalidate_state()

Every `YaatModelResource` (and its subclasses) gets the attribute `self.validator_form` when the `YaatModelResource.common(request, *args, **kwargs)` method is invoked. This form is used for validating the received data during paging but also for creating the initial data. If you set the resource meta to `stateful_init = True` the form keeps its last received data as the initialization state. If you'd like to drop this state call this method.

CHAPTER 6

Indices and tables

- genindex
- modindex
- search

I

Y