audoma - API Automatic Documentation Maker

Release 0.1

Iteo

Sep 20, 2022

CONTENTS:

Audoma is a Python library that was designed to make Django Rest Framework documentation easier and more automatic. It's build on the top of *drf-spectacular* and extends some of its functionalities.

CHAPTER

ONE

INSTALLATION

1.1 Python version

Audoma supports and was tested for the following versions of python:

- python 3.7
- python 3.8
- python 3.9

1.2 Dependencies

These distributions will be installed automatically when installing audoma (if you don't have them installed already):

- Django
- django-rest-framework
- drf-spectacular
- exrex - a regular expression generator
- django-phonenumber-field - a phone number field package for django
- django-macaddress - a mac address field package for django
- phonenumbers - python package for parsing phone numbers - we use it for generating examples of phone numbers
- django-filter - django package that allows to filter down a queryset based on a model's fields
- django-money - django package that allows to store money and currency values in the database
- lorem - generator for random text that looks like Latin.

1.3 Install Audoma

Using *pip*:

```
$ pip install audoma
```

1.4 Audoma initial configuration

After successful installation, set *AudomaAutoSchema* as a default schema class for your Django Rest Framework project:

```
REST_FRAMEWORK = {
    # YOUR SETTINGS
    'DEFAULT_SCHEMA_CLASS': 'audoma.drf.openapi.AudomaAutoSchema',
}
```

Audoma allows you to add path prefixes that should be included in schema exclusively. All you need to do is declare a variable *SCHEMA_PATTERN_PREFIX*` in your project's *settings.py* file and add preprocessing function *preprocess_include_path_format* as preprocessing hook in *SPECTACULAR_SETTINGS dictionary*` as in the example below.

```
SCHEMA_PATTERN_PREFIX = 'api'

SPECTACULAR_SETTINGS = {
    ...
    'PREPROCESSING_HOOKS': ['audoma.hooks.preprocess_include_path_format'],
    # OTHER SETTINGS
}
```

CHAPTER

TWO

USAGE

2.1 Viewset defined serializers

2.1.1 Serializer class custom configs

By, default Django Rest Framework provides a method to get a serializer - *get_serializer*.
This checks if viewset instance has set *serializer_class* property and returuns its instance.
Audoma Extends this behavior by, extending the number of possible serializer class declarations.

First of all, you are allowed to define *collect* and *response* serializer classes for viewset.
Collect serializer will be used to collect and process request data.
Response serializers will be used to process data for the response.

Variable name pattern: *common_{type}_serializer_class* (type can be result or collect)
Example:

```
from audoma.drf import viewsets
from audoma.drf import mixins
from example_app.serializers import (
    MyCollectSerializer,
    MyResultSerializer
)

class MyViewSet(
    mixins.ActionModelMixin,
    mixins.ListModelMixin,
    viewsets.GenericViewSet
):
    common_collect_serializer_class = MyCollectSerializer
    common_result_serializer_class = MyResultSerializer
```

Additionally audoma allows a definition of a custom serializer for each action in the viewset.
This is possible for generic drfs' actions and also for custom actions,
created with *@action* decorator.

Variable name pattern: *{action_name}_serializer_class*
Example:

```
from rest_framework.decorators import action
from rest_framework.response import Response

from audoma.drf import viewsets
from audoma.drf import mixins
from example_app.serializers import (
    MyCreateSerializer,
    MyCustomActionSerializer
)

class MyViewSet(
    mixins.ActionModelMixin,
    mixins.CreateModelMixin,
    viewsets.GenericViewSet
)::
    create_serializer_class = MyCreateSerializer
    custom_action_serializer_class = MyCustomActionSerializer

    @action(detail=True, methods=["post"])
    def custom_action(self, request):
        serializer = self.get_serializer(data=request.data)
        serializer.is_valid(raise_exception=True)
        serializer.save()
        return Response(serializer.instance, status_code=200)
```

It is also possible for action to serve more than one HTTP method.
In audoma, it is allowed to assign different serializers for each of the actions HTTP methods.

Variable name pattern: *{http_method}_{action_name}_serializer_class*
Example:

```
from audoma.drf import viewsets
from audoma.drf import mixins
from example_app.serializers import (
    MyCreateSerializer,
    MyCustomActionSerializer
)

class MyViewSet(
    mixins.ActionModelMixin,
    mixins.ListModelMixin,
    viewsets.GenericViewSet
):
    get_list_serializer_class = MyListSerializer
    post_list_serializer_class = MyBulkCreateSerializer
```

Back to *collect* and *result* serializers.
Each action may have defined different *collect* and *result* serializer classes.

Variable name pattern: *{action_name}_{type}_serializer_class* (type can be result or collect)
Example:

```
from rest_framework.decorators import action
from rest_framework.response import Response

from audoma.drf import viewsets
from example_app.serializers import (
    MyCreateSerializer,
    MyCustomActionSerializer
)

class MyViewSet(
    mixins.ActionModelMixin,
    viewsets.GenericViewSet
):
    custom_action_collect_serializer = MyModelCreateSerializer
    custom_action_result_serializer = MyModelSerializer

    @action(detail=True, methods=["post"])
    def custom_action(self, request):
        serializer = self.get_serializer(data=request.data, serializer_type="collect")
        serializer.is_valid(raise_exception=True)
        serializer.save()
        response_serializer = self.get_result_serializer(instance=serializer.instance)
        return Response(response_serializer.data, status_code=201)
```

The most atomic way of defining serializer classes in audoma is to define serializer
per method, action and type.
This means that each action's HTTP method will have *result* and *collect* serializer classes.

Variable name pattern: *{htp_method}_{action_name}_{type}_serializer_class* (type can be result or collect)
Example:

```
from rest_framework.decorators import action
from rest_framework.response import Response

from audoma.drf import viewsets
from audoma.drf import mixins
from example_app.serializers import (
    MyListSerializer,
    MySerializer,
    MyCreateSerializer
```

(continues on next page)

(continued from previous page)

```
)

class MyViewSet(
    mixins.ActionModelMixin,
    mixins.ListModelMixin,
    viewsets.GenericViewSet
):
    get_new_action_result_serializer_class = MyListSerializer
    post_new_action_result_serializer_class = MySerializer
    post_new_action_collect_serializer_class = MyCreateSerializer

    @action(detail=True, methods=["post", "get"])
    def new_action(self, request, *args, **kwargs):
        if request.method == "POST":
            serializer = self.get_serializer(data=request.data, serializer_type="collect
↪")
            serializer.is_valid(raise_exception=True)
            serializer.save()
            instance = serializer.instance
        else:
            instance = self.get_object()
        response_serializer = self.get_result_serializer(instance=instance)
        return Response(response_serializer.data, status_code=201)
```

As you surely presume, all of those serializer classes
variables may be defined on one viewset at once
Then those will be traversed in the defined order.
The first one matching will be used.

Let's have a look at an example viewset:

```
from rest_framework.decorators import action
from rest_framework.response import Response

from audoma.drf import viewsets
from example_app.serializers import (
    MySerachCollectSerializer,
    MySearchResultSerializer,
    MyCountCreateSerializer,
    MyCountUpdateSerializer,
    MyCountResultSerializer,
    MyDefaultSerializer
)
from example_app.models import (
    MyModel,
    CountModel
)

```

(continues on next page)

(continued from previous page)

```python

class MyViewSet(
    mixins.ActionModelMixin,
    mixins.CreateModelMixin,
    mixins.RetrieveModelMixin,
    mixins.DestroyModelMixin,
    mixins.ListModelMixin,
    viewsets.GenericViewSet,
):

    queryset = MyModel.objects.all()

    post_search_collect_serializer_class = MySerachCollectSerializer
    post_search_result_serializer_class = MySearchResultSerializer

    post_count_collect_serializer_class = MyCountCreateSerializer
    put_count_collect_serializer_class = MyCountUpdateSerializer
    count_result_serializer_class = MyCountResultSerializer

    serializer_class = MyDefaultSerializer

    def get_object(self, pk=None):
        return self.querset.get(pk=pk)

    @action(detail=False, methods=["post"])
    def search(self, request):
        serializer = self.get_serializer(data=request.data, serializer_type="collect")
        serializer.is_valid(raise_exception=True)
        serializer.save()
        result_serializer = self.get_result_serializer(instance=serializer.instance)
        return Response(result_serializer.data, status=201)

    @action(detail=True, methods=["post", "get", "put"])
    def count(self, request, *args, **kwargs):
        code = 200
        if request.method != "GET":
            serializer = self.get_serializer(data=request.data, serializer_type="collect
↪")
            serializer.is_valid(raise_exception=True)
            serializer.save()
            instance = serializer.instance
            code = 201 if request.method == "POST"
        else:
            instance = CountModel.objects.get_count(slug=kwargs.pop("slug"))

        result_serializer = self.get_result_serializer(instance=instance)
        return Response(result_serializer.data, status=code)
```

Let's examine the above example.
Action search has two serializers defined, both are defined for the POST method.
One of those will be used to collect data, the other to return the result.

In this case we may also simplify the serializer classes variable names,
because search only serves the POST method, so we may also name those variables like this:

```
...
search_collect_serializer_class = MySerachCollectSerializer
search_result_serializer_class = MySearchResultSerializer
...
```

This will work the same way as serializer classes defined in the example.

For the *count* action we have defined three serializers.
First two serializers handle collecting data for "*POST* and *PUT* HTTP methods.
The third serializer is common for all served by *count* HTTP methods, it is a result serializer.
No matter which method we will use, this is the serializer that will be used to return the result.
In this case, if there won't be further changes in *count* action
we may define *count_result_serializer_class* as *count_serializer_class*.
This will work the same way because of the name traversing order defined in audoma.
But this solution may be problematic during introducing any changes.

```
...
post_count_collect_serializer_class = MyCountCreateSerializer
put_count_collect_serializer_class = MyCountUpdateSerializer
count_serializer_class = MyCountResultSerializer
...
```

The one last thing that is left in this viewset is *serializer_class*.
This variable will be used by all other actions supported by this viewset.
In the viewset definition there are few mixin classes passed, so those will
provide some basic functionalities to our viewset.

If this is going to be necessary it is possible to create a separate serializer for those actions also.

Example:

```
from rest_framework.decorators import action
from rest_framework.response import Response

from audoma.drf import viewsets
from example_app.serializers import (
    MySerachCollectSerializer,
    MySearchResultSerializer,
    MyCountCreateSerializer,
```

(continues on next page)

(continued from previous page)

```
    MyCountUpdateSerializer,
    MyCountResultSerializer,
    MyDefaultSerializer,
    MyListSerializer,
    MyCreateSerializer
)
from example_app.models import (
    MyModel,
    CountModel
)


class MyViewSet(
    mixins.ActionModelMixin,
    mixins.CreateModelMixin,
    mixins.RetrieveModelMixin,
    mixins.DestroyModelMixin,
    mixins.ListModelMixin,
    viewsets.GenericViewSet,
):

    queryset = MyModel.objects.all()

    post_search_collect_serializer_class = MySerachCollectSerializer
    post_search_result_serializer_class = MySearchResultSerializer

    post_count_collect_serializer_class = MyCountCreateSerializer
    put_count_collect_serializer_class = MyCountUpdateSerializer
    count_result_serializer_class = MyCountResultSerializer

    list_serializer_class = MyListSerializer
    create_serializer_class = MyCreateSerializer
    serializer_class = MyDefaultSerializer

    def get_object(self, pk=None):
        return self.querset.get(pk=pk)

    @action(detail=False, methods=["post"])
    def search(self, request):
        serializer = self.get_serializer(data=request.data, serializer_type="collect")
        serializer.is_valid(raise_exception=True)
        serializer.save()
        result_serializer = self.get_result_serializer(instance=serializer.instance)
        return Response(result_serializer.data, status=201)

    @action(detail=True, methods=["post", "get", "put"])
    def count(self, request, *args, **kwargs):
    code = 200
        if request.method != "GET":
            serializer = self.get_serializer(data=request.data, serializer_type="collect
↪")
            serializer.is_valid(raise_exception=True)
```

(continues on next page)

(continued from previous page)

```
            serializer.save()
            instance = serializer.instance
            code = 201 if request.method == "POST"
        else:
            instance = CountModel.objects.get_count(slug=kwargs.pop("slug"))

        result_serializer = self.get_result_serializer(instance=instance)
        return Response(result_serializer.data, status=code)
```

2.1.2 Serializer classes name traverse order

After examining the above examples, it is obvious that there is some defined order while traversing defined variables. The variable which will be used as the serializer class is being picked in this order:

- *{htp_method}_{action_name}_{type}_serializer_class* (type can be result or collect)
- *{action_name}_{type}_serializer_class* (type can be result or collect)
- *{http_method}_{action_name}_serializer_class*
- *{action_name}_serializer_class*
- *common_{type}_serializer_class* (type can be result or collect)
- *serializer_class*

For all serializers defined this way, there is also support for proper documentation in api schema.

2.2 Permissions

By default, in the *drf-spectacular* viewset permissions were not documented at all. In audoma, permissions are being documented for each viewset separately.

You don't have to define anything extra, this is being handled just out of the box. The only thing it is required is to define permissions on your viewset.

Example:

```
from rest_framework.decorators import action
from rest_framework.response import Response

from audoma.drf import viewsets
from example_app.serializers import (
    MySerachCollectSerializer,
    MySearchResultSerializer,
    MyCountCreateSerializer,
    MyCountUpdateSerializer,
    MyCountResultSerializer,
    MyDefaultSerializer,
```

(continues on next page)

(continued from previous page)

```
    MyListSerializer,
    MyCreateSerializer
)
from example_app.permissions import (
    AlternatePermission1,
    AlternatePermission2,
    DetailPermission,
    ViewAndDetailPermission,
    ViewPermission,
)
from example_app.models import (
    MyModel,
    CountModel
)


class MyViewSet(
    mixins.ActionModelMixin,
    mixins.CreateModelMixin,
    mixins.RetrieveModelMixin,
    mixins.DestroyModelMixin,
    mixins.ListModelMixin,
    viewsets.GenericViewSet,
):
    permission_classes = [
        IsAuthenticated,
        ViewAndDetailPermission,
        DetailPermission,
        ViewPermission,
        AlternatePermission1 | AlternatePermission2,
    ]

    queryset = MyModel.objects.all()

    post_search_collect_serializer_class = MySerachCollectSerializer
    post_search_result_serializer_class = MySearchResultSerializer

    post_count_collect_serializer_class = MyCountCreateSerializer
    put_count_collect_serializer_class = MyCountUpdateSerializer
    count_result_serializer_class = MyCountResultSerializer

    list_serializer_class = MyListSerializer
    create_serializer_class = MyCreateSerializer
    serializer_class = MyDefaultSerializer

    def get_object(self, pk=None):
        return self.querset.get(pk=pk)

    @action(detail=False, methods=["post"])
    def search(self, request):
        serializer = self.get_serializer(data=request.data, serializer_type="collect")
        serializer.is_valid(raise_exception=True)
```

(continues on next page)

(continued from previous page)

```
        serializer.save()
        result_serializer = self.get_result_serializer(instance=serializer.instance)
        return Response(result_serializer.data, status=201)

    @action(detail=True, methods=["post", "get", "put"])
    def count(self, request, *args, **kwargs):
    code = 200
        if request.method != "GET":
            serializer = self.get_serializer(data=request.data, serializer_type="collect
↪")
            serializer.is_valid(raise_exception=True)
            serializer.save()
            instance = serializer.instance
            code = 201 if request.method == "POST"
        else:
            instance = CountModel.objects.get_count(slug=kwargs.pop("slug"))

        result_serializer = self.get_result_serializer(instance=instance)
        return Response(result_serializer.data, status=code)
```

Currently there is no way to customize this behavior in audoma, also it is not possible to disable permissions documentation.

2.3 Custom choices

Audoma provides a new way of defining choices and new choices class which allows calling choice by its name.

Example definition and usage:

```
from audoma.django.db import models
from audoma.choices import make_choices


class CarModel(models.Model):


    CAR_BODY_TYPES = make_choices(
        "BODY_TYPES",
        (
            (1, "SEDAN", "Sedan"),
            (2, "COUPE", "Coupe"),
            (3, "HATCHBACK", "Hatchback"),
            (4, "PICKUP", "Pickup Truck"),
        ),
    )
```

(continues on next page)

(continued from previous page)

```

    name = models.CharField(max_length=255)
    body_type = models.IntegerField(choices=CAR_BODY_TYPES.get_choices())

    engine_size = models.FloatField()

    def is_sedan(self):
        return self.body_type is BODY_TYPE_CHOICES.SEDAN
```

Additionally it's worth mentioning that those choices will be shown in docs in the fields description.
Those will also appear in the schema as *x-choices*.

2.4 Filters

2.4.1 Default Filters

In *drf*, it's possible to define *filterset_fields* and *filterset_class*.
By default, *drf-spectacular`* supports *django-filters*. Which are being documented.
Audoma has been tested with the default DRFs filter backend and
django_filters.rest_framework.DjangoFilterBackend.
For more accurate documentation, we recommend using *django_filters.rest_framework.DjangoFilterBackend* as the default one.
Filters and search fields are being documented out of the box.

Example:

```
from rest_framework.filters import SearchFilter
from audoma.drf import mixins
from audoma.drf import viewsets
from django_filters import rest_framework as df_filters

from example_app.models import CarModel
from example_app.serializers import CarModelSerializer

class CarViewSet(
    mixins.ActionModelMixin,
    mixins.RetrieveModelMixin,
    mixins.ListModelMixin,
    viewsets.GenericViewSet,
):
    queryset = CarModel.objects.all()
    serializer_class = CarModelSerializer

    filter_backends = [SearchFilter, df_filters.DjangoFilterBackend]

```

(continues on next page)

(continued from previous page)

```
    filterset_fields = ["body_type"]
    search_fields = ["=manufacturer", "name"]
```

It is also possible to define the *filterset* class which will also be documented without any additional steps.

```
from rest_framework.filters import SearchFilter
from audoma.drf import mixins
from audoma.drf import viewsets
from django_filters import rest_framework as df_filters

from example_app.models import CarModel
from example_app.serializers import CarModelSerializer


class CarFilter(df_filters.FilterSet):
    body_type = df_filters.TypedChoiceFilter(
        Car.CAR_BODY_TYPES.get_choices(), "body_type",
        lookup_expr="exact", field_name="body_type"
    )

    class Meta:
        model = CarModel
        fields = [
            "body_type",
        ]


class CarViewSet(
    mixins.ActionModelMixin,
    mixins.RetrieveModelMixin,
    mixins.ListModelMixin,
    viewsets.GenericViewSet,
):
    queryset = CarModel.objects.all()
    serializer_class = CarModelSerializer

    filter_backends = [SearchFilter, df_filters.DjangoFilterBackend]

    filterset_class = CarFilter
    search_fields = ["=manufacturer", "name"]
```

Audoma extends documenting filters with two main features.

Additional enum documentation in field description:
In *drf-spectacular*, enums are being shown only as values possible to pass to the filter.
With audoma, you also get a display value of enum field.

This is being shown as: * api value - display value

The next feature is schema extension which is not visible in OpenApi frontend.
This schema extension is *x-choices*. Which provides mapping for filter values.
Passing x-choices in schema allows frontend developers to use mapping
to show display/value fields without looking into a field description.

2.5 Validators

2.5.1 ExclusiveFieldsValidator

This is an additional validator, which allows defining mutually exclusive fields in the serializer.
It validates if any of the fields have been given and if not all exclusive fields have been given.

This validator takes params:

- fields - list or a tuple of field names
- message - string message, which will replace the default validator message
- required - boolean which determines if any of the fields must be given
- message_required - a message which will be displayed if one of the fields is required, and none has been passed

Usage is simple:

```
from audoma.drf import serializers
from audoma.drf.validators import ExclusiveFieldsValidator


class MutuallyExclusiveExampleSerializer(serializers.Serializer):
    class Meta:
        validators = [
            ExclusiveFieldsValidator(
                fields=[
                    "example_field",
                    "second_example_field",
                ]
            ),
        ]

    example_field = serializers.CharField(required=False)
    second_example_field = serializers.CharField(required=False)
```

2.6 Decorators

2.6.1 @extend_schema_field

Spectacular Docs

This decorator is by default *drf-spectacular* feature.
Audoma only changes its behavior, in *drf-spectacular* using this decorator causes overriding
all informations about the field. Audoma does not override information, it only updates available information
with those passed to the decorator.

This may be very useful while defining examples.
We don't want to erase all other field information
just because we want to define an example for this field.
Also passing all field information additionally just because we want
to define an example seems unnecessary and redundant.

Example:

```
from audoma.drf.fields import FloatField

from drf_spectacular.utils import extend_schema_field

@extend_schema_field(
    field={
        "example": 10.00
    }
)
class CustomExampleFloatField(FloatField):
    pass
```

Above we simply add a default example for all
fields which will be of class *CustomExampleFloatField*.

2.6.2 @audoma_action

DRFs action docs <https://www.django-rest-framework.org/api-guide/viewsets/#marking-extra-actions-for-routing>

This is one of the most complex features offered by audoma, an extension of an action decorator.
Decorator by default is Django Rest Framework functionality.
It also allows registering custom action for viewset.
In the case of *audoma_action*, it changes a bit how the action function should work,
using *audoma_action* action function should not return a *Response* object, it should return

tuple of instance and status code, *audoma_action* will take care of creating response out of it.

Usage

Let's take an example viewset:

```
from audoma.drf import mixins
from audoma.drf import viewsets

from app.serializers import (
    CarListSerializer,
    CarWriteSerializer,
    CarDetailsSerializer,
    CarCreateRateSerializer,
    CarRateSerializer
)
from app.models import (
    Car,
    CarRate
)


class CarViewSet(
    mixins.ActionModelMixin,
    mixins.CreateModelMixin,
    mixins.RetrieveModelMixin,
    mixins.ListModelMixin,
    viewsets.GenericViewSet,
):

    permission_classes = [
        IsAuthenticated,
        ViewAndDetailPermission,
        DetailPermission,
        ViewPermission,
        AlternatePermission1 | AlternatePermission2,
    ]

    create_collect_serializer_class = CarWriteSerializer
    create_result_serializer_class = CarDetailsSerializer
    retrieve_serializer_class = CarDetailsSerializer
    list_serializer_class = CarListSerializer

    queryset = {}
    @audoma_action(
        detail=True,
        methods=["get", "post"]
        collectors=CarCreateRateSerializer,
        results=CarRateSerializer,
        errors=[CustomCarRateException]
    )
```

(continues on next page)

(continued from previous page)

```
    def rate(self, request, pk=None, *args, **kwargs):
        if request.method == "POST":
            collect_serializer = kwargs.pop("collect_serializer")
            instance = collect_serializer.save()
        else:
            instance = CarRate.objects.get_random_car_rate(car_pk=pk)
        return instance, 200
```

Let's examine the above example.
We've created the viewset with some initial actions served, and serializers assigned to those actions.

Next we've defined a new custom action called *rate*.
This action serves *get* and *post* methods, in case of this action '
we use a single result and collect serializers.

As you may see, *audoma_action* method does not return the default response, it returns
instance and status_code, the *audoma_action* decorator takes care
of creating the response from this.

Let's modify our example, let there be a custom exception raised.

```
from audoma.drf import mixins
from audoma.drf import viewsets
from rest_framework.exceptions import APIException

from app.serializers import (
    CarListSerializer,
    CarWriteSerializer,
    CarDetailsSerializer,
    CarCreateRateSerializer,
    CarRateSerializer
)
from app.models import (
    Car,
    CarRate
)


class CustomCarRateException(APIException):
    default_detail = "Error during retrieving car rate!"
    status_code = 500


class CarViewSet(
    mixins.ActionModelMixin,
```

(continues on next page)

(continued from previous page)

```
    mixins.CreateModelMixin,
    mixins.RetrieveModelMixin,
    mixins.ListModelMixin,
    viewsets.GenericViewSet,
):

    permission_classes = [
        IsAuthenticated,
        ViewAndDetailPermission,
        DetailPermission,
        ViewPermission,
        AlternatePermission1 | AlternatePermission2,
    ]

    create_collect_serializer_class = CarWriteSerializer
    create_result_serializer_class = CarDetailsSerializer
    retrieve_serializer_class = CarDetailsSerializer
    list_serializer_class = CarListSerializer

    queryset = {}

    @audoma_action(
        detail=True,
        methods=["get", "post"]
        collectors=CarCreateRateSerializer,
        results=CarRateSerializer,
        errors=[CustomCarRateException]
    )
    def rate(self, request, pk=None, *args, **kwargs):
        if request.method == "POST":
            collect_serializer = kwargs.pop("collect_serializer")
            instance = collect_serializer.save()
        else:
            instance = CarRate.objects.get_random_car_rate(car_pk=pk)
            if not instance:
                raise CustomCarRateException
        return instance, 200
```

After this change it is possible to raise any exception of type *CustomCarRateException* in rate action.
Also this exception will be documented in this action schema.

Let's presume that we now want to return status code *201* and rate instance on *post*,
but on *get* we want to return the car instance with random rate and status code *200*.

```
from audoma.drf import mixins
from audoma.drf import viewsets
from rest_framework.exceptions import APIException

```

(continues on next page)

(continued from previous page)

```
from app.serializers import (
    CarListSerializer,
    CarWriteSerializer,
    CarDetailsSerializer,
    CarCreateRateSerializer,
    CarRateSerializer
)
from app.models import (
    Car,
    CarRate
)


class CustomCarException(APIException):
    default_detail = "Car can't be found"
    status_code = 500


class CarViewSet(
    mixins.ActionModelMixin,
    mixins.CreateModelMixin,
    mixins.RetrieveModelMixin,
    mixins.ListModelMixin,
    viewsets.GenericViewSet,
):

    permission_classes = [
        IsAuthenticated,
        ViewAndDetailPermission,
        DetailPermission,
        ViewPermission,
        AlternatePermission1 | AlternatePermission2,
    ]

    create_collect_serializer_class = CarWriteSerializer
    create_result_serializer_class = CarDetailsSerializer
    retrieve_serializer_class = CarDetailsSerializer
    list_serializer_class = CarListSerializer

    queryset = {}

    @audoma_action(
        detail=False,
        methods=["get", "post"]
        collectors=CarCreateRateSerializer,
        results={"post":{201: CarRateSerializer}, "get":{200: CarDetailsSerializer}},
        errors=[CustomCarException]
    )
    def rate(self, request, *args, **kwargs):
        if request.method == "POST":
            collect_serializer = kwargs.pop("collect_serializer")
            instance = collect_serializer.save()
```

(continues on next page)

(continued from previous page)

```
            return instance. 201
        else:
            instance = car.objects.get(pk=pk)
            if not instance:
                raise CustomCarException
            return instance, 200
```

Now we use different a serializer for each method, depending on returned status code.
Each of this serializer is using different model, *audoma_action* makes such situations super easy.

Let's take a different example, we have an action that should return a string message, depending on current car state.

```
from audoma.drf import mixins
from audoma.drf import viewsets
from rest_framework.exceptions import APIException

from app.serializers import (
    CarListSerializer,
    CarWriteSerializer,
    CarDetailsSerializer,
    CarCreateRateSerializer,
    CarRateSerializer
)
from app.models import (
    Car,
    CarRate
)


class CustomCarException(APIException):
    default_detail = "Car can't be found"
    status_code = 500


class CarViewSet(
    mixins.ActionModelMixin,
    mixins.CreateModelMixin,
    mixins.RetrieveModelMixin,
    mixins.ListModelMixin,
    viewsets.GenericViewSet,
):

    permission_classes = [
        IsAuthenticated,
        ViewAndDetailPermission,
        DetailPermission,
        ViewPermission,
```

(continues on next page)

(continued from previous page)

```
        AlternatePermission1 | AlternatePermission2,
    ]

    create_collect_serializer_class = CarWriteSerializer
    create_result_serializer_class = CarDetailsSerializer
    retrieve_serializer_class = CarDetailsSerializer
    list_serializer_class = CarListSerializer

    queryset = {}

    @audoma_action(
        detail=False,
        methods=["get", "post"]
        collectors=CarCreateRateSerializer,
        results={"post":{201: CarRateSerializer}, "get":{200: CarDetailsSerializer}},
        errors=[CustomCarException]
    )
    def rate(self, request, *args, **kwargs):
        if request.method == "POST":
            collect_serializer = kwargs.pop("collect_serializer")
            instance = collect_serializer.save()
            return instance. 201
        else:
            instance = car.objects.get(pk=pk)
            if not instance:
                raise CustomCarException
            return instance, 200


    @audoma_action(
        detail=False,
        methods=["get"],
        results="Car is available"
    )
    def active(self, request, pk=None):
        instance = self.get_object(pk=pk)
        if instance.active:
            return None, 200
        return "Car is unavailable", 200
```

This action may return *None* or *string*, but as you may see in the results we have also string defined.
The string default in the results is a string that will be the message returned by default.
The default message will be returned if the instance is *None*.
If returned string instance won't be None, then the returned instance will be
included in the response.

While returning string message as an instance, audoma simply wraps this message into *json*.
Wrapped message would look like this:

```
{
    "message": "Car is available"
}
```

We can combine those results, so in one action
we may return string instance and model instance.
Let's modify our rate function, so it'll return the default message if the rating is disabled.

```
from audoma.drf import mixins
from audoma.drf import viewsets
from rest_framework.exceptions import APIException
from django.conf import settings

from app.serializers import (
    CarListSerializer,
    CarWriteSerializer,
    CarDetailsSerializer,
    CarCreateRateSerializer,
    CarRateSerializer
)
from app.models import (
    Car,
    CarRate
)


class CustomCarException(APIException):
    default_detail = "Car can't be found"
    status_code = 500


class CarViewSet(
    mixins.ActionModelMixin,
    mixins.CreateModelMixin,
    mixins.RetrieveModelMixin,
    mixins.ListModelMixin,
    viewsets.GenericViewSet,
):

    permission_classes = [
        IsAuthenticated,
        ViewAndDetailPermission,
        DetailPermission,
        ViewPermission,
        AlternatePermission1 | AlternatePermission2,
    ]

    create_collect_serializer_class = CarWriteSerializer
    create_result_serializer_class = CarDetailsSerializer
```

(continues on next page)

(continued from previous page)

```
    retrieve_serializer_class = CarDetailsSerializer
    list_serializer_class = CarListSerializer

    queryset = {}

    @audoma_action(
        detail=False,
        methods=["get", "post"]
        collectors=CarCreateRateSerializer,
        results={
            "post":{201: CarRateSerializer},
            "get":{200: CarDetailsSerializer, 204:"Rate service currently unavailable"}
        },
        errors=[CustomCarException]
    )
    def rate(self, request, *args, **kwargs):
        if settings.RATE_AVAILABLE:
            return None, 204

        if request.method == "POST":
            collect_serializer = kwargs.pop("collect_serializer")
            instance = collect_serializer.save()
            return instance. 201
        else:
            instance = car.objects.get(pk=pk)
            if not instance:
                raise CustomCarException
            return instance, 200


    @audoma_action(
        detail=False,
        methods=["get"],
        results="Car is available"
    )
    def active(self, request, pk=None):
        instance = self.get_object(pk=pk)
        if instance.active:
            return None, 200
        return "Car is unavailable", 200
```

Params

Decorator *audoma_action* takes all params which may be passed to the *action* decorator.
It also takes additional params, which we will describe below:

collectors

This param allows defining serializer class which will collect and process request data.
To define this, action must serve POST/PATCH or PUT method, otherwise
defining those will cause an exception.
Collectors may be passed as:

- Serializer class which must inherit from *serializers.BaseSerializer*

```
@audoma_action(
    detail=False,
    methods=["post"],
    results=ExampleOneFieldSerializer,
    collectors=ExampleOneFieldSerializer,
)
```

- A dictionary with HTTP methods as keys and serializer classes as values. This allows defining different collector for each HTTP method.

```
@audoma_action(
    detail=True,
    methods=["post"],
    collectors={"post": ExampleModelCreateSerializer},
    results=ExampleModelSerializer,
)
```

If you are using PATCH or PUT method for your action, you may ask how to pass an instance
to your collect serializer. You simply have to override *get_object* method on your viewset, and make
it return the object you want to pass to collect serializer as an instance for given action and method.

Note:

Passing collectors is optional, so you don't have to pass them.
If collectors won't be passed, and request method will be in *[PUT, POST, PATCH]*
then by default, *audoma_action* fill fallback to default
get_serializer_class method for audoma.

Note:

If you are using collectors it is important to remember,
that your action method should accept additional kwarg *collect_serializer*
which will be a validated collector instance.

---

results

This param allows defining custom results for each method and each response status code.
Results param may be passed as:

- Serializer class or which must inherit from *serializers.BaseSerializer* or string variable In this case, the serializer class passed will be used to produce every response coming from this action.

```
@audoma_action(
    detail=True,
    methods=["put", "patch"],
    collectors=ExampleModelCreateSerializer,
    results=ExampleModelSerializer,
)
```

- A dictionary with HTTP methods as keys and serializer classes or string variables as values. In This case, there will be a different response serializer for each HTTP method.

```
@audoma_action(
    detail=False,
    methods=["get", "post"],
    collectors={"post": MyCreateSerializer},
    results={"post": MySerializer, "get": MyListSerializer}
)
```

- A dictionary with HTTP methods as keys and dictionaries as values. Those dictionaries have status codes as keys and serializer classes or string variables as values.

```
@audoma_action(
    detail=False,
    methods=["post"],
    collectors={"post": MyCreateSerializer},
    results={"post": {201: MySerializer, 204: MyNoContentSerializer}}
)
```

Note:

Results param is not mandatory, if you won't pass the results
param into audoma_action, then there will be a fallback to default
Viewset defined serializers.

errors

This param is a list of classes and instances of exceptions,
which are allowed to rise in this action.
Such behavior prevents rising, not defined exceptions, and allows to document
exceptions properly in OpenApi schema.

The main difference between passing exception class and exception instance, is that
if you pass exception instance, audoma will not only check if exception
type matches, it'll also validate its content.
We presume that if you pass, the exception class, you want to accept all exceptions of this class.

In case the risen exception is not defined in audoma_action errors, there will be another
exception risen: AudomaActionException, in case the settings.DEBUG = False, this exception
will be handled silently by logging it, but the code will pass.
In the case of *settings.DEBUG = True*, then the exception won't be silent.

By default audoma accepts some exceptions, which are defined globally.
Those exceptions are:

- NotFound
- NotAuthenticated
- AuthenticationFailed
- ParseError
- PermissionDenied

If you want to extend this list of globally accepted exceptions, you can do it by
defining *COMMON_API_ERRORS* in your settings, for example:

```
COMMON_API_ERRORS = [
    myexceptions.SomeException
]
```

Note:

Errors param is optional, but if they won't be passed, action will only
allow rising globally defined exceptions.

ignore_view_collectors

Boolean variable which tells if audoma_action should fallback to
default way of retrieving collector from view, if the collector has not been passed
and action use method which allows collecting serializer usage.

many

This param decides if the returned instance should be treated as *many* by a serializer
Currently it can only be set to the concrete action, it is impossible to return a instance and
multiple instances from one action method using *audoma_action*.

2.7 Examples

2.7.1 Define an example for the field

Above we described *@extend_schema_field* decorator which allows defining example for the field.
For all fields defined in audoma, there are examples generated automatically,
but you may also pass your example as a field parameter.

Example:

```
from audom.drf import serializers

class SalesContactSerializer(serializers.Serializer):
    phone_number = serializers.PhoneNumberField(example="+48 123 456 789")
    name = serializers.CharField(example="John", max_length=255)
```

After passing the example, it'll be the value shown in example requests in docs.

2.7.2 Define custom fields with auto-generated examples

If you want to define your field with auto example generation,
it is possible, that your field class should inherit from the base *ExampleMixin* class,
set proper example class.

```
from rest_framework import fields
from audoma.mixins import ExampleMixin
from audoma.examples import NumericExample,
```

(continues on next page)

(continued from previous page)

```
class SaleAmountField(ExampleMixin, fields.Field):
    audoma_example_class = NumericExample
```

2.7.3 Define custom example classes

It is possible to define your custom example classes, by default audio has defined
two specific example classes inside the *audoma.examples* module:

- *NumericExample*
- *RegexExample*

And one base class:

- *Example*

To define your example class, you should inherit from the *Example* class
and override the *generate_value* method

```
from audoma.examples import Example

class SaleExample(Example):
    def generate_value(self):
        return f"{self.amount} $"
```

2.8 Extra Fields

2.8.1 Money Field

django_money docs

Our money field is an extension of the *MoneyField* known from *django_money*.
This field is defined as one field in the model, but it creates two fields in the database.
There is nothing complex in this field usage, simply define it in your model:

```
from audoma.django.db import models

class SalesmanStats(models.Model):
    salesman = models.ForeignKey("sale.Salesman"e, on_delete=models.CASCADE)
    earned = models.MoneyField(max_digits=14, decimal_places=2, default_currency="PLN")
```

Field defined on the model required passing to it two variables.
Currency and amount, in our case we have set the default currency, so passing currency is not obligatory.
Those values may be passed in a few ways:

```
stats = SalesmanStats.objects.get(id=20)
# Simply pass the Money object
stats.earned = Money("99900.23", "PLN")
# You may also pass those variables to objects.create separately
sales = Salesman.objects.get(id=1)
stats = SalesmanStats.objects.create(
    salesman=sales, earned_amount=120,
    earned_courrency="PLN"
)
# In our case we defined the default currency, so it also may be
stats = SalesmanStats.objects.create(
    salesman=sales, earned_amount=120
)
# To get the amount we type
print(stats.earned) # this will print 120
print(stats.earned.currency) # will print PLN
```

2.8.2 PhoneNumberField

django-phonenumber-field docs

Audoma provides a *PhoneNumberField* which is an extension of the *django-phonenumber-field*.
You can use it in your models straight away, just as the original PhoneNumberField,
and what we added here is an automatically generated example in documentation,
based on country code.

Example:

```
from audoma.django.db import models

class SalesmanStats(models.Model):
    salesman = models.ForeignKey("sale.Salesman", on_delete=models.CASCADE)
    earned = models.MoneyField(max_digits=14, decimal_places=2, default_currency="PLN")
    phone_number = models.PhoneNumberField(region="GB")
```

The above code will result in the following example in the documentation:

```
{
    "salesman": 1,
    "earned": 500,
```

(continues on next page)

(continued from previous page)

```
    "phone_number": "+44 20 7894 5678",
}
```

2.9 Serializer Field links

Audoma allows defining links for serializer fields, which values
are related to other endpoints. This is useful if you want to limit value choices to
other filtered endpoint lists.

Such link won't be visible in redoc/swagger frontend.
It'll be included in OpenApi schema as *x-choices*.

Link definition:

- viewname - the name of a view from which variables should be retrieved
- value_field - field name from which value should be retrieved
- display_field - field name from which display value should be retrieved

2.10 Schema Extensions

2.10.1 x-choices

This extension is being added to all fields schema which have limited choice to some range.
All fields which have defined choices as enum will have this included in their schema.
If the filter field is also limited to choices this also will be included.

X-choices may have two different forms.
The first one when it's just a representation of choices enum.
Then it'll be a mapping:

```
{
    "x-choices": {
        "choices": {
            "value1": "displayValue1",
            "value2": "displayValue2",
            "value3": "displayValue3",
            "value4": "displayValue4",
        }
    }
}
```

This is simply a mapping of values to display values.
This may be useful during displaying choices in for example drop-down.

The second form of x-choices is:

```
{
    "x-choices": {
        "operationRef": "#/paths/manufacturer_viewset~1",
        "value": "$response.body#results/*/id",
        "display": "$response.body#results/*/name"
    }
}
```

This x-choices is a reference to a different endpoint.
This may be used to read limited choices from the related endpoint.
* operationRef - is a JSON pointer to the related endpoint which should be accessible in this chema
* value - shows which field should be taken as a field value
* display - shows which field should be taken as field display value

CHAPTER

THREE

CONTRIBUTING TO AUDOMA

Audoma as an open source project welcomes any form of contribution.
Every contribution is important for us, even if it's a small one

There are different types of contributions

- Documentation fixes, clarifications or improvements
- Creating pull requests
- Creating issues as feature requests or bug reports
- Questions that highlight inconsistencies or workflow issues

3.1 Issues

Please include to each submitted issue: * Brief description of the issue * Stack trace if there has been an exception raised * Code that caused the issue * audoma/drf-spectacular/Django/DRF versions

3.2 Pull requests

We also welcome pull requests with fixes or new features.
We strongly advise you to stick to these rules, during creating pull requests: * Test your changes before creating pull requests, your code has to pass the whole test suite to get merged. The best way will be by launching docker tests, included with the project * Use linters included in audoma * Write your own tests for your changes * If your changes are not trivial consider creating an issue first with changes proposal to get some early feedback * If you are introducing a new feature, please add an example for this feature in audoma example application

3.3 Running unit tests

To test audoma we are using tox with multiple Django/DRF/python configurations.
To run tests enter audoma root dir and simply use:
docker-compose -f docker/docker-compose.yml up tests
This will run the whole testing process.

3.4 Example application

Audoma has included an example Django/DRF application to demonstrate its possibilities.
Every supported feature should have its representation in an example application

To star an example application, from the root folder,
simply run *docker-compose -f docker/docker-compose.yml up example_app*.

As mentioned above the example application should illustrate every feature of audoma.
If you are going to introduce a new feature it would be nice of you
to include an example of this feature in the example application, this should
make it easier to keep the project consistent.

CHAPTER

FOUR

LICENSE

CHAPTER

FIVE

CHANGELOG

CHAPTER

SIX

OVERVIEW

6.1 audoma.choices

audoma.choices.**make_choices**(*name: str*, *choices_tuple: Tuple[Any, str, str]*) → audoma.choices._T

6.2 audoma.decorators

audoma_action.**__init__**(*collectors: Optional[Union[Dict[str, Dict[int, Union[str, Type[rest_framework.serializers.BaseSerializer]]]], Dict[str, Union[str, Type[rest_framework.serializers.BaseSerializer]]], str, Type[rest_framework.serializers.BaseSerializer]]] = None*, *results: Optional[Union[Dict[str, Dict[int, Union[str, Type[rest_framework.serializers.BaseSerializer]]]], Dict[str, Union[str, Type[rest_framework.serializers.BaseSerializer]]], str, Type[rest_framework.serializers.BaseSerializer]]] = None*, *errors: Optional[List[Union[Exception, Type[Exception]]]] = None*, *many: bool = False*, *ignore_view_collectors: bool = False*, ***kwargs*) → None

audoma_action.**__call__**(*func: Callable*) → Callable

” Call of audoma_action decorator. This is where the magic happens.

Args: func - decorated function

Returns: wrapper callable.

class audoma.decorators.**AudomaActionException**

class audoma.decorators.**AudomaArgs**(*results: Union[Dict[str, Dict[int, Union[str, Type[rest_framework.serializers.BaseSerializer]]]], Dict[str, Union[str, Type[rest_framework.serializers.BaseSerializer]]], str, Type[rest_framework.serializers.BaseSerializer]]*, *collectors: Union[Dict[str, Dict[int, Union[str, Type[rest_framework.serializers.BaseSerializer]]]], Dict[str, Union[str, Type[rest_framework.serializers.BaseSerializer]]], str, Type[rest_framework.serializers.BaseSerializer]]*, *errors: List[Union[Exception, Type[Exception]]]*, *many: bool*)

6.3 audoma.examples

class audoma.examples.**Example**(*field*, *example=<class 'audoma.examples.DEFAULT'>*)
Class that represents an example for a field. It allows to add example to the field during initialization.

generate_value() → Type[audoma.examples.DEFAULT]

get_value() → Any

to_representation(*value*) → Any

class audoma.examples.**NumericExample**(*field*, *example=<class 'audoma.examples.DEFAULT'>*)

generate_value() → float
Extracts information from the field and generates a random value based on min_value and max_value.

Returns: Random value between min_value and max_value

class audoma.examples.**RegexExample**(*field*, *example=<class 'audoma.examples.DEFAULT'>*)

generate_value() → str
Extracts information from the field and generates a random value based on field's RegexValidators.

Returns: Generated regex string if regex is found, otherwise returns None

6.4 audoma.hooks

audoma.hooks.**postprocess_common_errors_section**(*result: dict*, *request*, ***kwargs*) → dict
Postprocessing hook which adds COMMON_API_ERRORS description to the API description.

audoma.hooks.**preprocess_include_path_format**(*endpoints: Tuple[str, str, str, Callable]*, ***kwargs*) → List[Tuple[str, str, str, Callable]]
Preprocessing hook that filters {format} prefdixed paths, in case schema pattern prefix is used and {format} path params are wanted.

6.5 audoma.mixins

class audoma.mixins.**ExampleMixin**(**args*, *example=<class 'audoma.examples.DEFAULT'>*, ***kwargs*)
A mixin class that adds an example to the field in documentation by overriding *field* parameter in *_spectacular_annotation*.

Args:

audoma_example_class [Type[Example]] The class that will be used to create the example. Depends on the type of field

audoma_example_class
alias of *audoma.examples.Example*

class audoma.mixins.**NumericExampleMixin**(**args*, *example=<class 'audoma.examples.DEFAULT'>*, ***kwargs*)
A mixin class that adds an example to the field in documentation for numeric fields

audoma_example_class
alias of *audoma.examples.NumericExample*

class audoma.mixins.**RegexExampleMixin**(**args*, *example=<class 'audoma.examples.DEFAULT'>*, ***kwargs*)

A mixin class that adds an example to the field in documentation for regex fields

audoma_example_class

alias of *audoma.examples.RegexExample*

6.6 audoma.openapi

class audoma.openapi.**AudomaAutoSchema**

_build_exclusive_fields_schema(*schema: dict*, *exclusive_fields: List[str]*) → List[dict]

_extract_action_function(*view*) → Callable

_extract_audoma_action_operations(*view: django.views.generic.base.View*, *serializer_type: str*) → Union[Dict[str, drf_spectacular.utils.OpenApiResponse], Dict[str, rest_framework.serializers.BaseSerializer], Dict[str, Type[rest_framework.serializers.BaseSerializer]], Dict[str, Dict[int, rest_framework.serializers.BaseSerializer]], Dict[int, drf_spectacular.utils.OpenApiResponse]]

Extracts the audoma action operations from the view

_get_enum_choices_for_field(*field*)

_get_link_choices_for_field(*field*, *serializer*)

_get_permissions_description() → str

_get_request_for_media_type(*serializer*)

_get_response_for_code(*serializer*, *status_code*, *media_types=None*)

_get_serializer(*serializer_type: str = 'collect'*) → Union[rest_framework.serializers.BaseSerializer, Type[rest_framework.serializers.BaseSerializer]]

_handle_permission(*permission_class: Union[rest_framework.permissions.OperandHolder, rest_framework.permissions.SingleOperandHolder, rest_framework.permissions.BasePermission]*, *operations: list*, *current_operation: Type = <class 'rest_framework.permissions.AND'>*) → dict

_map_serializer(*serializer: Union[drf_spectacular.types.OpenApiTypes, rest_framework.serializers.BaseSerializer, Type[rest_framework.serializers.BaseSerializer]]*, *direction: str*, *bypass_extensions: bool = False*) → dict

_map_serializer_field(*field: rest_framework.fields.Field*, *direction: str*, *bypass_extensions=False*) → dict

Allows to use @extend_schema_field with *field* dict so that it gets updated instead of being overriden

_parse_action_errors(*action_errors*) → Dict[int, drf_spectacular.utils.OpenApiResponse]

_parse_action_result_serializer(*serializer: Union[Type[rest_framework.serializers.BaseSerializer], str]*, *many: bool = False*) → Union[rest_framework.serializers.BaseSerializer, str]

_parse_action_result_serializers(*action_serializers: Union[Dict[str, Dict[int, Union[str, Type[rest_framework.serializers.BaseSerializer]]]], Dict[str, Union[str, Type[rest_framework.serializers.BaseSerializer]]], str, Type[rest_framework.serializers.BaseSerializer]]*, *many: bool = False*) → Union[Dict[str, drf_spectacular.utils.OpenApiResponse], Dict[str, rest_framework.serializers.BaseSerializer], Dict[str, Dict[int, rest_framework.serializers.BaseSerializer]]]

choice_link_schema_generator = <audoma.links.ChoicesOptionsLinkSchemaGenerator object>

get_description() → str
override this for custom behaviour

get_operation(*path*, *path_regex*, *path_prefix*, *method*, *registry: drf_spectacular.plumbing.ComponentRegistry*)

get_operation_id()
override this for custom behaviour

get_response_serializers() → Union[rest_framework.serializers.BaseSerializer, Type[rest_framework.serializers.BaseSerializer]]
overrides this for custom behaviour

6.7 audoma.django.db.fields

Audoma Django model Fields This module contains all the fields from Django models with additional functionality. By inheriting from Audoma Mixins, an example is generated for each field (i.e. FloatField will have example generated based on field's min and max values). We can define custom example by simply passing *example* as an argument to the field.

class audoma.django.db.fields.**AutoField**(**args*, ***kwargs*)

class audoma.django.db.fields.**BigAutoField**(**args*, ***kwargs*)

class audoma.django.db.fields.**BigIntegerField**(**args*, ***kwargs*)

class audoma.django.db.fields.**BinaryField**(**args*, ***kwargs*)

class audoma.django.db.fields.**BooleanField**(**args*, ***kwargs*)

class audoma.django.db.fields.**CharField**(**args*, ***kwargs*)

class audoma.django.db.fields.**CommaSeparatedIntegerField**(**args*, ***kwargs*)

class audoma.django.db.fields.**CurrencyField**(**args*, ***kwargs*)

class audoma.django.db.fields.**DateField**(**args*, ***kwargs*)

class audoma.django.db.fields.**DateTimeField**(**args*, ***kwargs*)

class audoma.django.db.fields.**DecimalField**(**args*, ***kwargs*)

class audoma.django.db.fields.**DurationField**(**args*, ***kwargs*)

class audoma.django.db.fields.**EmailField**(**args*, ***kwargs*)

class audoma.django.db.fields.**Field**(**args*, ***kwargs*)

class audoma.django.db.fields.**FilePathField**(**args*, ***kwargs*)

class audoma.django.db.fields.**FloatField**(**args*, ***kwargs*)

class audoma.django.db.fields.**GenericIPAddressField**(**args*, ***kwargs*)

class audoma.django.db.fields.**IPAddressField**(**args*, ***kwargs*)

class audoma.django.db.fields.**IntegerField**(**args*, ***kwargs*)

class audoma.django.db.fields.**JSONField**(**args*, ***kwargs*)

class audoma.django.db.fields.**MACAddressField**(**args*, ***kwargs*)

class audoma.django.db.fields.**MoneyField**(**args*, ***kwargs*)

add_currency_field(*cls*, *name*)
Adds CurrencyField instance to a model class and creates example in documentation.

formfield(***kwargs*)
Return a django.forms.Field instance for this field.

class audoma.django.db.fields.**NullBooleanField**(**args*, ***kwargs*)

class audoma.django.db.fields.**PhoneNumberField**(**args*, *region=None*, ***kwargs*)

class audoma.django.db.fields.**PositiveBigIntegerField**(**args*, ***kwargs*)

class audoma.django.db.fields.**PositiveIntegerField**(**args*, ***kwargs*)

class audoma.django.db.fields.**PositiveSmallIntegerField**(**args*, ***kwargs*)

class audoma.django.db.fields.**SlugField**(**args*, ***kwargs*)

class audoma.django.db.fields.**SmallAutoField**(**args*, ***kwargs*)

class audoma.django.db.fields.**SmallIntegerField**(**args*, ***kwargs*)

class audoma.django.db.fields.**TextField**(**args*, ***kwargs*)

class audoma.django.db.fields.**TimeField**(**args*, ***kwargs*)

class audoma.django.db.fields.**URLField**(**args*, ***kwargs*)

class audoma.django.db.fields.**UUIDField**(**args*, ***kwargs*)

6.8 audoma.django.forms.fields

class audoma.django.forms.fields.**MoneyField**(***, *max_value=None*, *min_value=None*, *max_digits=None*, *decimal_places=None*, ***kwargs*)

prepare_value(*value*) → decimal.Decimal

6.9 audoma.drf.decorators

audoma.drf.decorators.**document_and_format**(*serializer_or_field: Any*) → Callable

6.10 audoma.drf.fields

This module is an override for default drf's field module. Most of those fields, are providing additional example functionality, and also has defined schema type.

class audoma.drf.fields.**BooleanField**(**args*, ***kwargs*)

class audoma.drf.fields.**CharField**(**args*, ***kwargs*)

class audoma.drf.fields.**ChoiceField**(**args*, ***kwargs*)

class audoma.drf.fields.**DateField**(**args*, ***kwargs*)

class audoma.drf.fields.**DateTimeField**(**args*, ***kwargs*)

class audoma.drf.fields.**DecimalField**(**args*, ***kwargs*)

class audoma.drf.fields.**DictField**(**args*, ***kwargs*)

class audoma.drf.fields.**DurationField**(**args*, ***kwargs*)

class audoma.drf.fields.**EmailField**(**args*, ***kwargs*)

class audoma.drf.fields.**FileField**(**args*, ***kwargs*)

class audoma.drf.fields.**FilePathField**(**args*, ***kwargs*)

class audoma.drf.fields.**FloatField**(**args*, ***kwargs*)

class audoma.drf.fields.**HStoreField**(**args*, ***kwargs*)

class audoma.drf.fields.**IPAddressField**(**args*, ***kwargs*)

class audoma.drf.fields.**ImageField**(**args*, ***kwargs*)

class audoma.drf.fields.**IntegerField**(**args*, ***kwargs*)

class audoma.drf.fields.**JSONField**(**args*, ***kwargs*)

class audoma.drf.fields.**ListField**(**args*, ***kwargs*)

class audoma.drf.fields.**MACAddressField**(**args*, ***kwargs*)

class audoma.drf.fields.**MoneyField**(**args*, ***kwargs*)

class audoma.drf.fields.**MultipleChoiceField**(**args*, ***kwargs*)

class audoma.drf.fields.**NullBooleanField**(**args*, ***kwargs*)

class audoma.drf.fields.**PhoneNumberField**(**args*, ***kwargs*)

class audoma.drf.fields.**ReadOnlyField**(**args*, ***kwargs*)

class audoma.drf.fields.**RegexField**(**args*, ***kwargs*)

class audoma.drf.fields.**SerializerMethodField**(**args*, ***kwargs*)

class audoma.drf.fields.**SlugField**(**args*, ***kwargs*)

class audoma.drf.fields.**TimeField**(**args*, ***kwargs*)

class audoma.drf.fields.**URLField**(**args*, ***kwargs*)

class audoma.drf.fields.**UUIDField**(**args*, ***kwargs*)

6.11 audoma.drf.filters

class audoma.drf.filters.**DocumentedTypedChoiceFilter**(*full_choices: NamedTuple*, *parameter_name: str*, ***kwargs*)

Extended TypedChoiceFilter to generate documentation automatically

6.12 audoma.drf.generics

class audoma.drf.generics.**GenericAPIView**(***kwargs*)

Extended GenericAPIView known from rest_framework. This class extends *get_serializer* and *get_serializer_class* methods. Also provides *get_result_serializer*, which is a shourtcut for *get_serializer* with proper param.

_check_action_function()

_extract_audoma_action_serializer(*serializer_type: str*, *audoma_args:* audoma.decorators.AudomaArgs, *status_code: Optional[int] = None*)

get_audoma_action_config()

get_audoma_action_serializer_class(*serializer_type: str*, *status_code: Optional[int] = None*)

get_result_serializer(**args*, ***kwargs*) → rest_framework.serializers.BaseSerializer

Shortuct for get_serializer. Simply has serializer_type set to *result*

get_serializer(**args*, ***kwargs*) → rest_framework.serializers.BaseSerializer

Passes additional param to *get_serializer_class*.

kwargs:

serializer_type - defines if serializer is collect or result serializer. result serializer will be used to produce response, collect to process incoming data.

serializer_class - it is possible to pass serializer_class to get_serializer, this will ends with returning passed serializer_class object.

Returns: Object of obtained serializer class.

get_serializer_class(*serializer_type: str = 'collect'*, *many: bool = False*, *status_code: Optional[int] = None*) → Type[rest_framework.serializers.BaseSerializer]

Extends defuault *get_serializer_class* method. This returns proper serializer_class for current request.

Args: serializer_type - serializer_type of serializer to be returned, it may be collect or result serializer.

Returns: This returns serializer_class

6.13 audoma.drf.mixins

This module overwrites basic mixins provided bu django rest framework. Mixins defined here should be used instead of default drf's mixins. Those mixins should be used to allow usage of extended *get_serializer* method>

Example:

from audoma.drf import mixins from audoma.drf import viewsets

class ExampleModelViewSet(mixins.ActionModelMixin, mixins.CreateModelMixin, viewsets.GenericViewSet,

): serializer_class = ExampleModelSerializer queryset = ExampleModel.objects.all()

class audoma.drf.mixins.**ActionModelMixin**

get_success_headers(*data: dict*) → dict

perform_action(*request: rest_framework.request.Request*, *success_status: int = 200*, *instance: Optional[Any] = None*, *partial: bool = False*, ***kwargs*) → rest_framework.response.Response

retrieve_instance(*request: rest_framework.request.Request*, *instance: Optional[Any] = None*, *success_status: int = 200*, ***kwargs*) → rest_framework.response.Response

class audoma.drf.mixins.**BulkCreateModelMixin**

Either create a single or many model instances in bulk by using the Serializers `many=True` ability from Django REST >= 2.2.5. .. note:

```
This mixin uses the same method to create model instances
as ``CreateModelMixin`` because both non-bulk and bulk
requests will use ``POST`` request method.
```

create(*request: rest_framework.request.Request*, **args*, ***kwargs*) → rest_framework.response.Response

perform_bulk_create(*serializer: rest_framework.serializers.BaseSerializer*) → None

class audoma.drf.mixins.**BulkUpdateModelMixin**

Update model instances in bulk by using the Serializers `many=True` ability from Django REST >= 2.2.5.

bulk_update(*request: rest_framework.request.Request*, **args*, ***kwargs*) → rest_framework.response.Response

get_object() → Any

partial_bulk_update(*request: rest_framework.request.Request*, **args*, ***kwargs*) → rest_framework.response.Response

perform_bulk_update(*serializer: rest_framework.serializers.BaseSerializer*) → None

perform_update(*serializer: rest_framework.serializers.BaseSerializer*) → None

class audoma.drf.mixins.**CreateModelMixin**

create(*request: rest_framework.request.Request*, **args*, ***kwargs*) → rest_framework.response.Response

class audoma.drf.mixins.**DestroyModelMixin**

destroy(*request: rest_framework.request.Request*, **args*, ***kwargs*) → rest_framework.response.Response

class audoma.drf.mixins.**ListModelMixin**

get_paginated_response(*data: List[Dict]*) → rest_framework.response.Response

list(*request: rest_framework.request.Request*, **args*, ***kwargs*) → rest_framework.response.Response

class audoma.drf.mixins.**RetrieveModelMixin**

retrieve(*request: rest_framework.request.Request*, **args*, ***kwargs*) → rest_framework.response.Response

class audoma.drf.mixins.**UpdateModelMixin**

update(*request: rest_framework.request.Request*, **args*, ***kwargs*) → rest_framework.response.Response

6.14 audoma.drf.serializers

class audoma.drf.serializers.**BulkListSerializer**(**args*, ***kwargs*)

class audoma.drf.serializers.**DefaultMessageSerializer**(**args*, ***kwargs*)

class audoma.drf.serializers.**ListSerializer**(**args*, ***kwargs*)

class audoma.drf.serializers.**ModelSerializer**(**args*, ***kwargs*)

Extends default ModelSerializer, modifies serializer_field_mapping (replaces some fields with audoma fields). Adds support for generating audoma example for field.

build_standard_field(*field_name*, *model_field*) → Tuple[Union[Type[rest_framework.fields.Field], dict]]

Adds support for mapping example from model fields to model serializer fields.

serializer_choice_field

alias of *audoma.drf.fields.ChoiceField*

class audoma.drf.serializers.**ResultSerializerClassMixin**

Allows to define wrap for serializer result.

class audoma.drf.serializers.**Serializer**(**args*, ***kwargs*)

audoma.drf.serializers.**result_serializer_class**(*SerializerClass: Type[rest_framework.serializers.BaseSerializer]*) → Type[rest_framework.serializers.BaseSerializer]

Helper function which wraps the serializer result if necessary.

Args:

- SerializerClass - serializer class which result should be wrapped

Returns: ResultSerializer class

6.15 audoma.drf.validators

class audoma.drf.validators.**ExclusiveFieldsValidator**(*fields: Union[List[str], Tuple[str]]*, *message: Optional[str] = None*, *required: bool = True*, *message_required: Optional[str] = None*)

This is extra validator defined in audoma. This validator allows to define mutually exclusvie fields.

Attributes: fields - list or a tuple of mutually exclusive field names message - string validation error message required - boolean value, determines if fields are required message_reqiured - string message if one of fields is required and none has been passed

Args: fields - list or a tuple of mutually exclusive field names message - string validation error message required - boolean value, determines if fields are required message_reqiured - string message if one of fields is required and none has been passed

message = 'The fields {field_names} are mutually exclusive arguments.'

message_required = 'One of the fields {field_names} is required.'

6.16 audoma.drf.viewsets

class audoma.drf.viewsets.**AudomaPagination**

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

Note: If this won't be used it'll cause less explicit pagination documentation. This class does not provide any additional functionality.

Args: page_size (int) - number of items per page - by default this is set to 25 max_page_size (int) - maximum number of items per page - by default this is set to 2000

get_paginated_response_schema(*schema: List[dict]*) → dict

Simple method to add pagination information to the schema.

Args: schema (List[dict]) - list of schema items

Returns: Dictionary with pagination information including examples

max_page_size = 2000

page_size = 25

class audoma.drf.viewsets.**GenericViewSet**(***kwargs*)

_parse_response_data(*response_data: List[Any]*) → List[str]

handle_exception(*exc: Exception*) → rest_framework.response.Response

Handle any exception that occurs, by returning an appropriate response, or re-raising the error.

pagination_class

alias of *audoma.drf.viewsets.AudomaPagination*

CHAPTER

SEVEN

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

a

INDEX

Symbols

A

B

C

D

E

F

G

R

S

T

U