Developer documentation

Release 5.9.0

VST Consulting

Feb 08, 2024

Contents

VST Utils is a small framework for quick generation of a single-page applications. The main feature of the VST Utils framework is autogenerated GUI, which is formed based on the OpenAPI schema. OpenAPI schema is a JSON, that contents description of the models used in the REST API and info about all paths of the application.

In the documentation you can find info about QuickStart of new project based on VST Utils, description of base models, views and fields available in the framework, and also you will know how you can redefine some standard models, views and fields in your project.

1 Quick Start

Starting of new project, based on VST Utils Framework, is rather simple. We recommend to create a virtual environment for each project to avoid conflicts in the system.

Let's learn by example. All you need to do is run several commands. This manual consist of two parts:

1. Description of the process of creating a new application and the main commands for launching and deploying.
2. Description of the process of creating new entities in the application.

1.1 New application creation

Throughout this tutorial, we'll go through a creation of a basic poll application.

1. **Install VST Utils**

   ```
   pip install vstutils
   ```

   In this case, we install a package with the minimum required set of dependencies for creating new projects. However, inside the project, the extra argument *prod* is used, which additionally installs the packages necessary for working in the production environment. There is also a set of test dependencies, which contains everything you need for testing and analyzing code coverage.

   It is also worth noting extra dependencies as:

   - **rpc** - install dependencies for asynchronous tasks working
   - **ldap** - a set of dependencies for ldap authorization support
   - **doc** - everything which needed to build documentation and to implement the delivery of documentation inside a running server
   - **pil** - library for correct work of image validators
   - **boto3** - an additional set of packages for working with S3 storage outside of AWS
   - **sqs** - a set of dependencies for connecting asynchronous tasks to SQS queues (can be used instead of the **rpc**).

   You can combine multiple dependencies at the same time to assemble your set of functionality into a project. For example, to work an application with asynchronous tasks and media storage in MinIO, you will need the following command:

```
pip install vstutils[prod,rpc,boto3]
```

To install the most complete set of dependencies, you can use the common parameter **all**.

```
pip install vstutils[all]
```

2. **Create new project, based on VST Utils**

   If this is your first time using vstutils, you'll have to take care of some initial setup. Namely, you'll need to auto-generate some code that establishes a vstutils application – a collection of settings for an instance of vstutils, including database configuration, Django-specific and vstutils-specific options and application-specific settings. To create new project execute following command:

```
python -m vstutils newproject --name {{app_name}}
```

   This command will offer you to specify such options of new app, as:

   - **project name** - name of your new application;
   - **project guiname** - name of your new application, that will be used in GUI (web-interface);
   - **project directory** - path to directory, where project will be created.

   Or you can execute following command, that includes all needed data for new project creation.

```
python -m vstutils newproject --name {{app_name}} --dir {{app_dir}} --
↪guiname {{app_guiname}} --noinput
```

   This command creates new project without confirming any data.

   These commands create several files in `project directory`.

```
/{{app_dir}}/{{app_name}}
 ├── frontend_src
 │   ├── app
 │   │   └── index
 │   ├── .editorconfig
 │   ├── .eslintrc.js
 │   └── .prettierrc
 ├── MANIFEST.in
 ├── package.json
 ├── README.rst
 ├── requirements-test.txt
 ├── requirements.txt
 ├── pyproject.toml
 ├── setup.py
 ├── {{app_name}}
 │   ├── __init__.py
 │   ├── __main__.py
 │   ├── models
 │   │   └── __init__.py
 │   ├── settings.ini
 │   ├── settings.py
 │   ├── web.ini
 │   └── wsgi.py
 ├── test.py
 ├── tox.ini
 └── webpack.config.jsdefault
```

   where:

- **frontend_src** - directory that contains all sources for frontend;
- **MANIFEST.in** - this file is used for building installation package;
- **{{app_name}}** - directory with files of your application;
- **package.json** - this file contains list of frontend dependencies and commands to build;
- **README.rst** - default README file for your application (this file includes base commands for starting/stopping your application);
- **requirements-test.txt** - file with list of requirements for test environment;
- **requirements.txt** - file with list of requirements for your application;
- **pyproject.toml** - this file is used for building installation package;
- **setup.py** - this file is used for building installation package;
- **test.py** - this file is used for tests creation;
- **tox.ini** - this file is used for tests execution;
- **webpack.config.js.default** - this file contain minimal script for webpack (replace '.default' if write smthg in 'app.js').

You should execute below commands from the `/{{app_dir}}/{{app_name}}/` directory. It is good practice to use tox (should be installed before use) to create a debugging environment for your application. For these purposes, it is recommended to use `tox -e contrib` in the project directory, which will automatically create a new environment with the required dependencies.

3. **Apply migrations**

   Let's verify a newly created vstutils project does work. Change into the outer `/{{app_dir}}/{{app_name}}` directory, if you haven't already, and run the following command:

   ```
   python -m {{app_name}} migrate
   ```

   This command create SQLite (by default) database with default SQL-schema. VSTUTILS supports all databases Django does[1].

4. **Create superuser**

   ```
   python -m {{app_name}} createsuperuser
   ```

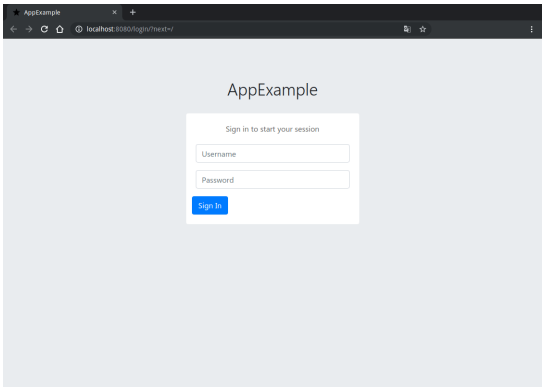
5. **Start your application**

   ```
   python -m {{app_name}} web
   ```

   Web-interface of your application has been started on the port 8080. You've started the vstutils production server based on uWSGI[2].

   Warning: Now's a good time to note: if you want to run the web-server with a debugger, then you should run the standard Django's dev-server[3].

[1] https://docs.djangoproject.com/en/4.1/ref/databases/#databases
[2] https://uwsgi-docs.readthedocs.io/
[3] https://docs.djangoproject.com/en/4.1/intro/tutorial01/#the-development-server



If you need to stop the server, use following command:

```
python -m {{app_name}} web stop=/tmp/{{app_name}}_web.pid
```

You've created the simplest application, based on VST Utils framework. This application only contains User Model. If you want to create your own models look at the section below.

1.2 Adding new models to application

If you want to add some new entities to your application, you need to do following on the back-end:

1. Create Model;
2. Create Serializer (optional);
3. Create View (optional);
4. Add created Model or View to the API;
5. Make migrations;
6. Apply migrations;
7. Restart your application.

Let's look how you can do it on the AppExample - application, that has 2 custom models:

- Task (abstraction for some tasks/activities, that user should do);
- Stage (abstraction for some stages, that user should do to complete the task. This model is nested into the Task Model).

1.2.1 Models creation

Firstly, you need to create file `{{model_name}}.py` in the `/{{app_dir}}/{{app_name}}/{{app_name}}/models` directory.

Let make out an example from **BModel**:

class vstutils.models.**BModel**(**args*, ***kwargs*)

Default model class that generates model viewset, separate serializers for list() and retrieve(), filters, api endpoints and nested views.

Examples:

```
from django.db import models
from rest_framework.fields import ChoiceField
from vstutils.models import BModel

class Stage(BModel):
    name = models.CharField(max_length=256)
    order = models.IntegerField(default=0)

    class Meta:
        default_related_name = "stage"
        ordering = ('order', 'id',)
        # fields which would be showed on list.
        _list_fields = [
            'id',
            'name',
        ]
        # fields which would be showed on detail view and creation.
        _detail_fields = [
            'id',
            'name',
            'order'
        ]
        # make order as choices from 0 to 9
        _override_detail_fields = {
            'order': ChoiceField((str(i) for i in range(10)))
        }


class Task(BModel):
    name = models.CharField(max_length=256)
    stages = models.ManyToManyField(Stage)
    _translate_model = 'Task'

    class Meta:
        # fields which would be showed.
        _list_fields = [
            'id',
            'name',
        ]
        # create nested views from models
        _nested = {
            'stage': {
                'allow_append': False,
                'model': Stage
            }
        }
```

In this case, you create models which could converted to simple view, where:

- `POST`/`GET` to `/api/version/task/` - creates new or get list of tasks
- `PUT`/`PATCH`/`GET`/`DELETE` to `/api/version/task/:id/` - updates, retrieves or removes instance of task
- `POST`/`GET` to `/api/version/task/:id/stage/` - creates new or get list of stages in task
- `PUT`/`PATCH`/`GET`/`DELETE` to `/api/version/task/:id/stage/:stage_id` - updates, retrieves or removes instance of stage in task.

To attach a view to an API insert the following code in *settings.py*:

```
API[VST_API_VERSION][r'task'] = {
    'model': 'your_application.models.Task'
}
```

For primary access to generated view inherit from *Task.generated_view* property.

To make translation on frontend easier use `_translate_model` attribute with model_name.

List of meta-attributes for generating a view:

- `_view_class` - list of additional view classes to inherit from, class or string to import with base class ViewSet. Constants are also supported:
  - `read_only` - to create a view only for viewing;
  - `list_only` - to create a view with list only;
  - `history` - to create a view only for viewing and deleting records.

  CRUD-view is applied by default.
- `_serializer_class` - class of API serializer; use this attribute to specify parent class for auto-generated serializers. Default is *vstutils.api.serializers.VSTSerializer*. Can take a string to import, serializer class or `django.utils.functional.SimpleLazyObject`.
- `_serializer_class_name` - model name for OpenAPI definitions.This would be a model name in generated admin interface. Default is name of model class.
- `_list_fields` or `_detail_fields` - list of fields which will be listed in entity list or detail view accordingly. Same as DRF serializers meta-attribute "fields".
- `_override_list_fields` or `_override_detail_fields` - mapping with names and field types that will be redeclared in serializer attributes(think of it as declaring fields in DRF ModelSerializer).
- `_properties_groups` - dict with key as group name and value as list of fields(str). Allows to group fields in sections on frontend.
- `_view_field_name` - name of field frontend shows as main view name.
- `_non_bulk_methods` - list of methods which must not used via bulk requests.
- `_extra_serializer_classes` - mapping with additional serializers in viewset. For example, custom serializer, which will compute smth in action (mapping name). Value can be string for import. Important note: setting *model* attribute to None allows to use standard serializer generation mechanism and get fields from a list or detail serializer (set *__inject_from__* serializer's meta attribute to *list* or *detail* accordingly). In some cases, it is required to pass the model to the serializer. For these purposes, the constant *LAZY_MODEL* can be used as a meta attribute. Each time the serializer is used, the exact model where this serializer was declared will be set.
- `_filterset_fields` - list/dict of filterset names for API-filtering. Default is list of fields in list view. During processing a list of fields checks for the presence of special field names and inherit additional parent classes. If the list contains `id`, class will inherit from *vstutils.api.filters.DefaultIDFilter*. If the list contains `name`, class will inherit from *vstutils.api.filters.DefaultNameFilter*. If both conditions are present, inheritance will be from all of the above classes. Possible values include *list* of fields to filter or *dict* where key is a field name and value is a Filter class. Dict extends attribute functionality and provides ability to override filter field class (None value disables overriding).
- `_search_fields` - tuple or list of fields using for search requests. By default (or *None*) get all filterable fields in detail view.
- `_copy_attrs` - list of model-instance attributes indicates that object is copyable with this attrs.

- `_nested` - key-value mapping with nested views (key - nested name, kwargs for *vstutils.api.decorators.nested_view* decorator but supports `model` attribute as nested). `model` can be string for import. Use `override_params` when you need to override generated view parameters for nested view (works only with `model` as view).
- `_extra_view_attributes` - key-value mapping with additional view attributes, but has less priority over generated attributes.

In common, you can also add custom attributes to override or extend the default list of processing classes. Supported view attributes are `filter_backends`, `permission_classes`, `authentication_classes`, `throttle_classes`, `renderer_classes` and `parser_classes`. List of meta-attributes for settings of view is looks like:

- `_pre_{attribute}` - List of classes included before defaults.
- `_{attribute}` - List of classes included after defaults.
- `_override_{attribute}` - boolean flag indicates that attribute override default viewset (otherwise appends). Default is `False`.

Note: You may need to create an action[4] on generated view. Use *vstutils.models.decorators.register_view_action* decorator with the `detail` argument to determine applicability to a list or detail entry. In this case, the decorated method will take an instance of the view object as `self` attribute.

Note: In some cases, inheriting models may require to inherit Meta class from the base model. If the Meta is explicitly declared in the base class, then you can get it through the attribute *OriginalMeta* and use it for inheritance.

Note: Docstring of model will be reused for view descriptions. It is possible to write both a general description for all actions and description for each action using the following syntax:

```
General description for all actions.

action_name:
    Description for this action.

another_action:
    Description for another action.
```

The `get_view_class()` method is a utility method in the Django ORM model designed to facilitate the configuration and instantiation of Django Rest Framework (DRF) Generic ViewSets. It allows developers to define and customize various aspects of the associated DRF view class.

Examples:

```
# Create simple list view with same fields
TaskViewSet = Task.get_view_class(view_class='list_only')

# Create view with overriding nested view params
from rest_framework.mixins import CreateModelMixin

TaskViewSet = Task.get_view_class(
    nested={
```

(continues on next page)

(continued from previous page)

```
        "milestones": {
            "model": Stage,
            "override_params": {
                "view_class": ("history", CreateModelMixin)
            },
        },
    },
)
```

Developers can use this method to customize various aspects of the associated view, such as serializer classes, field configurations, filter backends, permission classes, etc. It uses attributes declared in meta attributes, but allows individual parts to be overriden.

More information about Models you can find in Django Models documentation[5].

If you don't need to create custom *serializers* or *view sets*, you can go to this *stage*.

1.2.2 Serializers creation

Note - If you don't need custom serializer you can skip this section

Firstly, you need to create file `serializers.py` in the `/{{app_dir}}/{{app_name}}/{{app_name}}/` directory.

Then you need to add some code like this to `serializers.py`:

```
from datetime import datetime
from vstutils.api import serializers as vst_serializers
from . import models as models

class StageSerializer(models.Stage.generated_view.serializer_class):

    class Meta:
        model = models.Stage
        fields = ('id',
                  'name',
                  'order',)

    def update(self, instance, validated_data):
        # Put custom logic to serializer update
        instance.last_update = datetime.utcnow()
        super().update(instance, validated_data)
```

More information about Serializers you can find in Django REST Framework documentation for Serializers[6].

[4] https://www.django-rest-framework.org/api-guide/viewsets/#marking-extra-actions-for-routing
[5] https://docs.djangoproject.com/en/4.1/topics/db/models/
[6] https://www.django-rest-framework.org/api-guide/serializers/#modelserializer

1.2.3 Views creation

Note - If you don't need custom view set you can skip this section

Firstly, you need to create file `views.py` in the `/{{app_dir}}/{{app_name}}/{{app_name}}/` directory.

Then you need to add some code like this to `views.py`:

```
from vstutils.api import decorators as deco
from vstutils.api.base import ModelViewSet
from . import serializers as sers
from .models import Stage, Task


class StageViewSet(Stage.generated_view):
    serializer_class_one = sers.StageSerializer


'''
Decorator, that allows to put one view into another
    * 'tasks' - suburl for nested view
    * 'methods=["get"]' - allowed methods for this view
    * 'manager_name='hosts' - Name of related QuerySet to the child model instances␣
↪(we set it in HostGroup model as "hosts = models.ManyToManyField(Host)")
    *  'view=Task.generated_view' - Nested view, that will be child view for␣
↪decorated view
'''
@nested_view('stage', view=StageViewSet)
class TaskViewSet(Task.generated_view):
    '''
    Task operations.
    '''
```

More information about Views and ViewSets you can find in Django REST Framework documentation for views[7].

1.2.4 Adding Models to API

To add created Models to the API you need to write something like this at the end of your `settings.py` file:

```
'''
Some code generated by VST Utils
'''

'''
Add Task view set to the API
Only 'root' (parent) views should be added there.
Nested views added automatically, that's why there is only Task view.
Stage view is added altogether with Task as nested view.
'''
API[VST_API_VERSION][r'task'] = {
    'view': 'newapp2.views.TaskViewSet'
}

'''
You can add model too.
```

(continues on next page)

[7] https://www.django-rest-framework.org/api-guide/viewsets/

(continued from previous page)

```
All model generate base ViewSet with data that they have, if you don't create custom␣
↪ViewSet or Serializer
'''
API[VST_API_VERSION][r'task'] = dict(
    model='newapp2.models.Task'
)

# Adds link to the task view to the GUI menu
PROJECT_GUI_MENU.insert(0, {
    'name': 'Task',
     # CSS class of font-awesome icon
    'span_class': 'fa fa-list-alt',
    'url': '/task'
})
```

1.2.5 Migrations creation

To make migrations open `/{{app_dir}}/{{app_name}}/` directory and execute following command:

```
python -m {{app_name}} makemigrations {{app_name}}
```

More information about Migrations you can find in Django Migrations documentation[8].

1.2.6 Migrations applying

To apply migrations you need to open `/{{app_dir}}/{{app_name}}/` directory and execute following command:

```
python -m {{app_name}} migrate
```

1.2.7 Restart of Application

To restart your application, firstly, you need to stop it (if it was started before):

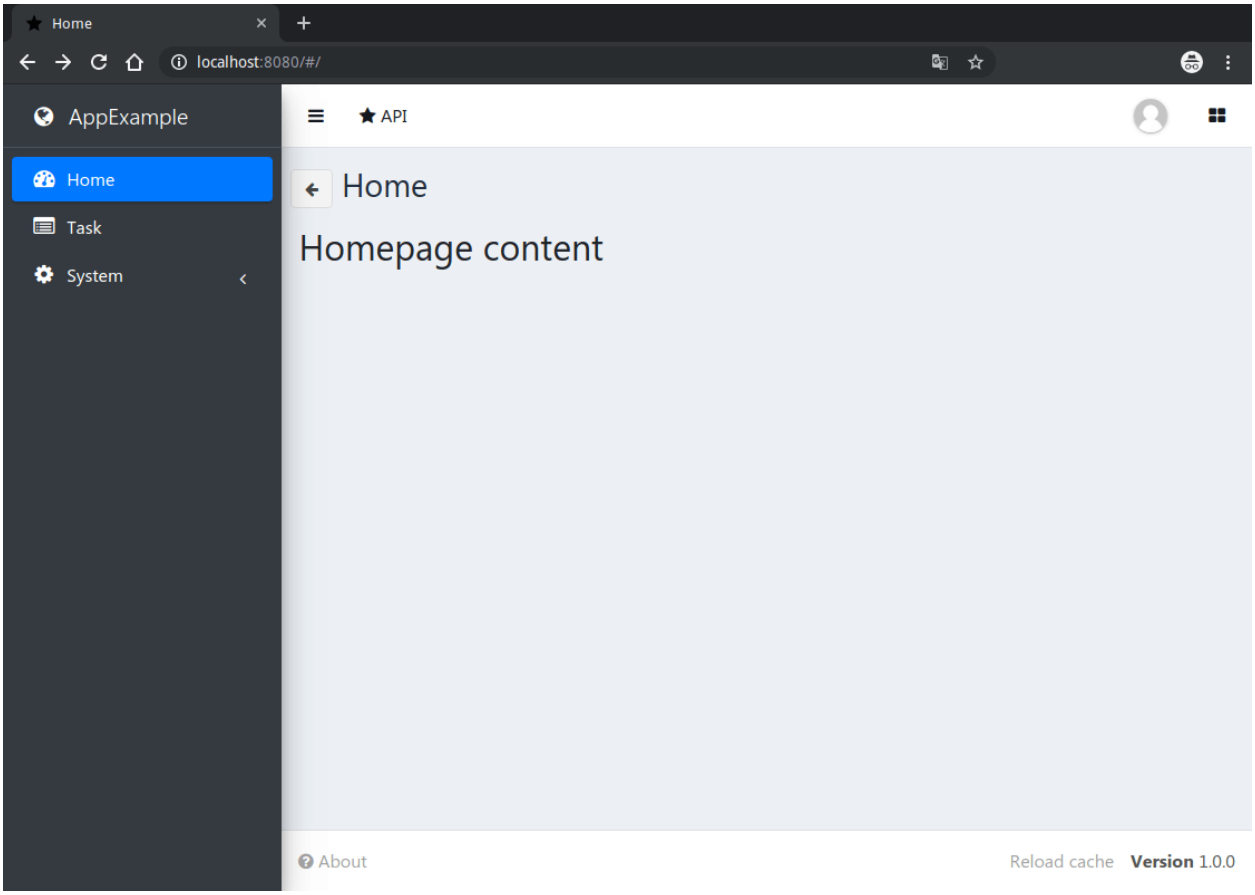
```
python -m {{app_name}} web stop=/tmp/{{app_name}}_web.pid
```

And then start it again:

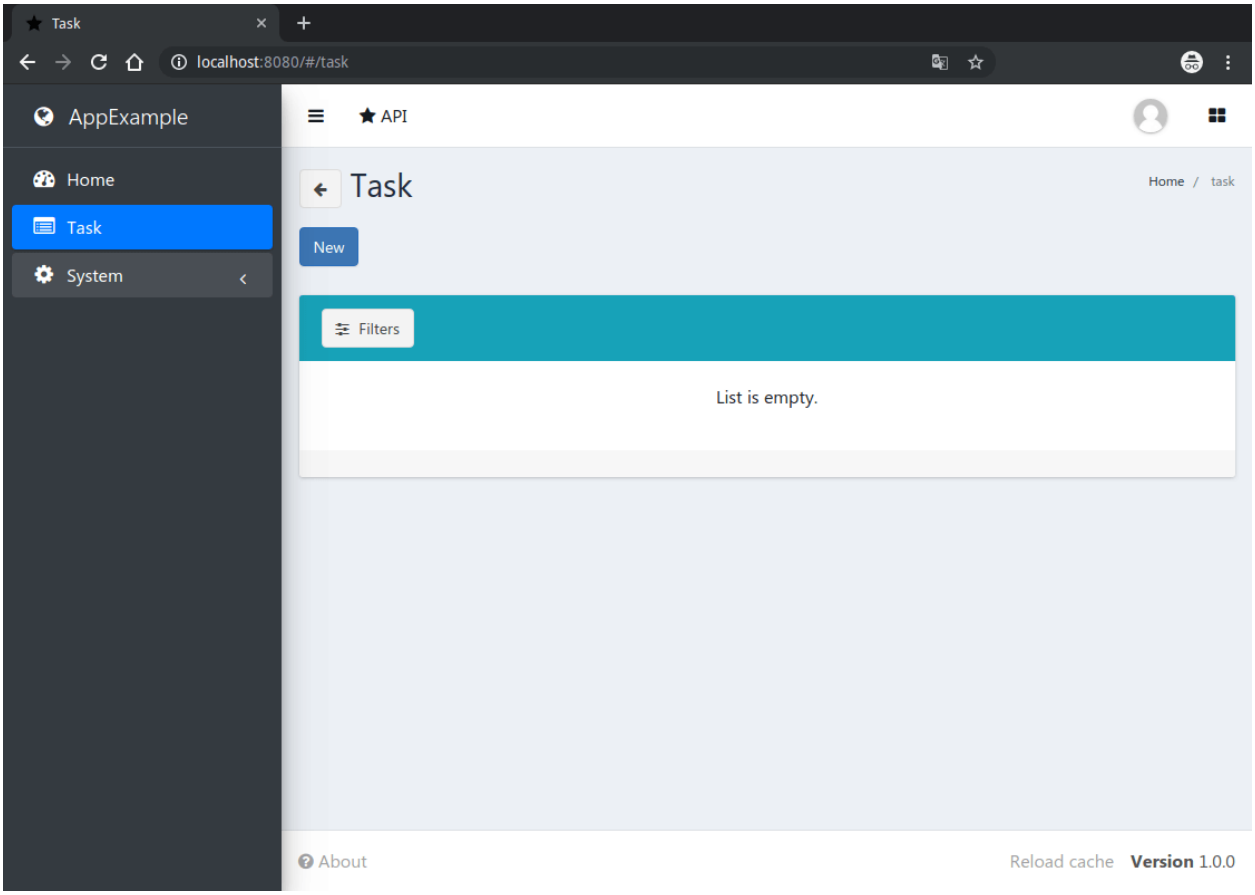
```
python -m {{app_name}} web
```

After cache reloading you will see following page:

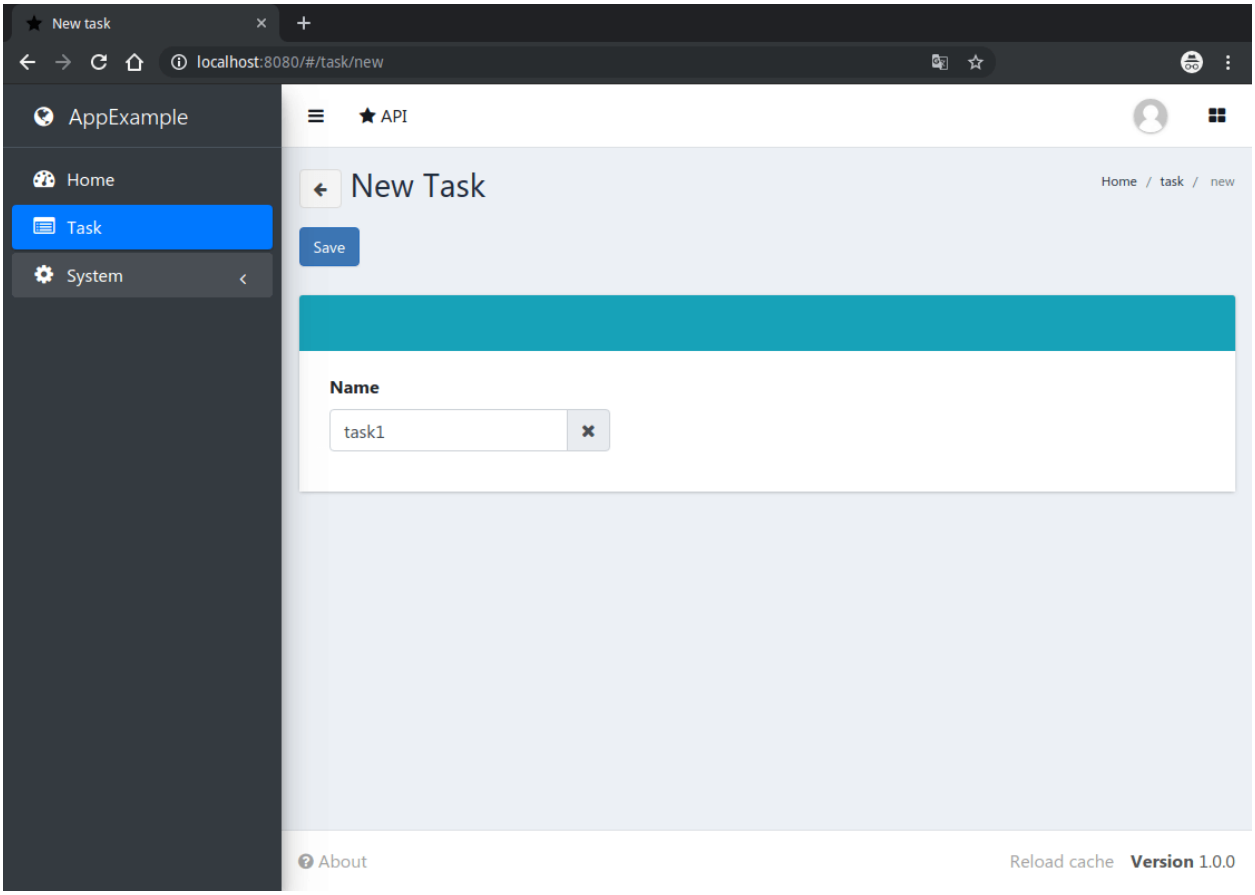
[8] https://docs.djangoproject.com/en/4.1/topics/migrations/



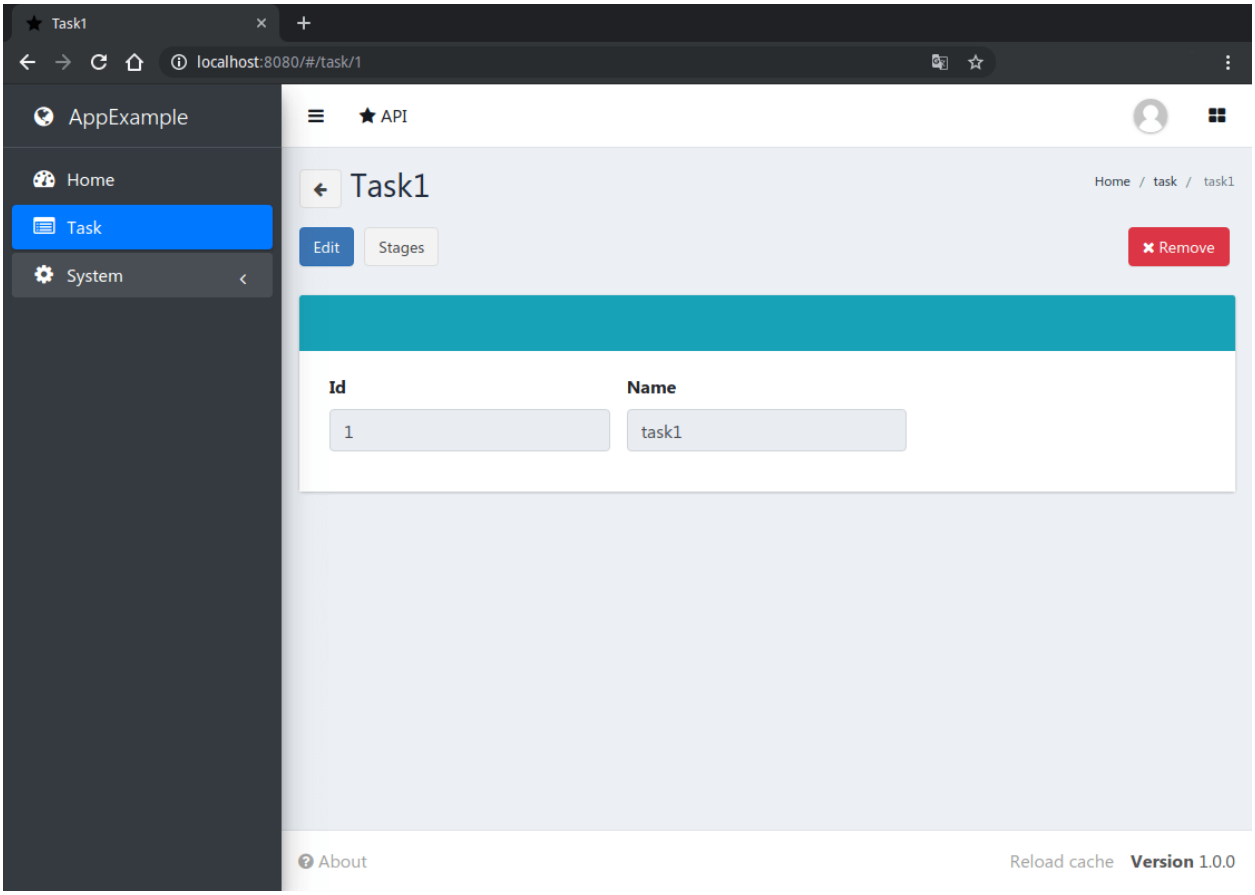
As you can see, link to new Task View has been added to the sidebar menu. Let's click on it.



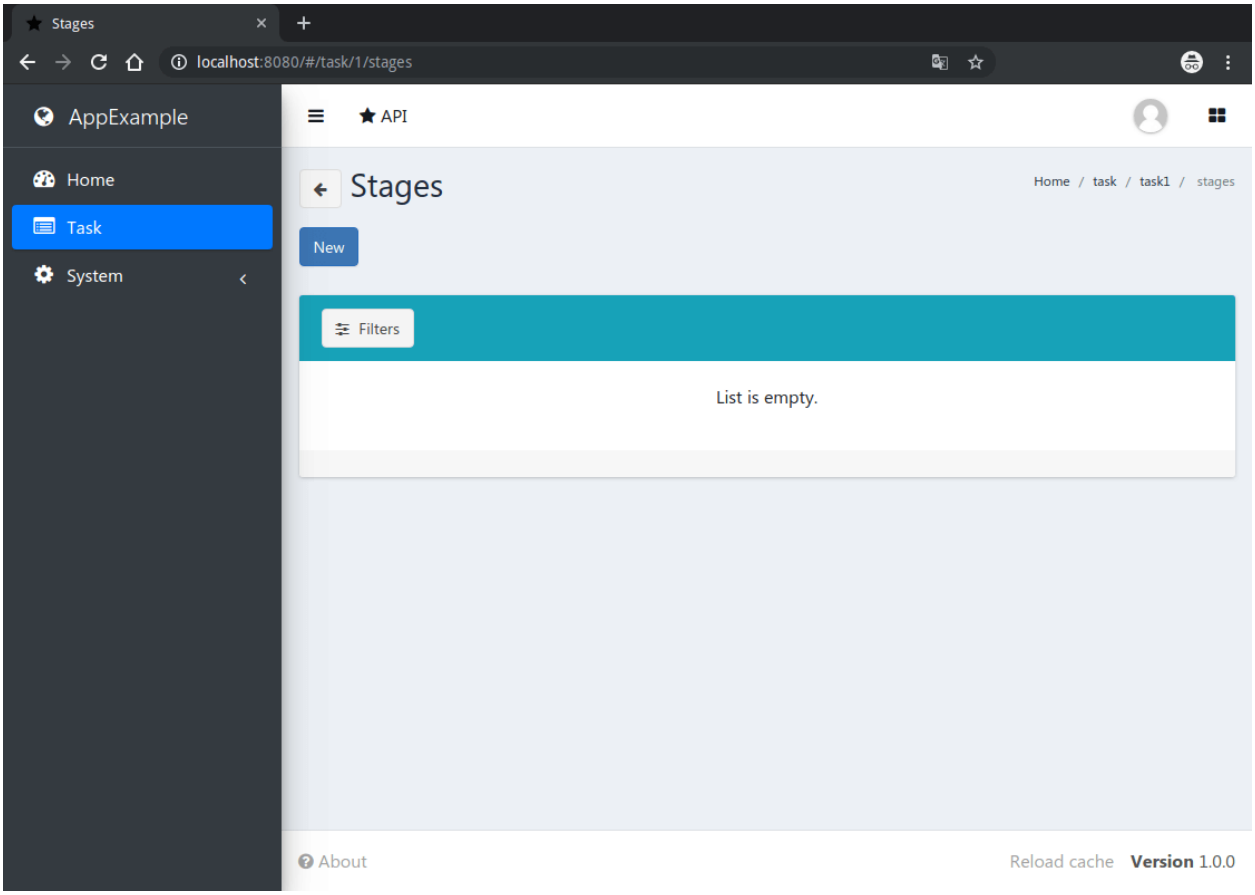
There is no task instance in your app. Add it using 'new' button.



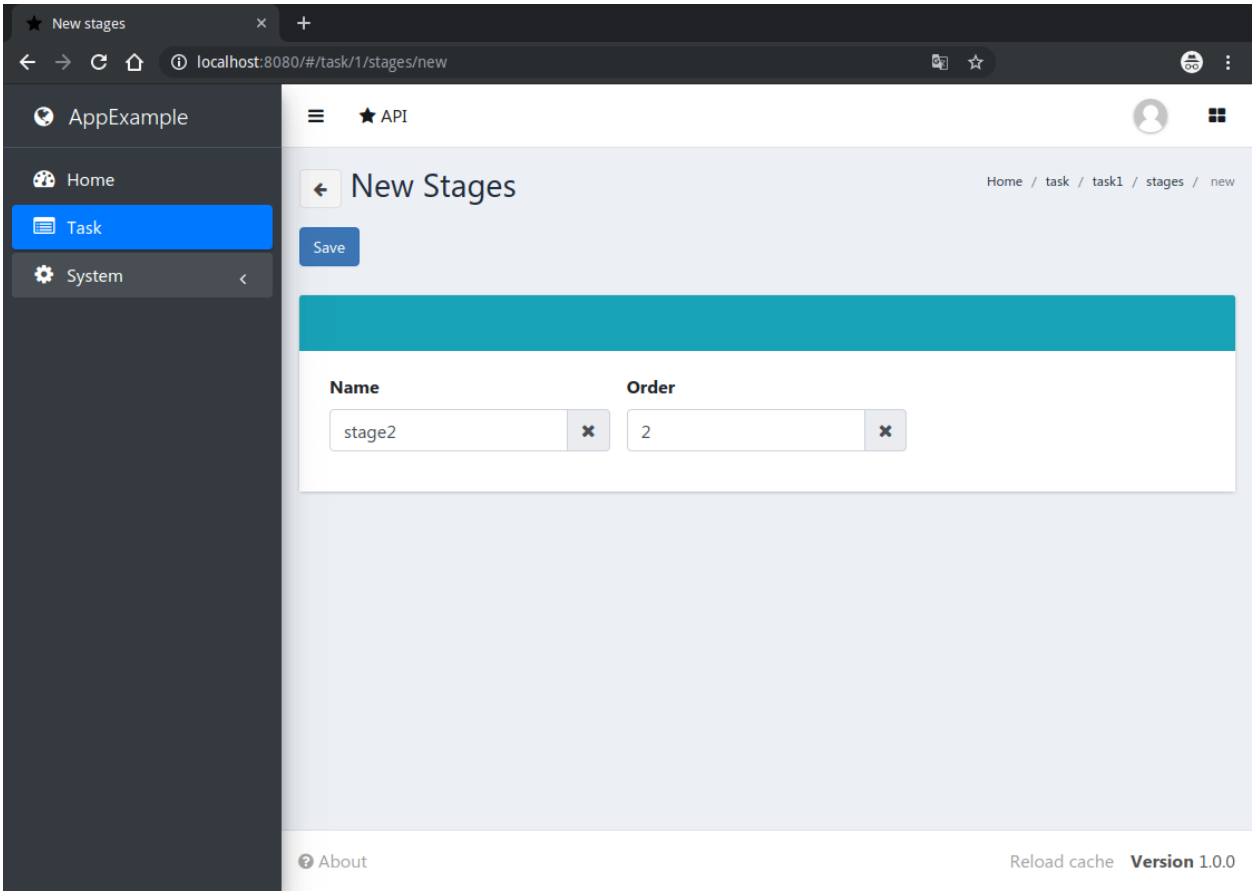
After creating a new task you'll see a following page:

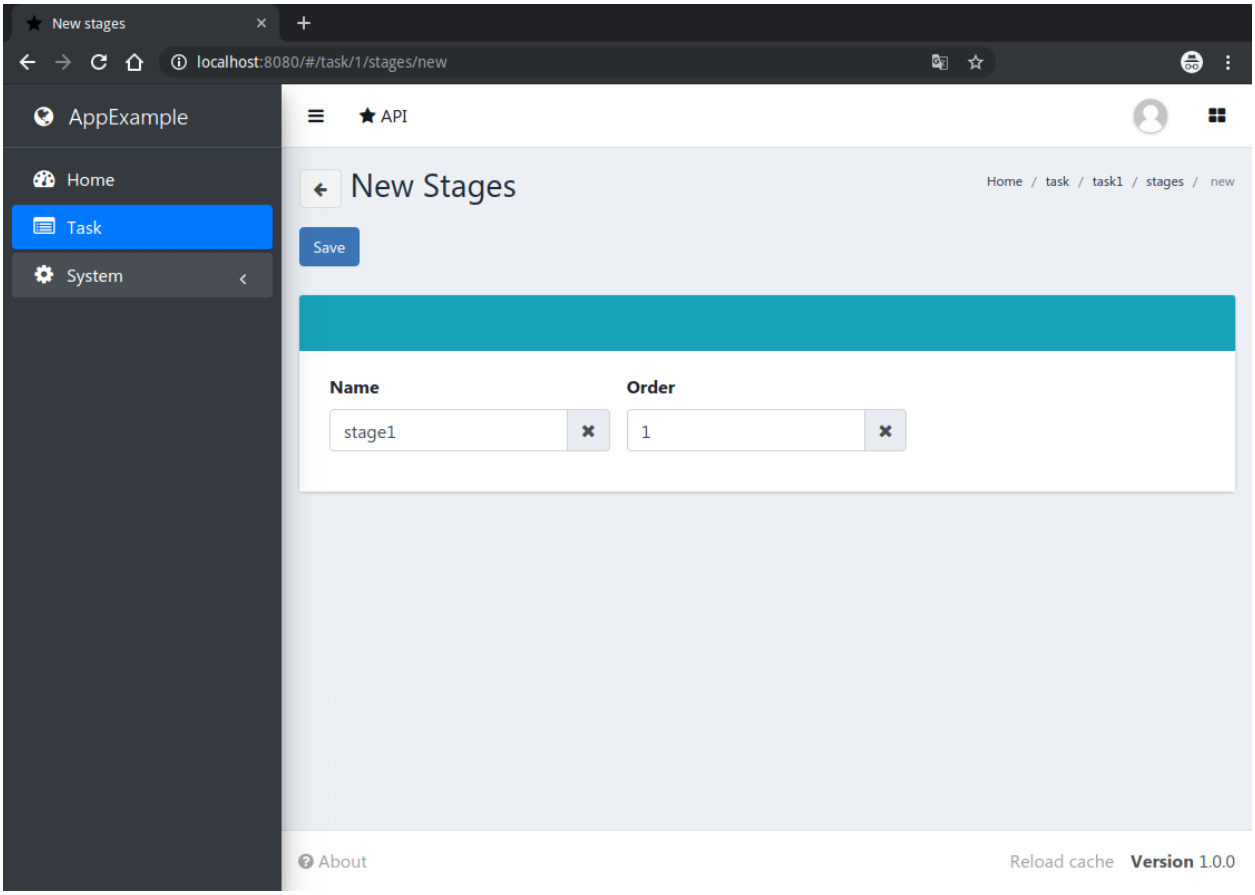


As you can see, there is 'stages' button, that opens page with this task's stages list. Let's click on it.

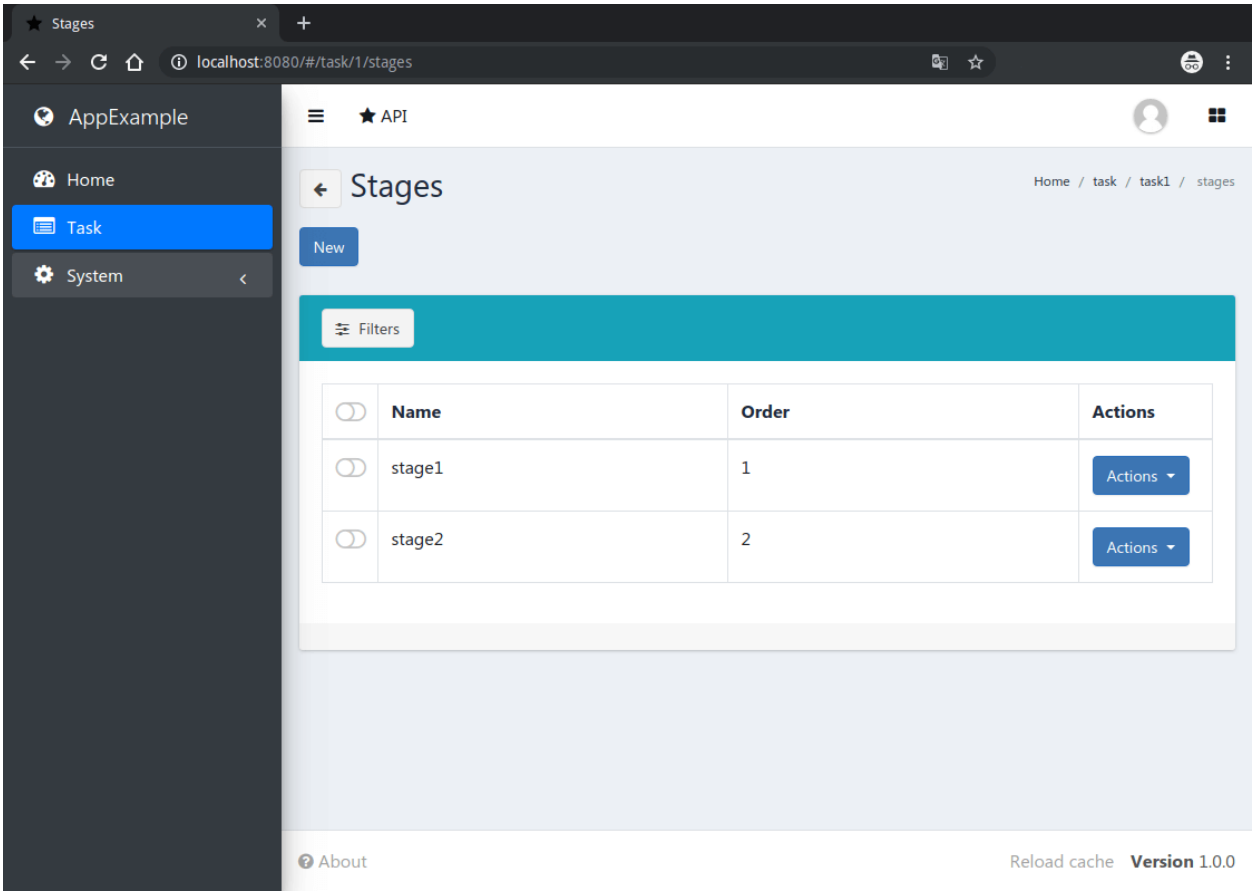


There is no stage instances in your app. Let's create 2 new stages.


New stages
localhost:8080/#/task/1/stages/new
AppExample
Home
Task
System
API
New Stages
Home / task / task1 / stages / new
Save
Name
stage2
Order
2
About
Reload cache
Version 1.0.0



After stages creation page with stages list will looks like this:



Sorting by 'order' field works, as we mentioned in the our `models.py` file for Stage Model.

Additional information about Django and Django REST Framework you can find in Django documentation[9] and Django REST Framework documentation[10].

[9] https://docs.djangoproject.com/en/4.1/
[10] https://www.django-rest-framework.org/

2 Configuration manual

2.1 Introduction

Though default configuration is suitable for many common cases, vstutils-based applications is highly configurable system. For advanced settings (scalability, dedicated DB, custom cache, logging or directories) you can configure vstutils-based application deeply by tweaking `/etc/{{app_name or app_lib_name}}/settings.ini`.

The most important thing to keep in mind when planning your application architecture is that vstutils-based applications have a service-oriented structure. To build a distributed scalable system you only need to connect to a shared *database*, shared *cache*, *locks* and a shared *rpc* service (MQ such as RabbitMQ, Redis, Tarantool, etc.). A shared file storage may be required in some cases, but vstutils does not require it.

Let's cover the main sections of the config and its parameters:

2.2 Main settings

Section `[main]`.

This section is intended for settings related to whole vstutils-based application (both worker and web). Here you can specify verbosity level of vstutils-based application during work, which can be useful for troubleshooting (logging level etc). Also there are settings for changing timezone for whole app and allowed domains.

To use LDAP protocol, create following settings in section `[main]`.

```
ldap-server = ldap://server-ip-or-host:port
ldap-default-domain = domain.name
ldap-auth_format = cn=<username>,ou=your-group-name,<domain>
```

ldap-default-domain is an optional argument, that is aimed to make user authorization easier (without input of domain name).

ldap-auth_format is an optional argument, that is aimed to customize LDAP authorization. Default value: cn=<username>,<domain>

In the example above authorization logic will be the following:

1. System checks combination of login:password in database;
2. System checks combination of login:password in LDAP:

  - if domain was mentioned, it will be set during authorization (if user enter login without `user@domain.name` or without `DOMAIN\user`);
  - if authorization was successful and there is user with entered credentials in database, server creates session that user.

- **debug** - Enable debug mode. Default: false.
- **allowed_hosts** - Comma separated list of domains, which allowed to serve. Default: `*`.
- **first_day_of_week** - Integer value with first day of week. Default: `0`.
- **ldap-server** - LDAP server connection. For example: `ldap://your_ldap_server:389`
- **ldap-default-domain** - Default domain for auth.
- **ldap-auth_format** - Default search request format for auth. Default: `cn=<username>,<domain>`.
- **timezone** - Timezone for web-application. Default: UTC.
- **log_level** - Logging level. The verbosity level, configurable in Django and Celery, dictates the extent of log information, with higher levels providing detailed debugging insights for development and lower levels streamlining logs for production environments. Default: WARNING.
- **enable_django_logs** - Enable or disable Django logger to output. Useful for debugging. Default: false.
- **enable_admin_panel** - Enable or disable Django Admin panel. Default: false.
- **enable_registration** - Enable or disable user self-registration. Default: false.
- **enable_user_self_remove** - Enable or disable user self-removing. Default: false.
- **auth-plugins** - Comma separated list of django authentication backends. Authorization attempt is made until the first successful one in order specified in the list.
- **auth-cache-user** - Enable or disable user instance caching. It increases session performance on each request but saves model instance in unsafe storage (default django cache). The instance is serialized to a string using the `standard python module pickle`[11] and then encrypted with Vigenère cipher[12]. Read more in the *`vstutils.utils.SecurePickling`* documentation. Default: false.

2.3 Databases settings

Section `[databases]`.

The main section that is designed to manage multiple databases connected to the project.

These settings are for all databases and are vendor-independent, with the exception of tablespace management.

- **default_tablespace** - Default tablespace to use for models that don't specify one, if the backend supports it. A tablespace is a storage location on a database server where the physical data files corresponding to database tables are stored. It allows you to organize and manage the storage of your database tables, specifying the location on disk where the table data is stored. Configuring tablespaces can be beneficial for various reasons, such as optimizing performance by placing specific tables or indexes (with `default_index_tablespace`) on faster storage devices, managing disk space efficiently, or segregating data for administrative purposes. It provides a level of control over the physical organization of data within the database, allowing developers to tailor storage strategies based on the requirements and characteristics of their application. Read more at Declaring tablespaces for tables[13].

[11] https://docs.python.org/3.8/library/pickle.html#module-pickle
[12] https://en.wikipedia.org/wiki/Vigenèrecipher
[13] https://docs.djangoproject.com/en/4.2/topics/db/tablespaces/#declaring-tablespaces-for-tables

- **default_index_tablespace** - Default tablespace to use for indexes on fields that don't specify one, if the backend supports it. Read more at Declaring tablespaces for indexes[14].
- **databases_without_cte_support** - A comma-separated list of database section names that do not support CTEs (Common Table Expressions).

> **Warning:** Although MariaDB supports Common Table Expressions, but database connected to MariaDB still needs to be added to `databases_without_cte_support` list. The problem is that the implementation of recursive queries in the MariaDB does not allow using it in a standard form. MySQL (since 8.0) works as expected.

Also, all subsections of this section are available connections to the DBMS. So the `databases.default` section will be used by django as the default connection.

Here you can change settings related to database, which vstutils-based application will use. vstutils-based application supports all databases supported by `django`. List of supported out of the box: SQLite (default choice), MySQL, Oracle, or PostgreSQL. Configuration details available at Django database documentation[15]. To run vstutils-based application at multiple nodes (cluster), use client-server database (SQLite not suitable) shared for all nodes.

You can also set the base template for connecting to the database in the `database` section.

Section `[database]`.

This section is designed to define the basic template for connections to various databases. This can be useful to reduce the list of settings in the `databases.*` subsections by setting the same connection for a different set of databases in the project. For more details read the django docs about Multiple databases[16]

There is a list of settings, required for MySQL/MariaDB database.

Firstly, if you use MySQL/MariaDB and you have set timezone different from "UTC" you should run command below:

```
mysql_tzinfo_to_sql /usr/share/zoneinfo | mysql -u root -p mysql
```

Secondly, to use MySQL/MariaDB set following options in `settings.ini` file:

```
[database.options]
connect_timeout = 10
init_command = SET sql_mode='STRICT_TRANS_TABLES', default_storage_engine=INNODB,␣
↪NAMES 'utf8', CHARACTER SET 'utf8', SESSION collation_connection = 'utf8_unicode_ci'
```

Finally, add some options to MySQL/MariaDB configuration:

```
[client]
default-character-set=utf8
init_command = SET collation_connection = @@collation_database

[mysqld]
character-set-server=utf8
collation-server=utf8_unicode_ci
```

[14] https://docs.djangoproject.com/en/4.2/topics/db/tablespaces/#declaring-tablespaces-for-indexes
[15] https://docs.djangoproject.com/en/4.2/ref/settings/#databases
[16] https://docs.djangoproject.com/en/4.2/topics/db/multi-db/#multiple-databases

2.4 Cache settings

Section `[cache]`.

This section is cache backend related settings used by vstutils-based application. vstutils supports all cache backends that Django does. Filesystem, in-memory, memcached are supported out of the box and many others are supported with additional plugins. You can find details about cache configs supported Django caches documentation[17]. In clusters we advice to share cache between nodes to improve performance using client-server cache realizations. We recommend to use Redis in production environments.

2.4.1 Tarantool Cache Backend for Django

The `TarantoolCache` is a custom cache backend for Django that allows you to use Tarantool as a caching mechanism. To use this backend, you need to configure the following settings in your project's configuration:

```
[cache]
location = localhost:3301
backend = vstutils.drivers.cache.TarantoolCache

[cache.options]
space = default
user = guest
password = guest
```

Explanation of Settings:

- **location** - The host name and port for connecting to the Tarantool server.
- **backend** - The path to the TarantoolCache backend class.
- **space** - The name of the space in Tarantool to use as the cache (default is `DJANGO_CACHE`).
- **user** - The username for connecting to the Tarantool server (default is `guest`).
- **password** - The password for connecting to the Tarantool server. Optional.

Additionally, you can set the `connect_on_start` variable in the `[cache.options]` section. When set to `true` value, this variable triggers an initial connection to the Tarantool server to configure spaces and set up the service for automatic removal of outdated entries.

Warning: Note that this requires the `expirationd` module to be installed on the Tarantool server.

Note: When utilizing Tarantool as a cache backend in VST Utils, temporary spaces are automatically created to facilitate seamless operation. These temporary spaces are dynamically generated as needed and are essential for storing temporary data efficiently.

It's important to mention that while temporary spaces are automatically handled, if you intend to use persistent spaces on disk, it is necessary to pre-create them on the Tarantool server with schema settings similar to those used by the VST Utils configuration. Ensure that any persistent spaces required for your application are appropriately set up on the Tarantool server with the same schema configurations for consistent and reliable operation.

[17] https://docs.djangoproject.com/en/4.2/ref/settings/#caches

Note: It's important to note that this cache driver is unique to vstutils and tailored to seamlessly integrate with the VST Utils framework.

2.5 Locks settings

Section `[locks]`.

Locks is a system that vstutils-based application uses to avoid damage from parallel actions working on the same entity simultaneously. It is based on Django cache, so there is another bunch of same settings as *cache*. And why there is another section for them, you may ask. Because cache backend is used for locking must provide some guarantees, which do not required to usual cache: it MUST be shared for all vstutils-based application threads and nodes. So, for example, in-memory backend is not suitable. In case of clusterization we strongly recommend to use Tarantool, Redis or Memcached as backend because they have enough speed for this purposes. Cache and locks backend can be the same, but don't forget about requirement we said above.

2.6 Session cache settings

Section `[session]`.

vstutils-based application store sessions in *database*, but for better performance, we use a cache-based session backend. It is based on Django cache, so there is another bunch of same settings as *cache*. By default, settings are got from *cache*.

2.7 Rpc settings

Section `[rpc]`.

Celery is a distributed task queue system for handling asynchronous tasks in web applications. Its primary role is to facilitate the execution of background or time-consuming tasks independently from the main application logic. Celery is particularly useful for offloading tasks that don't need to be processed immediately, improving the overall responsiveness and performance of an application.

Key features and roles of Celery in an application with asynchronous tasks include:

1. Asynchronous Task Execution: Celery allows developers to define tasks as functions or methods and execute them asynchronously. This is beneficial for tasks that might take a considerable amount of time, such as sending emails, processing data, or generating reports.
2. Distributed Architecture: Celery operates in a distributed manner, making it suitable for large-scale applications. It can distribute tasks across multiple worker processes or even multiple servers, enhancing scalability and performance.
3. Message Queue Integration: Celery relies on message brokers (such as RabbitMQ, Redis, Tarantool, SQS or others) to manage the communication between the main application and the worker processes. This decoupling ensures reliable task execution and allows for the efficient handling of task queues.
4. Periodic Tasks: Celery includes a scheduler that enables the execution of periodic or recurring tasks. This is useful for automating tasks that need to run at specific intervals, like updating data or performing maintenance operations.
5. Error Handling and Retry Mechanism: Celery provides mechanisms for handling errors in tasks and supports automatic retries. This ensures robustness in task execution, allowing the system to recover from transient failures.

6. Task Result Storage: Celery supports storing the results of completed tasks, which can be useful for tracking task progress or retrieving results later. This feature is especially valuable for long-running tasks.

vstutils-based application uses Celery for long-running async tasks. Those settings relate to this broker and Celery itself. Those kinds of settings: broker backend, number of worker-processes per node and some settings used for troubleshoot server-broker-worker interaction problems.

This section require vstutils with *rpc* extra dependency.

- **connection** - Celery broker connection[18]. Default: `filesystem:///var/tmp`.
- **concurrency** - Count of celery worker threads. Default: 4.
- **heartbeat** - Interval between sending heartbeat packages, which says that connection still alive. Default: 10.
- **enable_worker** - Enable or disable worker with webserver. Default: true.

The following variables from Django settings[19] are also supported (with the corresponding types):

- **prefetch_multiplier** - CELERYD_PREFETCH_MULTIPLIER[20]
- **max_tasks_per_child** - CELERYD_MAX_TASKS_PER_CHILD[21]
- **results_expiry_days** - CELERY_RESULT_EXPIRES[22]
- **default_delivery_mode** - CELERY_DEFAULT_DELIVERY_MODE[23]
- **task_send_sent_event** - CELERY_DEFAULT_DELIVERY_MODE[24]
- **worker_send_task_events** - CELERY_DEFAULT_DELIVERY_MODE[25]

VST Utils provides seamless support for using Tarantool as a transport for Celery, allowing efficient and reliable message passing between distributed components. To enable this feature, ensure that the Tarantool server has the *queue* and *expirationd* modules installed.

To configure the connection, use the following example URL: `tarantool://guest@localhost:3301/rpc`

- `tarantool://`: Specifies the transport.
- `guest`: Authentication parameters (in this case, no password).
- `localhost`: Server address.
- `3301`: Port for connection.
- `rpc`: Prefix for queue names and/or result storage.

VST Utils also supports Tarantool as a backend for storing Celery task results. Connection string is similar to the transport.

Note: When utilizing Tarantool as a result backend or transport in VST Utils, temporary spaces and queues are automatically created to facilitate seamless operation. These temporary spaces are dynamically generated as needed and are essential for storing temporary data efficiently.

It's important to mention that while temporary spaces are automatically handled, if you intend to use persistent spaces on disk, it is necessary to pre-create them on the Tarantool server with schema settings similar to those used by the VST Utils

[18] http://docs.celeryproject.org/en/latest/userguide/configuration.html#conf-broker-settings
[19] http://docs.celeryproject.org/en/latest/userguide/configuration.html#new-lowercase-settings
[20] http://docs.celeryproject.org/en/latest/userguide/configuration.html#std-setting-worker_prefetch_multiplier
[21] http://docs.celeryproject.org/en/latest/userguide/configuration.html#std-setting-worker_max_tasks_per_child
[22] http://docs.celeryproject.org/en/latest/userguide/configuration.html#std-setting-result_expires
[23] http://docs.celeryproject.org/en/latest/userguide/configuration.html#task-default-delivery-mode
[24] http://docs.celeryproject.org/en/latest/userguide/configuration.html#task_send_sent_event
[25] http://docs.celeryproject.org/en/latest/userguide/configuration.html#worker_send_task_events

configuration. Ensure that any persistent spaces required for your application are appropriately set up on the Tarantool server with the same schema configurations for consistent and reliable operation.

2.8 Worker settings

Section `[worker]`.

Warning: These settings are needed only for rpc-enabled applications.

Celery worker options:

- **loglevel** - Celery worker log level. Default: from *main* section `log_level`.
- **pidfile** - Celery worker pidfile. Default: `/run/{app_name}_worker.pid`
- **autoscale** - Options for autoscaling. Two comma separated numbers: max,min.
- **beat** - Enable or disable celery beat scheduler. Default: `true`.

See other settings via `celery worker --help` command.

2.9 SMTP settings

Section `[mail]`.

Django comes with several email sending backends. With the exception of the SMTP backend (default when `host` is set), these backends are useful only in testing and development.

Applications based on vstutils uses only `smtp` and `console` backends. These two backends serve distinct purposes in different environments. The SMTP backend ensures the reliable delivery of emails in a production setting, while the console backend provides a convenient way to inspect emails during development without the risk of unintentional communication with external mail servers. Developers often switch between these backends based on the context of their work, choosing the appropriate one for the stage of development or testing.

- **host** - IP or domain for smtp-server. If it not set vstutils uses `console` backends. Default: `None`.
- **port** - Port for smtp-server connection. Default: `25`.
- **user** - Username for smtp-server connection. Default: `""`.
- **password** - Auth password for smtp-server connection. Default: `""`.
- **tls** - Enable/disable tls for smtp-server connection. Default: `False`.
- **send_confirmation** - Enable/disable confirmation message after registration. Default: `False`.
- **authenticate_after_registration** - Enable/disable autologin after registration confirmation. Default: `False`.

2.10 Web settings

Section `[web]`.

These settings are related to web-server. Those settings includes: session_timeout, static_files_url and pagination limit.

- **allow_cors** - enable cross-origin resource sharing. Default: `False`.
- **cors_allowed_origins**, **cors_allowed_origins_regexes**, **cors_expose_headers**, **cors_allow_methods**, **cors_allow_headers**, **cors_preflight_max_age** - Settings[26] from `django-cors-headers` lib with their defaults.
- **enable_gravatar** - Enable/disable gravatar service using for users. Default: `True`.
- **rest_swagger_description** - Help string in Swagger schema. Useful for dev-integrations.
- **openapi_cache_timeout** - Cache timeout for storing schema data. Default: `120`.
- **health_throttle_rate** - Count of requests to `/api/health/` endpoint. Default: `60`.
- **bulk_threads** - Threads count for PATCH `/api/endpoint/` endpoint. Default: `3`.
- **session_timeout** - Session lifetime. Default: `2w` (two weeks).
- **etag_default_timeout** - Cache timeout for Etag headers to control models caching. Default: `1d` (one day).
- **rest_page_limit** and **page_limit** - Default limit of objects in API list. Default: `1000`.
- **session_cookie_domain** - The domain to use for session cookies. Read more[27]. Default: `None`.
- **csrf_trusted_origins** - A list of hosts which are trusted origins for unsafe requests. Read more[28]. Default: from **session_cookie_domain**.
- **case_sensitive_api_filter** - Enables/disables case sensitive search for name filtering. Default: `True`.
- **secure_proxy_ssl_header_name** - Header name which activates SSL urls in responses. Read more[29]. Default: `HTTP_X_FORWARDED_PROTOCOL`.
- **secure_proxy_ssl_header_value** - Header value which activates SSL urls in responses. Read more[30]. Default: `https`.

The following variables from Django settings are also supported (with the corresponding types):

- **secure_browser_xss_filter** - SECURE_BROWSER_XSS_FILTER[31]
- **secure_content_type_nosniff** - SECURE_CONTENT_TYPE_NOSNIFF[32]
- **secure_hsts_include_subdomains** - SECURE_HSTS_INCLUDE_SUBDOMAINS[33]
- **secure_hsts_preload** - SECURE_HSTS_PRELOAD[34]
- **secure_hsts_seconds** - SECURE_HSTS_SECONDS[35]
- **password_reset_timeout_days** - PASSWORD_RESET_TIMEOUT_DAYS[36]

[26] https://github.com/adamchainz/django-cors-headers#configuration
[27] https://docs.djangoproject.com/en/4.2/ref/settings/#std:setting-SESSION_COOKIE_DOMAIN
[28] https://docs.djangoproject.com/en/4.2/ref/settings/#csrf-trusted-origins
[29] https://docs.djangoproject.com/en/4.2/ref/settings/#secure-proxy-ssl-header
[30] https://docs.djangoproject.com/en/4.2/ref/settings/#secure-proxy-ssl-header
[31] https://docs.djangoproject.com/en/4.2/ref/settings/#secure-browser-xss-filter
[32] https://docs.djangoproject.com/en/4.2/ref/settings/#secure-content-type-nosniff
[33] https://docs.djangoproject.com/en/4.2/ref/settings/#secure-hsts-include-subdomains
[34] https://docs.djangoproject.com/en/4.2/ref/settings/#secure-hsts-preload
[35] https://docs.djangoproject.com/en/4.2/ref/settings/#secure-hsts-seconds
[36] https://docs.djangoproject.com/en/4.2/ref/settings/#std:setting-PASSWORD_RESET_TIMEOUT

- **request_max_size** - DATA_UPLOAD_MAX_MEMORY_SIZE[37]
- **x_frame_options** - X_FRAME_OPTIONS[38]
- **use_x_forwarded_host** - USE_X_FORWARDED_HOST[39]
- **use_x_forwarded_port** - USE_X_FORWARDED_PORT[40]

The following settings affects prometheus metrics endpoint (which can be used for monitoring application):

- **metrics_throttle_rate** - Count of requests to `/api/metrics/` endpoint. Default: `120`.
- **enable_metrics** - Enable/disable `/api/metrics/` endpoint for app. Default: `true`
- **metrics_backend** - Python class path with metrics collector backend. Default: `vstutils.api.metrics.DefaultBackend` Default backend collects metrics from uwsgi workers and python version info.

Section `[uvicorn]`.

You can configure the necessary settings to run the uvicorn server. `vstutils` supports almost all options from the cli, except for those that configure the application and connection.

See all available uvicorn settings via `uvicorn --help` command.

2.11 Centrifugo client settings

Section `[centrifugo]`.

Centrifugo is employed to optimize real-time data updates within a Django application by orchestrating seamless communication among its various components. The operational paradigm involves the orchestrated generation of Django signals, specifically `post_save` and `post_delete` signals, dynamically triggered during HTTP requests or the execution of Celery tasks. These signals, when invoked on user or BaseModel-derived models within the vstutils framework, initiate the creation of messages destined for all subscribers keen on the activities related to these models. Subsequent to the completion of the HTTP request or Celery task, the notification mechanism dispatches tailored messages to all relevant subscribers. In effect, each active browser tab with a pertinent subscription promptly receives a notification, prompting an immediate data update request. Centrifugo's pivotal role lies in obviating the necessity for applications to engage in periodic REST API polling at fixed intervals (e.g., every 5 seconds). This strategic elimination of redundant requests significantly alleviates the REST API's operational load, rendering it more scalable to accommodate a larger user base. Importantly, this real-time communication model ensures prompt and synchronized data updates, fostering a highly responsive user experience.

To install app with centrifugo client, `[centrifugo]` section must be set. Centrifugo is used by application to auto-update page data. When user change some data, other clients get notification on channel with model label and primary key. Without the service all GUI-clients get page data every 5 seconds (by default).

- **address** - Centrifugo server address.
- **api_key** - API key for clients.
- **token_hmac_secret_key** - API key for jwt-token generation.
- **timeout** - Connection timeout.
- **verify** - Connection verification.
- **subscriptions_prefix** - Prefix used for generating update channels, by default "{VST_PROJECT}.update".

[37] https://docs.djangoproject.com/en/4.2/ref/settings/#std:setting-DATA_UPLOAD_MAX_MEMORY_SIZE
[38] https://docs.djangoproject.com/en/4.2/ref/settings/#x-frame-options
[39] https://docs.djangoproject.com/en/4.2/ref/settings/#use-x-forwarded-host
[40] https://docs.djangoproject.com/en/4.2/ref/settings/#use-x-forwarded-port

Note: These settings also add parameters to the OpenAPI schema and change how the auto-update system works in the GUI. `token_hmac_secret_key` is used for jwt-token generation (based on session expiration time). Token will be used for Centrifugo-JS client.

2.12 Storage settings

Section `[storages]`.

Applications based on `vstutils` supports filesystem storage out of box. Setup `media_root` and `media_url` in `[storages.filesystem]` section to configure custom media dir and relative url. By default it would be `{/path/to/project/module}/media` and `/media/`.

Applications based on `vstutils` supports store files in external services with Apache Libcloud[41] and Boto3[42].

Apache Libcloud settings grouped by sections named `[storages.libcloud.provider]`, where `provider` is name of storage. Each section has four keys: `type`, `user`, `key` and `bucket`. Read more about the settings in django-storages libcloud docs[43]

This setting is required to configure connections to cloud storage providers. Each entry corresponds to a single 'bucket' of storage. You can have multiple buckets for a single service provider (e.g., multiple S3 buckets), and you can define buckets at multiple providers.

For `Boto3` all settings grouped by section named `[storages.boto3]`. Section must contain following keys: `access_key_id`, `secret_access_key`, `storage_bucket_name`. Read more about the settings in django-storages amazon-S3 docs[44]

Storage has following priority to choose storage engine if multiple was provided:

1. Libcloud store when config contains this section.
2. Boto3 store, when you have section and has all required keys.
3. FileSystem store otherwise.

Once you have defined your Libcloud providers, you have an option of setting one provider as the default provider of Libcloud storage. You can do it by setup `[storages.libcloud.default]` section or vstutils will set the first storage as default.

If you configure default libcloud provider, vstutils will use it as global file storage. To override it set `default=django.core.files.storage.FileSystemStorage` in `[storages]` section. When `[storages.libcloud.default]` is empty `django.core.files.storage.FileSystemStorage` is used as default. To override it set `default=storages.backends.apache_libcloud.LibCloudStorage` in `[storages]` section and use Libcloud provider as default.

Here is example for boto3 connection to minio cluster with public read permissions, external proxy domain and internal connection support:

```
[storages.boto3]
access_key_id = EXAMPLE_KEY
secret_access_key = EXAMPLEKEY_SECRET
# connection to internal service behind proxy
s3_endpoint_url = http://127.0.0.1:9000/
```

(continues on next page)

[41] http://libcloud.apache.org/
[42] https://boto3.amazonaws.com/v1/documentation/api/latest/index.html
[43] https://django-storages.readthedocs.io/en/latest/backends/apache_libcloud.html#libcloud-providers
[44] https://django-storages.readthedocs.io/en/latest/backends/amazon-S3.html

(continued from previous page)

```
# external domain to bucket 'media'
storage_bucket_name = media
s3_custom_domain = media-api.example.com/media
# external domain works behind tls
s3_url_protocol = https:
s3_secure_urls = true
# settings to connect as plain http for uploading
s3_verify = false
s3_use_ssl = false
# allow to save files with similar names by adding prefix
s3_file_overwrite = false
# disables query string auth and setup default acl as RO for public users
querystring_auth = false
default_acl = public-read
```

2.13 Throttle settings

Section `[throttle]`.

By including this section to your config, you can setup global and per-view throttle rates. Global throttle rates are specified under root [throttle] section.To specify per-view throttle rate, you need to include child section.

For example, if you want to apply throttle to `api/v1/author`:

```
[throttle.views.author]
rate=50/day
actions=create,update
```

- **rate** - Throttle rate in format number_of_requests/time_period. Expected time_periods: second/minute/hour/day.
- **actions** - Comma separated list of drf actions. Throttle will be applied only on specified here actions. Default: update, partial_update.

More on throttling at DRF Throttle docs[45].

2.14 Web Push settings

Section `[webpuwsh]`.

- **enabled**: A boolean flag that enables or disables web push notifications. Set to *true* to activate web push notifications, and *false* to deactivate them. Default: *false*. If set to false then notifications settings on user page will be hidden and *send* method of notification class will do nothing.
- **vapid_private_key**, **vapid_public_key**: These are the application server keys used for sending push notifications. The VAPID (Voluntary Application Server Identification) keys consist of a public and a private key. These keys are essential for secure communication between your server and the push service. For generating a VAPID key pair and understanding their usage, refer to the detailed guide available here: Creating VAPID Keys[46].
- **vapid_admin_email**: This setting specifies the email address of the administrator or the person responsible for the server. It is a contact point for the push service to get in touch in case of any issues or policy violations.

[45] https://www.django-rest-framework.org/api-guide/throttling/
[46] https://web.dev/articles/push-notifications-subscribing-a-user#how_to_create_application_server_keys

- **default_notification_icon**: URL of the default icon image to be used for web push notifications, to avoid confusion absolute URL is preferred. This icon will be displayed in the notifications if no other icon is specified at the notification level. More information about icon can be found here[47].

For more detailed guidance on using and implementing web push notifications in VSTUtils, refer to the Web Push manual provided *here*.

Remember, these settings are crucial for the proper functioning and reliability of web push notifications in your application. Ensure that they are configured accurately for optimal performance.

2.15 Production web settings

Section `[uwsgi]`.

Settings related to web-server used by vstutils-based application in production (for deb and rpm packages by default). Most of them related to system paths (logging, PID-file and so on). More settings in uWSGI docs[48].

But keep in mind that uWSGI is deprecated and may be removed in future releases. Use the uvicorn settings to manage your app server.

2.16 Working behind the proxy server with TLS

2.16.1 Nginx

To configure vstutils for operation behind Nginx with TLS, follow these steps:

1. **Install Nginx:**

Ensure that Nginx is installed on your server. You can install it using the package manager specific to your operating system.

2. **Configure Nginx:**

Create an Nginx configuration file for your vstutils application. Below is a basic example of an Nginx configuration. Adjust the values based on your specific setup.

```
server {
    listen 80;
    server_name your_domain.com;

    return 301 https://$host$request_uri;
}

server {
    listen 443 ssl;
    server_name your_domain.com;

    ssl_certificate /path/to/your/certificate.crt;
    ssl_certificate_key /path/to/your/private.key;
    ssl_protocols TLSv1.2 TLSv1.3;
    ssl_ciphers 'TLS_AES_128_GCM_SHA256:TLS_AES_256_GCM_SHA384:TLS_CHACHA20_POLY1305_
↪SHA256:ECDHE-RSA-AES128-GCM-SHA256:ECDHE-RSA-AES256-GCM-SHA384';
```

(continues on next page)

[47] https://web.dev/articles/push-notifications-display-a-notification#icon
[48] http://uwsgi-docs.readthedocs.io/en/latest/Configuration.html

(continued from previous page)

```
    gzip            on;
    gzip_types      text/plain application/xml application/json application/
↪openapi+json text/css application/javascript;
    gzip_min_length 1000;

    charset utf-8;

    location / {
        proxy_pass http://127.0.0.1:8080;  # Assuming application is running on the␣
↪default port
        proxy_set_header Host $host;
        proxy_set_header X-Real-IP $remote_addr;
        proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
        proxy_set_header X-Forwarded-Proto https;  # Set to 'https' since it's a␣
↪secure connection
        proxy_set_header X-Forwarded-Host   $host;
        proxy_set_header X-Forwarded-Port   $server_port;
    }
}
```

Replace `your_domain.com` with your actual domain, and update the paths for SSL certificates.

3. **Update vstutils settings:**

Ensure that your vstutils settings have the correct configurations for HTTPS. In your `/etc/vstutils/settings.ini` (or project `settings.ini`):

```
[web]
secure_proxy_ssl_header_name = HTTP_X_FORWARDED_PROTO
secure_proxy_ssl_header_value = https
```

This ensures that vstutils recognizes the HTTPS connection.

4. **Restart Nginx:**

After making these changes, restart Nginx to apply the new configurations:

```
sudo systemctl restart nginx
```

Now, your vstutils application should be accessible via HTTPS through Nginx. Adjust these instructions based on your specific environment and security considerations.

2.16.2 Traefik

To configure vstutils for operation behind Traefik with TLS, follow these steps:

1. **Install Traefik:**

Ensure that Traefik is installed on your server. You can download the binary from the official website or use a package manager specific to your operating system.

2. **Configure Traefik:**

Create a Traefik configuration file `/path/to/traefik.toml`. Here's a basic example:

3. **Create Traefik Toml Configuration:**

Create the `/path/to/traefik_config.toml` file with the following content:

```
[http.routers]
  [http.routers.vstutils]
    rule = "Host(`your_domain.com`)"
    entryPoints = ["websecure"]
    service = "vstutils"
    middlewares = ["customheaders", "compress"]

[http.middlewares]
  [http.middlewares.customheaders.headers.customRequestHeaders]
    X-Forwarded-Proto = "https"

  [http.middlewares.compress.compress]
    compress = true

[http.services]
  [http.services.vstutils.loadBalancer]
    [[http.services.vstutils.loadBalancer.servers]]
      url = "http://127.0.0.1:8080"  # Assuming application is running on the default␣
↪port
```

Make sure to replace `your_domain.com` with your actual domain.

4. **Update vstutils settings:**

Ensure that your vstutils settings have the correct configurations for HTTPS. In your `/etc/vstutils/settings.ini` (or project `settings.ini`):

5. **Start Traefik:**

Start Traefik with the following command:

```
traefik --configfile /path/to/traefik.toml
```

Now, your vstutils application should be accessible via HTTPS through Traefik. Adjust these instructions based on your specific environment and requirements.

2.16.3 HAProxy

1. **Install HAProxy:**

Ensure that HAProxy is installed on your server. You can install it using the package manager specific to your operating system.

2. **Configure HAProxy:**

Create an HAProxy configuration file for your vstutils application. Below is a basic example of an HAProxy configuration. Adjust the values based on your specific setup.

```
frontend http-in
    bind *:80
    mode http
    redirect scheme https code 301 if !{ ssl_fc }

frontend https-in
    bind *:443 ssl crt /path/to/your/certificate.pem
    mode http
    option forwardfor
    http-request set-header X-Forwarded-Proto https
```

(continues on next page)

(continued from previous page)

```
    default_backend vstutils_backend

backend vstutils_backend
    mode http
    server vstutils-server 127.0.0.1:8080 check
```

Replace `your_domain.com` with your actual domain and update the paths for SSL certificates.

3. **Update vstutils settings:**

Ensure that your vstutils settings have the correct configurations for HTTPS. In your `/etc/vstutils/settings.ini` (or project `settings.ini`):

4. **Restart HAProxy:**

After making these changes, restart HAProxy to apply the new configurations:

```
sudo systemctl restart haproxy
```

Now, your vstutils application should be accessible via HTTPS through HAProxy. Adjust these instructions based on your specific environment and security considerations.

2.17 Configuration options

This section contains additional information for configure additional elements.

1. If you need set `https` for your web settings, you can do it using HAProxy, Nginx, Traefik or configure it in `settings.ini`.

```
[uwsgi]
addrport = 0.0.0.0:8443

[uvicorn]
ssl_keyfile = /path/to/key.pem
ssl_certfile = /path/to/cert.crt
```

1. We strictly do not recommend running the web server from root. Use HTTP proxy to run on privileged ports.
2. You can use *{ENV[HOME:-value]}* (where *HOME* is environment variable, *value* is default value) in configuration values.
3. You can use environment variables for setup important settings. But config variables has more priority then env. Available settings are: `DEBUG`, `DJANGO_LOG_LEVEL`, `TIMEZONE` and some settings with `[ENV_NAME]` prefix.

   For project without special settings and project levels named `project` these variables will start with `PROJECT_` prefix. There is a list of these variables: `{ENV_NAME}_ENABLE_ADMIN_PANEL`, `{ENV_NAME}_ENABLE_REGISTRATION`, `{ENV_NAME}_MAX_TFA_ATTEMPTS`, `{ENV_NAME}_ETAG_TIMEOUT`, `{ENV_NAME}_SEND_CONFIRMATION_EMAIL`, `{ENV_NAME}_SEND_EMAIL_RETRIES`, `{ENV_NAME}_SEND_EMAIL_RETRY_DELAY`, `{ENV_NAME}_AUTHENTICATE_AFTER_REGISTRATION`, `{ENV_NAME}_MEDIA_ROOT` (dir with uploads), `{ENV_NAME}_GLOBAL_THROTTLE_RATE`, and `{ENV_NAME}_GLOBAL_THROTTLE_ACTIONS`.

   There are also URI-specific variables for connecting to various services such as databases and caches. There are `DATABASE_URL`, `CACHE_URL`, `LOCKS_CACHE_URL`, `SESSIONS_CACHE_URL` and `ETAG_CACHE_URL`. As you can see from the names, they are closely related to the keys and names of the corresponding config sections.

4. We recommend to install `uvloop` to your environment and setup `loop = uvloop` in `[uvicorn]` section for performance reasons.

In the context of vstutils, the adoption of `uvloop` is paramount for optimizing the performance of the application, especially because utilizing `uvicorn` as the ASGI server. `uvloop` is an ultra-fast, drop-in replacement for the default event loop provided by Python. It is built on top of `libuv`, a high-performance event loop library, and is specifically designed to optimize the execution speed of asynchronous code.

By leveraging `uvloop`, developers can achieve substantial performance improvements in terms of reduced latency and increased throughput. This is especially critical in scenarios where applications handle a large number of concurrent connections. The improved efficiency of event loop handling directly translates to faster response times and better overall responsiveness of the application.

3 Backend API manual

VST Utils framework uses Django, Django Rest Framework, drf-yasg and Celery.

3.1 Models

A model is the single, definitive source of truth about your data. It contains essential fields and behavior for the data you're storing. Usually best practice is to avoid writing views and serializers manually, as BModel provides plenty of Meta attributes to autogenerate serializers and views for many use cases.

Default Django model classes overrides in *vstutils.models* module.

class vstutils.models.**BModel**(**args*, ***kwargs*)

Default model class that generates model viewset, separate serializers for list() and retrieve(), filters, api endpoints and nested views.

Examples:

```
from django.db import models
from rest_framework.fields import ChoiceField
from vstutils.models import BModel

class Stage(BModel):
    name = models.CharField(max_length=256)
    order = models.IntegerField(default=0)

    class Meta:
        default_related_name = "stage"
        ordering = ('order', 'id',)
        # fields which would be showed on list.
        _list_fields = [
            'id',
            'name',
        ]
        # fields which would be showed on detail view and creation.
        _detail_fields = [
            'id',
            'name',
            'order'
        ]
```

(continues on next page)

(continued from previous page)

```
        # make order as choices from 0 to 9
        _override_detail_fields = {
            'order': ChoiceField((str(i) for i in range(10)))
        }


class Task(BModel):
    name = models.CharField(max_length=256)
    stages = models.ManyToManyField(Stage)
    _translate_model = 'Task'

    class Meta:
        # fields which would be showed.
        _list_fields = [
            'id',
            'name',
        ]
        # create nested views from models
        _nested = {
            'stage': {
                'allow_append': False,
                'model': Stage
            }
        }
```

In this case, you create models which could converted to simple view, where:

- `POST/GET` to `/api/version/task/` - creates new or get list of tasks
- `PUT/PATCH/GET/DELETE` to `/api/version/task/:id/` - updates, retrieves or removes instance of task
- `POST/GET` to `/api/version/task/:id/stage/` - creates new or get list of stages in task
- `PUT/PATCH/GET/DELETE` to `/api/version/task/:id/stage/:stage_id` - updates, retrieves or removes instance of stage in task.

To attach a view to an API insert the following code in *settings.py*:

```
API[VST_API_VERSION][r'task'] = {
    'model': 'your_application.models.Task'
}
```

For primary access to generated view inherit from *Task.generated_view* property.

To make translation on frontend easier use `_translate_model` attribute with model_name.

List of meta-attributes for generating a view:

- `_view_class` - list of additional view classes to inherit from, class or string to import with base class ViewSet. Constants are also supported:
  - `read_only` - to create a view only for viewing;
  - `list_only` - to create a view with list only;
  - `history` - to create a view only for viewing and deleting records.

  CRUD-view is applied by default.

- `_serializer_class` - class of API serializer; use this attribute to specify parent class for auto-generated serializers. Default is *vstutils.api.serializers.VSTSerializer*. Can take a string to import, serializer class or `django.utils.functional.SimpleLazyObject`.
- `_serializer_class_name` - model name for OpenAPI definitions.This would be a model name in generated admin interface. Default is name of model class.
- `_list_fields` or `_detail_fields` - list of fields which will be listed in entity list or detail view accordingly. Same as DRF serializers meta-attribute "fields".
- `_override_list_fields` or `_override_detail_fields` - mapping with names and field types that will be redeclared in serializer attributes(think of it as declaring fields in DRF ModelSerializer).
- `_properties_groups` - dict with key as group name and value as list of fields(str). Allows to group fields in sections on frontend.
- `_view_field_name` - name of field frontend shows as main view name.
- `_non_bulk_methods` - list of methods which must not used via bulk requests.
- `_extra_serializer_classes` - mapping with additional serializers in viewset. For example, custom serializer, which will compute smth in action (mapping name). Value can be string for import. Important note: setting *model* attribute to None allows to use standard serializer generation mechanism and get fields from a list or detail serializer (set *__inject_from__* serializer's meta attribute to *list* or *detail* accordingly). In some cases, it is required to pass the model to the serializer. For these purposes, the constant *LAZY_MODEL* can be used as a meta attribute. Each time the serializer is used, the exact model where this serializer was declared will be set.
- `_filterset_fields` - list/dict of filterset names for API-filtering. Default is list of fields in list view. During processing a list of fields checks for the presence of special field names and inherit additional parent classes. If the list contains `id`, class will inherit from *vstutils.api.filters.DefaultIDFilter*. If the list contains `name`, class will inherit from *vstutils.api.filters.DefaultNameFilter*. If both conditions are present, inheritance will be from all of the above classes. Possible values include *list* of fields to filter or *dict* where key is a field name and value is a Filter class. Dict extends attribute functionality and provides ability to override filter field class (None value disables overriding).
- `_search_fields` - tuple or list of fields using for search requests. By default (or *None*) get all filterable fields in detail view.
- `_copy_attrs` - list of model-instance attributes indicates that object is copyable with this attrs.
- `_nested` - key-value mapping with nested views (key - nested name, kwargs for *vstutils.api.decorators.nested_view* decorator but supports `model` attribute as nested). `model` can be string for import. Use `override_params` when you need to override generated view parameters for nested view (works only with `model` as view).
- `_extra_view_attributes` - key-value mapping with additional view attributes, but has less priority over generated attributes.

In common, you can also add custom attributes to override or extend the default list of processing classes. Supported view attributes are `filter_backends`, `permission_classes`, `authentication_classes`, `throttle_classes`, `renderer_classes` and `parser_classes`. List of meta-attributes for settings of view is looks like:

- `_pre_{attribute}` - List of classes included before defaults.
- `_{attribute}` - List of classes included after defaults.
- `_override_{attribute}` - boolean flag indicates that attribute override default viewset (otherwise appends). Default is `False`.

Note: You may need to create an action[49] on generated view. Use *vstutils.models.decorators.register_view_action* decorator with the `detail` argument to determine applicability to a list or detail entry. In this case, the decorated method will take an instance of the view object as `self` attribute.

Note: In some cases, inheriting models may require to inherit Meta class from the base model. If the Meta is explicitly declared in the base class, then you can get it through the attribute *OriginalMeta* and use it for inheritance.

Note: Docstring of model will be reused for view descriptions. It is possible to write both a general description for all actions and description for each action using the following syntax:

```
General description for all actions.

action_name:
    Description for this action.

another_action:
    Description for another action.
```

The `get_view_class()` method is a utility method in the Django ORM model designed to facilitate the configuration and instantiation of Django Rest Framework (DRF) Generic ViewSets. It allows developers to define and customize various aspects of the associated DRF view class.

Examples:

```
# Create simple list view with same fields
TaskViewSet = Task.get_view_class(view_class='list_only')

# Create view with overriding nested view params
from rest_framework.mixins import CreateModelMixin

TaskViewSet = Task.get_view_class(
    nested={
        "milestones": {
            "model": Stage,
            "override_params": {
                "view_class": ("history", CreateModelMixin)
            },
        },
    },
)
```

Developers can use this method to customize various aspects of the associated view, such as serializer classes, field configurations, filter backends, permission classes, etc. It uses attributes declared in meta attributes, but allows individual parts to be overriden.

hidden

If hidden is set to True, entry will be excluded from query in BQuerySet.

id

Primary field for select and search in API.

[49] https://www.django-rest-framework.org/api-guide/viewsets/#marking-extra-actions-for-routing

class vstutils.models.**Manager**(**args*, ***kwargs*)

Default VSTUtils manager. Used by *BaseModel* and *BModel*. Uses *BQuerySet* as base.

class vstutils.models.queryset.**BQuerySet**(*model=None*, *query=None*, *using=None*, *hints=None*)

Represent a lazy database lookup for a set of objects. Allows to override default iterable class by *custom_iterable_class* attribute (class with *__iter__* method which returns generator of model objects) and default query class by *custom_query_class* attribute (class inherited from `django.db.models.sql.query.Query`).

cleared()

Filter queryset for models with attribute 'hidden' and exclude all hidden objects.

get_paginator(**args*, ***kwargs*)

Returns initialized object of *vstutils.utils.Paginator* over current instance's QuerySet. All args and kwargs go to to Paginator's constructor.

paged(**args*, ***kwargs*)

Returns paginated data with custom Paginator-class. Uses *PAGE_LIMIT* from global settings by default.

class vstutils.models.decorators.**register_view_action**(**args*, ***kwargs*)

Decorator for turning model methods to generated view actions[50]. When a method is decorated, it becomes a part of the generated view and the *self* reference within the method points to the view object. This allows you to extend the functionality of generated views with custom actions.

The *register_view_action* decorator supports various arguments, and you can refer to the documentation for *vstutils.api.decorators.subaction()* to explore the complete list of supported arguments. These arguments provide flexibility in defining the behavior and characteristics of the generated view actions.

Note: In scenarios where you're working with proxy models that share a common set of actions, you can use the *inherit* named argument with a value of *True*. This allows the proxy model to inherit actions defined in the base model, reducing redundancy and promoting code reuse.

Note: In many cases, an action may not require any parameters and can be executed by sending an empty query. To streamline development and enhance efficiency, the *register_view_action* decorator sets the default serializer to *vstutils.api.serializers.EmptySerializer*. This means that the action expects no input data, making it convenient for actions that operate without additional parameters.

Example:

This example demonstrates how to use the decorator to create a custom action within a model view. The `empty_action` method becomes part of the generated view and expects no input parameters.

```
from vstutils.models import BModel
from vstutils.models.decorators import register_view_action
from vstutils.api.responses import HTTP_200_OK


class MyModel(BModel):
    # ... model fields ...

    @register_view_action(detail=False, inherit=True)
    def empty_action(self, request, *args, **kwargs):
        # in this case `self` will be reference within the method points␣
↪to the view object
        return HTTP_200_OK('OK')
```

Vstutils supports models that don't necessitate direct database interaction or aren't inherently tied to database tables. These models exhibit diverse behaviors, such as fetching data directly from class attributes, loading data from files, or implementing custom data retrieval mechanisms. Remarkably, there are models that, in a sense, implement the mechanism of SQL views with pre-defined queries. This flexibility allows developers to define a wide range of models that cater to specific data needs, from in-memory models to those seamlessly integrating external data sources. Vstutils' model system is not confined to traditional database-backed structures, providing a versatile foundation for crafting various data representations.

class vstutils.models.custom_model.**ExternalCustomModel**(**args*, ***kwargs*)

Represents a custom model designed for the self-implementation of requests to external services.

This model facilitates the seamless interaction with external services by allowing the passing of filtering, limiting, and sorting parameters to an external request. It is designed to receive data that is already filtered and limited.

To utilize this model effectively, developers need to implement the `get_data_generator()` class method. This method receives a query object containing the necessary parameters, enabling developers to customize interactions with external services.

Example:

```
class MyExternalModel(ExternalCustomModel):
    # ... model fields ...

    class Meta:
        managed = False

    @classmethod
    def get_data_generator(cls, query):
        data = ... # some fetched data from the external resource or generated
↪from memory calculations.
        for row in data:
            yield row
```

classmethod get_data_generator(*query*)

This class method must be implemented by derived classes to define custom logic for fetching data from an external service based on the provided query parameters.

Query object might contain the following parameters:

- filter (dict): A dictionary specifying the filtering criteria.
- exclude (dict): A dictionary specifying the exclusion criteria.
- order_by (list): A list specifying the sorting order.
- low_mark (int): The low index for slicing (if sliced).
- high_mark (int): The high index for slicing (if sliced).
- is_sliced (bool): A boolean indicating whether the query is sliced.

Parameters
query (*dict*[51]) – An object containing filtering, limiting, and sorting parameters.

Returns
A generator that yields the requested data.

Return type
Generator

[50] https://www.django-rest-framework.org/api-guide/viewsets/#marking-extra-actions-for-routing

Raises

NotImplementedError[52] – If the method is not implemented by the derived class.

class vstutils.models.custom_model.**FileModel**(**args*, ***kwargs*)

Custom model that loads data from a YAML file instead of a database. The path to the file is specified in the *FileModel.file_path* attribute.

Examples:

Suppose the source file is stored at */etc/authors.yaml* with the following content:

```
- name: "Sergey Klyuykov"
- name: "Michael Taran"
```

You can create a custom model using this file:

```
from vstutils.custom_model import FileModel, CharField

class Authors(FileModel):
    name = CharField(max_length=512)

    file_path = '/etc/authors.yaml'
```

class vstutils.models.custom_model.**ListModel**(**args*, ***kwargs*)

Custom model which uses a list of dicts with data (attribute *ListModel.data*) instead of database records. Useful when you have a simple list of data.

Examples:

```
from vstutils.custom_model import ListModel, CharField


class Authors(ListModel):
    name = CharField(max_length=512)

    data = [
        {"name": "Sergey Klyuykov"},
        {"name": "Michael Taran"},
    ]
```

Sometimes, it may be necessary to switch the data source. For these purposes, you should use the *setup_custom_queryset_kwargs* function, which takes various named arguments, which are also passed to the data initialization function. One such argument for *ListModel* is date_source, which takes any iterable object.

Examples:

```
from vstutils.custom_model import ListModel, CharField


class Authors(ListModel):
    name = CharField(max_length=512)

qs = Authors.objects.setup_custom_queryset_kwargs(data_source=[
    {"name": "Sergey Klyuykov"},
    {"name": "Michael Taran"},
])
```

[51] https://docs.python.org/3.8/library/stdtypes.html#dict
[52] https://docs.python.org/3.8/library/exceptions.html#NotImplementedError

In this case, we setup source list via *setup_custom_queryset_kwargs* function, and any other chained call is going to work with this data.

Variables

data (*list*[53]) – List with data dicts. Empty by default.

class vstutils.models.custom_model.**ViewCustomModel**(**args*, ***kwargs*)

Implements the SQL View programming mechanism over other models.

This model provides a mechanism for implementing SQL View-like behavior over other models. In the `get_view_queryset()` method, a base query is prepared, and all further actions are implemented on top of it.

Example Usage:

```
class MyViewModel(ViewCustomModel):
    # ... model fields ...

    class Meta:
        managed = False

    @classmethod
    def get_view_queryset(cls):
        return SomeModel.objects.annotate(...)  # add some additional annotations␣
↪to query
```

classmethod get_view_queryset()

This class method must be implemented by derived classes to define custom logic for generating the base queryset for the SQL View.

Returns

The base queryset for the SQL View.

Return type

django.db.models.query.QuerySet[54]

Raises

NotImplementedError[55] – If the method is not implemented by the derived class.

3.1.1 Model Fields

class vstutils.models.fields.**FkModelField**(*to*, *on_delete*, *related_name=None*, *related_query_name=None*, *limit_choices_to=None*, *parent_link=False*, *to_field=None*, *db_constraint=True*, ***kwargs*)

Extends `django.db.models.ForeignKey`[56]. Use this field in *vstutils.models.BModel* to get *vstutils.api.FkModelField* in serializer. To set Foreign Key relation set *to* argument to string path to model or to Model Class as in `django.db.models.ForeignKey`[57]

class vstutils.models.fields.**HTMLField**(**args*, *db_collation=None*, ***kwargs*)

Extends `django.db.models.TextField`[58]. A simple field for storing HTML markup. The field is based

[53] https://docs.python.org/3.8/library/stdtypes.html#list
[54] https://docs.djangoproject.com/en/4.2/ref/models/querysets/#django.db.models.query.QuerySet
[55] https://docs.python.org/3.8/library/exceptions.html#NotImplementedError
[56] https://docs.djangoproject.com/en/4.2/ref/models/fields/#django.db.models.ForeignKey
[57] https://docs.djangoproject.com/en/4.2/ref/models/fields/#django.db.models.ForeignKey

on `django.db.models.TextField`[59], therefore it does not support indexing and is not recommended for use in filters.

class vstutils.models.fields.**MultipleFieldFile**(*instance*, *field*, *name*)

Subclasses `django.db.models.fields.files.FieldFile`[60]. Provides *`MultipleFieldFile.save()`* and *`MultipleFieldFile.delete()`* to manipulate the underlying file, as well as update the associated model instance.

delete(*save=True*)

Delete file from storage and from object attr.

save(*name*, *content*, *save=True*)

Save changes in file to storage and to object attr.

class vstutils.models.fields.**MultipleFileDescriptor**(*field*)

Subclasses `django.db.models.fields.files.FileDescriptor` to handle list of files. Return a list of *`MultipleFieldFile`* when accessed so you can write code like:

```
from myapp.models import MyModel
instance = MyModel.objects.get(pk=1)
instance.files[0].size
```

get_file(*file*, *instance*)

Always return valid attr_class object. For details on logic see `django.db.models.fields.files.FileDescriptor.__get__()`.

class vstutils.models.fields.**MultipleFileField**(***kwargs*)

Subclasses `django.db.models.fields.files.FileField`. Field for storing a list of Storage-kept files. All args passed to FileField.

attr_class

alias of *`MultipleFieldFile`*

descriptor_class

alias of *`MultipleFileDescriptor`*

class vstutils.models.fields.**MultipleFileMixin**(***kwargs*)

Mixin suited to use with `django.db.models.fields.files.FieldFile`[61] to transform it to a Field with list of files.

get_prep_value(*value*)

Prepare value for database insertion

pre_save(*model_instance*, *add*)

Call .save() method on every file in list

class vstutils.models.fields.**MultipleImageField**(***kwargs*)

Field for storing a list of storage-kept images. All args are passed to `django.db.models.fields.files.ImageField`, except height_field and width_field, they are not currently implemented.

attr_class

alias of *`MultipleImageFieldFile`*

[58] https://docs.djangoproject.com/en/4.2/ref/models/fields/#django.db.models.TextField
[59] https://docs.djangoproject.com/en/4.2/ref/models/fields/#django.db.models.TextField
[60] https://docs.djangoproject.com/en/4.2/ref/models/fields/#django.db.models.fields.files.FieldFile
[61] https://docs.djangoproject.com/en/4.2/ref/models/fields/#django.db.models.fields.files.FieldFile

descriptor_class

alias of *MultipleFileDescriptor*

class vstutils.models.fields.**MultipleImageFieldFile**(*instance*, *field*, *name*)

Subclasses *MultipleFieldFile* and `ImageFile mixin`, handles deleting _dimensions_cache when file is deleted.

class vstutils.models.fields.**MultipleNamedBinaryFileInJSONField**(**args*, *db_collation=None*, ***kwargs*)

Extends `django.db.models.TextField`[62]. Use this field in *vstutils.models.BModel* to get *vstutils.api.MultipleNamedBinaryFileInJSONField* in serializer.

class vstutils.models.fields.**MultipleNamedBinaryImageInJSONField**(**args*, *db_collation=None*, ***kwargs*)

Extends `django.db.models.TextField`[63]. Use this field in *vstutils.models.BModel* to get *vstutils.api.MultipleNamedBinaryImageInJSONField* in serializer.

class vstutils.models.fields.**NamedBinaryFileInJSONField**(**args*, *db_collation=None*, ***kwargs*)

Extends `django.db.models.TextField`[64]. Use this field in *vstutils.models.BModel* to get *vstutils.api.NamedBinaryFileInJSONField* in serializer.

class vstutils.models.fields.**NamedBinaryImageInJSONField**(**args*, *db_collation=None*, ***kwargs*)

Extends `django.db.models.TextField`[65]. Use this field in *vstutils.models.BModel* to get *vstutils.api.NamedBinaryImageInJSONField* in serializer.

class vstutils.models.fields.**WYSIWYGField**(**args*, *db_collation=None*, ***kwargs*)

Extends `django.db.models.TextField`[66]. A simple field for storing Markdown data. The field is based on `django.db.models.TextField`[67], therefore it does not support indexing and is not recommended for use in filters.

3.2 Web API

Web API is based on Django Rest Framework with additional nested functions.

[62] https://docs.djangoproject.com/en/4.2/ref/models/fields/#django.db.models.TextField
[63] https://docs.djangoproject.com/en/4.2/ref/models/fields/#django.db.models.TextField
[64] https://docs.djangoproject.com/en/4.2/ref/models/fields/#django.db.models.TextField
[65] https://docs.djangoproject.com/en/4.2/ref/models/fields/#django.db.models.TextField
[66] https://docs.djangoproject.com/en/4.2/ref/models/fields/#django.db.models.TextField
[67] https://docs.djangoproject.com/en/4.2/ref/models/fields/#django.db.models.TextField

3.2.1 Fields

The Framework includes a list of convenient serializer fields. Some of them take effect only in generated admin interface.

Additional serializer fields for generating OpenAPI and GUI.

class vstutils.api.fields.**AutoCompletionField**(**args*, ***kwargs*)

Serializer field that provides autocompletion on the frontend, using a specified list of objects.

Parameters

- **autocomplete** (*list*[68], *tuple*[69], *str*[70]) – Autocompletion reference. You can set a list or tuple with values or specify the name of an OpenAPI schema definition. For a definition name, the GUI will find the optimal link and display values based on the `autocomplete_property` and `autocomplete_represent` arguments.
- **autocomplete_property** (*str*[71]) – Specifies which attribute to retrieve from the OpenAPI schema definition model as the value. Default is 'id'.
- **autocomplete_represent** – Specifies which attribute to retrieve from the OpenAPI schema definition model as the representational value. Default is 'name'.
- **use_prefetch** (*bool*[72]) – Prefetch values on the frontend in list view. Default is `True`.

Note: This functionality is effective only in the GUI. In the API, it behaves similarly to *VSTCharField*.

Usage:

This field is designed to be used in serializers where a user needs to input a value, and autocompletion based on a predefined list or an OpenAPI schema definition is desired. If an OpenAPI schema is specified, two additional parameters, `autocomplete_property` and `autocomplete_represent`, can be configured to customize the appearance of the dropdown list.

Example:

```
from vstutils.api import serializers
from vstutils.api.fields import AutoCompletionField


class MyModelSerializer(serializers.BaseSerializer):
    name = AutoCompletionField(autocomplete=['Option 1', 'Option 2',
↪'Option 3'])

# or

class MyModelSerializer(serializers.BaseSerializer):
    name = AutoCompletionField(
        autocomplete='MyModelSchema',
        autocomplete_property='custom_property',
        autocomplete_represent='display_name'
    )
```

[68] https://docs.python.org/3.8/library/stdtypes.html#list
[69] https://docs.python.org/3.8/library/stdtypes.html#tuple
[70] https://docs.python.org/3.8/library/stdtypes.html#str
[71] https://docs.python.org/3.8/library/stdtypes.html#str
[72] https://docs.python.org/3.8/library/functions.html#bool

class vstutils.api.fields.**Barcode128Field**(**args*, ***kwargs*)

A field for representing data as a Barcode (Code 128) in the user interface.

This field accepts and validates data in the form of a valid ASCII string. It is designed to display the data as a Code 128 barcode in the graphical user interface. The underlying data is serialized or deserialized using the specified child field.

Parameters

child (*rest_framework.fields.Field*) – The original data field for serialization or deserialization. Default: `rest_framework.fields.CharField`

Example:

Suppose you have a model with a *product_code* field, and you want to display its Code 128 barcode representation in the GUI. You can use *Barcode128Field* in your serializer:

```
class Product(BModel):
    product_code = models.CharField(max_length=20)

class ProductSerializer(VSTSerializer):
    barcode = Barcode128Field(child=serializers.CharField(source='product_code
↪'))

    class Meta:
        model = Product
        fields = '__all__'
```

class vstutils.api.fields.**BinFileInStringField**(**args*, ***kwargs*)

This field extends *FileInStringField* and is specifically designed to handle binary files. In the GUI, it functions as a file input field, accepting binary files from the user, which are then converted to base64-encoded strings and stored in this field.

Parameters

media_types (*tuple*[73], *list*[74]) – A list of MIME types to filter on the user's side. Supports the use of `*` as a wildcard. Default: *['*/*']*

Note: This functionality is effective only in the GUI. In the API, it behaves similarly to *VSTCharField*.

---

class vstutils.api.fields.**CSVFileField**(**args*, ***kwargs*)

Field extends *FileInStringField*, using for works with csv files. This field provides the display of the loaded data in the form of a table.

Parameters

- **items** (*Serializer*) – The config of the table. This is a drf or vst serializer which includes char fields which are the keys in the dictionaries into which the data from csv is serialized and the names for columns in a table. The fields must be in the order you want them to appear in the table. Following options may be included:
  - `label`: human readable column name
  - `required`: Defines whether the field should be required. False by default.
- **min_column_width** (*int*[75]) – Minimum cell width. Default is 200 px.
- **delimiter** (*str*[76]) – The delimiting character.

[73] https://docs.python.org/3.8/library/stdtypes.html#tuple
[74] https://docs.python.org/3.8/library/stdtypes.html#list

- **lineterminator** (*str*[77]) – The newline sequence. Leave blank to auto-detect. Must be one of `\r`, `\n`, or `\r\n`.
- **quotechar** (*str*[78]) – The character used to quote fields.
- **escapechar** (*str*[79]) – The character used to escape the quote character within a field.
- **media_types** (*tuple*[80], *list*[81]) – List of MIME types to select on the user's side. Supported syntax using `*`. Default: `['text/csv']`

class vstutils.api.fields.**CommaMultiSelect**(**args*, ***kwargs*)

Field that allows users to input multiple values, separated by a specified delimiter (default: ","). It retrieves a list of values from another model or a custom list and provides autocompletion similar to *AutoCompletionField*. This field is suitable for property fields in a model where the main logic is already implemented or for use with `model.CharField`.

Parameters

- **select** (*str*[82], *tuple*[83], *list*[84]) – OpenAPI schema definition name or a list with values.
- **select_separator** (*str*[85]) – The separator for values. The default is a comma.
- **select_represent** (*select_property,*) – These parameters function similarly to `autocomplete_property` and `autocomplete_represent`. The default is `name`.
- **use_prefetch** (*bool*[86]) – Prefetch values on the frontend in list view. The default is `False`.
- **make_link** (*bool*[87]) – Show values as links to the model. The default is `True`.
- **dependence** (*dict*[88]) – A dictionary where keys are the names of fields from the same model, and values are the names of query filters. If at least one of the fields we depend on is non-nullable, required, and set to null, the autocompletion list will be empty, and the field will be disabled.

Example:

```
from vstutils.api import serializers
from vstutils.api.fields import CommaMultiSelect

class MyModelSerializer(serializers.BaseSerializer):
    tags = CommaMultiSelect(
        select="TagsReferenceSchema",
        select_property='slug',
        select_represent='slug',
        use_prefetch=True,
        make_link=False,
        dependence={'some_field': 'value'},
    )

# or

class MyModelSerializer(serializers.BaseSerializer):
    tags = CommaMultiSelect(select=['tag1', 'tag2', 'tag3'])
```

[75] https://docs.python.org/3.8/library/functions.html#int
[76] https://docs.python.org/3.8/library/stdtypes.html#str
[77] https://docs.python.org/3.8/library/stdtypes.html#str
[78] https://docs.python.org/3.8/library/stdtypes.html#str
[79] https://docs.python.org/3.8/library/stdtypes.html#str
[80] https://docs.python.org/3.8/library/stdtypes.html#tuple
[81] https://docs.python.org/3.8/library/stdtypes.html#list

Note: This functionality is effective only in the GUI and works similarly to *VSTCharField* in the API.

class vstutils.api.fields.**CrontabField**(**args*, ***kwargs*)

Simple crontab-like field which contains the schedule of cron entries to specify time. A crontab field has five fields for specifying day, date and time. `*` in the value field above means all legal values as in braces for that column.

The value column can have a `*` or a list of elements separated by commas. An element is either a number in the ranges shown above or two numbers in the range separated by a hyphen (meaning an inclusive range).

The time and date fields are:

field	allowed value
minute	0-59
hour	0-23
day of month	1-31
month	1-12
day of week	0-7 (0 or 7 is Sunday)

Default value of each field if not specified is `*`.

```
.---------------- minute (0 - 59)
| .-------------- hour (0 - 23)
| | .------------ day of month (1 - 31)
| | | .---------- month (1 - 12)
| | | | .-------- day of week (0 - 6) (Sunday=0 or 7)
| | | | |
* * * * *
```

class vstutils.api.fields.**DeepFkField**(*only_last_child=False*, *parent_field_name='parent'*, ***kwargs*)

Extends *FkModelField*, specifically designed for hierarchical relationships in the frontend.

This field is intended for handling ForeignKey relationships within a hierarchical or tree-like structure. It displays as a tree in the frontend, offering a clear visual representation of parent-child relationships.

Warning: This field intentionally does not support the `dependence` parameter, as it operates in a tree structure. Usage of `filters` should be approached with caution, as inappropriate filters may disrupt the tree hierarchy.

Parameters

- **only_last_child** (*bool*[89]) – If True, the field restricts the selection to nodes without children. Default is *False*. Useful when you want to enforce selection of leaf nodes.
- **parent_field_name** (*str*[90]) – The name of the parent field in the related model. Default is *parent*. Should be set to the ForeignKey field in the related model, representing the

[82] https://docs.python.org/3.8/library/stdtypes.html#str
[83] https://docs.python.org/3.8/library/stdtypes.html#tuple
[84] https://docs.python.org/3.8/library/stdtypes.html#list
[85] https://docs.python.org/3.8/library/stdtypes.html#str
[86] https://docs.python.org/3.8/library/functions.html#bool
[87] https://docs.python.org/3.8/library/functions.html#bool
[88] https://docs.python.org/3.8/library/stdtypes.html#dict

parent-child relationship. For example, if your related model has a ForeignKey like *parent = models.ForeignKey('self', …)*, then *parent_field_name* should be set to *'parent'*.

Examples:

Consider a related model with a ForeignKey field representing parent-child relationships:

```
class Category(BModel):
    name = models.CharField(max_length=255)
    parent = models.ForeignKey('self', null=True, default=None, on_
↪delete=models.CASCADE)
```

To use the DeepFkField with this related model in a serializer, you would set the parent_field_name to 'parent':

```
class MySerializer(VSTSerializer):
    category = DeepFkField(select=Category, parent_field_name='parent')
```

This example assumes a Category related model with a ForeignKey 'parent' field. The DeepFkField will then display the categories as a tree structure in the frontend, providing an intuitive selection mechanism for hierarchical relationships.

Note: Effective only in GUI. Works similarly to `rest_framework.fields.IntegerField` in API.

---

class vstutils.api.fields.**DependEnumField**(**args*, ***kwargs*)

Field extends *DynamicJsonTypeField* but its value is not transformed to json and would be given as is. Useful for `property`[91] in models or for actions.

Parameters

- **field** (*str*[92]) – field in model which value change will change type of current value.
- **types** – key-value mapping where key is value of subscribed field and value is type (in OpenAPI format) of current field.
- **choices** (*dict*[93]) – variants of choices for different subscribed field values. Uses mapping where key is value of subscribed field and value is list with values to choice.

Note: Effective only in GUI. In API works similar to *VSTCharField* without value modification.

---

class vstutils.api.fields.**DependFromFkField**(**args*, ***kwargs*)

A specialized field that extends *DynamicJsonTypeField* and validates field data based on a `field_attribute` chosen in a related model. The field data is validated against the type defined by the chosen value of `field_attribute`.

Note: By default, any value of `field_attribute` validates as *VSTCharField*. To override this behavior, set the class attribute `{field_attribute}_fields_mapping` in the related model. The attribute should be a dictionary where keys are string representations of the values of `field_attribute`, and values are instances of `rest_framework.Field` for validation. If a value is not found in the dictionary, the default type will be *VSTCharField*.

---

[89] https://docs.python.org/3.8/library/functions.html#bool
[90] https://docs.python.org/3.8/library/stdtypes.html#str
[91] https://docs.python.org/3.8/library/functions.html#property
[92] https://docs.python.org/3.8/library/stdtypes.html#str
[93] https://docs.python.org/3.8/library/stdtypes.html#dict

Parameters

- **field** (*str*[94]) – The field in the model whose value change determines the type of the current value. The field must be of type *FkModelField*.
- **field_attribute** (*str*[95]) – The attribute of the related model instance containing the name of the type.
- **types** (*dict*[96]) – A key-value mapping where the key is the value of the subscribed field, and the value is the type (in OpenAPI format) of the current field.

Warning: The `field_attribute` in the related model must be of type `rest_framework.fields.ChoicesField` to ensure proper functioning in the GUI; otherwise, the field will be displayed as simple text.

Example:

Suppose you have a model with a ForeignKey field *related_model* and a field *type_attribute* in *RelatedModel* that determines the type of data. You can use *DependFromFkField* to dynamically adapt the serialization based on the value of *type_attribute*:

```
class RelatedModel(BModel):
    # ... other fields ...
    type_attribute = models.CharField(max_length=20, choices=[('type1', 'Type␣
↪1'), ('type2', 'Type 2')])

    type_attribute_fields_mapping = {
        'type1': SomeSerializer(),
        'type2': IntegerField(max_value=1337),
    }

class MyModel(BModel):
    related_model = models.ForeignKey(RelatedModel, on_delete=models.CASCADE)

class MySerializer(VSTSerializer):
    dynamic_field = DependFromFkField(
        field='related_model',
        field_attribute='type_attribute'
    )

    class Meta:
        model = MyModel
        fields = '__all__'
```

class vstutils.api.fields.**DynamicJsonTypeField**(**args*, ***kwargs*)

A versatile serializer field that dynamically adapts its type based on the value of another field in the model. It facilitates complex scenarios where the type of data to be serialized depends on the value of a related field.

Parameters

- **field** (*str*[97]) – The field in the model whose value change will dynamically determine the type of the current field.

[94] https://docs.python.org/3.8/library/stdtypes.html#str
[95] https://docs.python.org/3.8/library/stdtypes.html#str
[96] https://docs.python.org/3.8/library/stdtypes.html#dict

- **types** (*dict*[98]) – A key-value mapping where the key is the value of the subscribed field, and the value is the type (in OpenAPI format) of the current field.
- **choices** (*dict*[99]) – Variants of choices for different subscribed field values. Uses a mapping where the key is the value of the subscribed field, and the value is a list with values to choose from.
- **source_view** (*str*[100]) – Allows using parent views data as a source for field creation. The exact view path (*/user/{id}/*) or relative parent specifier (*<<parent>>.<<parent>>.<<parent>>*) can be provided. For example, if the current page is */user/1/role/2/* and *source_view* is *<<parent>>.<<parent>>*, then data from */user/1/* will be used. Only detail views are supported.

Example:

Suppose you have a serializer *MySerializer* with a *field_type* (e.g., a *ChoiceField*) and a corresponding *object_data*. The *object_data* field can have different types based on the value of *field_type*. Here's an example configuration:

```
class MySerializer(VSTSerializer):
    field_type = serializers.ChoiceField(choices=['serializer', 'integer',
↪'boolean'])
    object_data = DynamicJsonTypeField(
        field='field_type',
        types={
            'serializer': SomeSerializer(),
            'integer': IntegerField(max_value=1337),
            'boolean': 'boolean',
        },
    )
```

In this example, the *object_data* field dynamically adapts its type based on the selected value of *field_type*. The *types* argument defines different types for each possible value of *field_type*, allowing for flexible and dynamic serialization.

class vstutils.api.fields.**FileInStringField**(**args*, ***kwargs*)

This field extends *VSTCharField* and stores the content of a file as a string.

The value must be text (not binary) and is saved in the model as is.

Parameters

media_types (*tuple*[101], *list*[102]) – A list of MIME types to filter on the user's side. Supports the use of `*` as a wildcard. Default: `['*/*']`

Note: This setting only takes effect in the GUI. In the API, it behaves like *VSTCharField*.

---

class vstutils.api.fields.**FkField**(**args*, ***kwargs*)

An implementation of ForeignKeyField, designed for use in serializers. This field allows you to specify which field of a related model will be stored in the field (default: "id"), and which field will represent the value on the frontend.

Parameters

[97] https://docs.python.org/3.8/library/stdtypes.html#str
[98] https://docs.python.org/3.8/library/stdtypes.html#dict
[99] https://docs.python.org/3.8/library/stdtypes.html#dict
[100] https://docs.python.org/3.8/library/stdtypes.html#str
[101] https://docs.python.org/3.8/library/stdtypes.html#tuple
[102] https://docs.python.org/3.8/library/stdtypes.html#list

- **select** (*str*[103]) – OpenAPI schema definition name.
- **autocomplete_property** (*str*[104]) – Specifies which attribute will be retrieved from the OpenAPI schema definition model as the value. Default is `id`.
- **autocomplete_represent** – Specifies which attribute will be retrieved from the OpenAPI schema definition model as the representational value. Default is `name`.
- **field_type** (*type*[105]) – Defines the type of the autocomplete_property for further definition in the schema and casting to the type from the API. Default is passthrough but requires *int* or *str* objects.
- **use_prefetch** (*bool*[106]) – Prefetch values on the frontend at list-view. Default is `True`.
- **make_link** (*bool*[107]) – Show the value as a link to the related model. Default is `True`.
- **dependence** (*dict*[108]) – A dictionary where keys are names of fields from the same model, and values are names of query filters. If at least one of the fields that we depend on is non-nullable, required, and set to null, the autocompletion list will be empty, and the field will be disabled.

  There are some special keys for the dependence dictionary to get data that is stored on the frontend without additional database query:

  - `'<<pk>>'` gets the primary key of the current instance,
  - `'<<view_name>>'` gets the view name from the Vue component,
  - `'<<parent_view_name>>'` gets the parent view name from the Vue component,
  - `'<<view_level>>'` gets the view level,
  - `'<<operation_id>>'` gets the operation_id,
  - `'<<parent_operation_id'>>` gets the parent_operation_id.

Examples:

```
field = FkField(select=Category, dependence={'<<pk>>': 'my_filter'})
```

This filter will get the primary key of the current object and make a query on the frontend `/category?my_filter=3` where `3` is the primary key of the current instance.

Parameters

filters (*dict*[109]) – A dictionary where keys are names of fields from a related model (specified by this FkField), and values are values of that field.

Note: The intersection of *dependence.values()* and *filters.keys()* will throw an error to prevent ambiguous filtering.

Note: Effective only in the GUI. Works similarly to `rest_framework.fields.IntegerField` in the API.

---

[103] https://docs.python.org/3.8/library/stdtypes.html#str
[104] https://docs.python.org/3.8/library/stdtypes.html#str
[105] https://docs.python.org/3.8/library/functions.html#type
[106] https://docs.python.org/3.8/library/functions.html#bool
[107] https://docs.python.org/3.8/library/functions.html#bool
[108] https://docs.python.org/3.8/library/stdtypes.html#dict
[109] https://docs.python.org/3.8/library/stdtypes.html#dict

class vstutils.api.fields.**FkModelField**(**args*, ***kwargs*)

Extends *FkField*, but stores referred model class. This field is useful for django.db.models.ForeignKey[110] fields in model to set.

Parameters

- **select** (vstutils.models.BModel, vstutils.api.serializers.VSTSerializer) – model class (based on *vstutils.models.BModel*) or serializer class which used in API and has path in OpenAPI schema.
- **autocomplete_property** (*str*[111]) – this argument indicates which attribute will be get from OpenAPI schema definition model as value. Default is `id`.
- **autocomplete_represent** – this argument indicates which attribute will be get from OpenAPI schema definition model as represent value. Default is `name`.
- **use_prefetch** – prefetch values on frontend at list-view. Default is `True`.
- **make_link** – Show value as link to model. Default is `True`.

Warning: Model class get object from database during *.to_internal_value* execution. Be careful on mass save executions.

Warning: Permissions to model which is referred by this field, are not to be checked. You should check it manually in signals or validators.

class vstutils.api.fields.**HtmlField**(**args*, ***kwargs*)

A specialized field for handling HTML text content, marked with the format:html.

Note: This field is designed for use in the graphical user interface (GUI) and functions similarly to *VSTCharField* in the API.

Example:

If you have a model with an *html_content* field that stores HTML-formatted text, you can use *HtmlField* in your serializer to handle this content in the GUI:

```
class MyModel(BModel):
    html_content = models.TextField()

class MySerializer(VSTSerializer):
    formatted_html_content = HtmlField(source='html_content')

    class Meta:
        model = MyModel
        fields = '__all__'
```

class vstutils.api.fields.**MaskedField**(**args*, ***kwargs*)

Extends the 'rest_framework.serializers.CharField' class. Field that applies a mask to the input value.

This field is designed for applying a mask to the input value on the frontend. It extends the 'rest_framework.serializers.CharField' and allows the use of the IMask[112] library for defining masks.

[110] https://docs.djangoproject.com/en/4.2/ref/models/fields/#django.db.models.ForeignKey
[111] https://docs.python.org/3.8/library/stdtypes.html#str

Parameters

mask (*dict*[113], *str*[114]) – The mask to be applied to the value. It can be either a dictionary or a string following the *IMask* library format.

Example:

In a serializer, include this field to apply a mask to a value:

```
class MySerializer(serializers.Serializer):
    masked_value = MaskedField(mask='000-000')
```

This example assumes a serializer where the `masked_value` field represents a value with a predefined mask. The `MaskedField` will apply the specified mask on the frontend, providing a masked input for users.

Note: The effectiveness of this field is limited to the frontend, and the mask is applied during user input.

---

class vstutils.api.fields.**MultipleNamedBinaryFileInJsonField**(**args*, ***kwargs*)

Extends *NamedBinaryFileInJsonField* but uses list of JSONs. Allows to operate with multiple files as list of *NamedBinaryFileInJsonField*.

Attrs: `NamedBinaryInJsonField.file`: if True, accept only subclasses of File as input. If False, accept only string input. Default: False.

file_field

alias of *MultipleFieldFile*

class vstutils.api.fields.**MultipleNamedBinaryImageInJsonField**(**args*, ***kwargs*)

Extends *MultipleNamedBinaryFileInJsonField* but uses list of JSONs. Used for operating with multiple images and works as list of *NamedBinaryImageInJsonField*.

Parameters

background_fill_color (*str*[115]) – Color to fill area that is not covered by image after cropping. Transparent by default but will be black if image format is not supporting transparency. Can be any valid CSS color.

class vstutils.api.fields.**NamedBinaryFileInJsonField**(**args*, ***kwargs*)

Field that represents a binary file in JSON format.

Parameters

- **file** (*bool*[116]) – If True, accept only subclasses of File as input. If False, accept only string input. Default: False.
- **post_handlers** (*tuple*[117], *list*[118]) – Functions to process the file after validation. Each function takes two arguments: `binary_data` (file bytes) and `original_data` (reference to the original JSON object). The function should return the processed `binary_data`.
- **pre_handlers** (*tuple*[119], *list*[120]) – Functions to process the file before validation. Each function takes two arguments: `binary_data` (file bytes) and `original_data` (reference to the original JSON object). The function should return the processed `binary_data`.

[112] https://imask.js.org/guide.html
[113] https://docs.python.org/3.8/library/stdtypes.html#dict
[114] https://docs.python.org/3.8/library/stdtypes.html#str
[115] https://docs.python.org/3.8/library/stdtypes.html#str

- **max_content_size** (*int*[121]) – Maximum allowed size of the file content in bytes.
- **min_content_size** (*int*[122]) – Minimum allowed size of the file content in bytes.
- **min_length** (*int*[123]) – Minimum length of the file name. Only applicable when `file=True`.
- **max_length** (*int*[124]) – Maximum length of the file name. Only applicable when `file=True`.

This field is designed for storing binary files alongside their names in `django.db.models.CharField`[125] or `django.db.models.TextField`[126] model fields. All manipulations involving decoding and encoding binary content data occur on the client, imposing reasonable limits on file size.

Additionally, this field can construct a `django.core.files.uploadedfile.SimpleUploadedFile` from incoming JSON and store it as a file in `django.db.models.FileField`[127] if the *file* attribute is set to *True*.

Example:

In a serializer, include this field to handle binary files:

```
class MySerializer(serializers.ModelSerializer):
    binary_data = NamedBinaryFileInJsonField(file=True)
```

This example assumes a serializer where the `binary_data` field represents binary file information in JSON format. The `NamedBinaryFileInJsonField` will then handle the storage and retrieval of binary files in a user-friendly manner.

The binary file is represented in JSON with the following properties:

- **name** (str): Name of the file.
- **mediaType** (str): MIME type of the file.
- **content** (str or File): Content of the file. If *file* is True, it will be a reference to the file; otherwise, it will be base64-encoded content.

Warning: The client application will display the content as a download link. Users will interact with the binary file through the application, with the exchange between the Rest API and the client occurring through the presented JSON object.

class vstutils.api.fields.**NamedBinaryImageInJsonField**(**args*, ***kwargs*)

Field that represents an image in JSON format, extending *NamedBinaryFileInJsonField*.

Parameters

background_fill_color (*str*[128]) – Color to fill the area not covered by the image after

[116] https://docs.python.org/3.8/library/functions.html#bool
[117] https://docs.python.org/3.8/library/stdtypes.html#tuple
[118] https://docs.python.org/3.8/library/stdtypes.html#list
[119] https://docs.python.org/3.8/library/stdtypes.html#tuple
[120] https://docs.python.org/3.8/library/stdtypes.html#list
[121] https://docs.python.org/3.8/library/functions.html#int
[122] https://docs.python.org/3.8/library/functions.html#int
[123] https://docs.python.org/3.8/library/functions.html#int
[124] https://docs.python.org/3.8/library/functions.html#int
[125] https://docs.djangoproject.com/en/4.2/ref/models/fields/#django.db.models.CharField
[126] https://docs.djangoproject.com/en/4.2/ref/models/fields/#django.db.models.TextField
[127] https://docs.djangoproject.com/en/4.2/ref/models/fields/#django.db.models.FileField

cropping. Transparent by default but will be black if the image format does not support transparency. Can be any valid CSS color.

This field is designed for handling image files alongside their names in `django.db.models.CharField`[129] or `django.db.models.TextField`[130] model fields. It extends the capabilities of *`NamedBinaryFileInJsonField`* to specifically handle images.

Additionally, this field validates the image using the following validators: - *`vstutils.api.validators.ImageValidator`* - *`vstutils.api.validators.ImageResolutionValidator`* - *`vstutils.api.validators.ImageHeightValidator`*

When saving and with the added validators, the field will display a corresponding window for adjusting the image to the specified parameters, providing a user-friendly interface for managing images.

The image file is represented in JSON with the following properties:

- **name** (str): Name of the image file.
- **mediaType** (str): MIME type of the image file.
- **content** (str or File): Content of the image file. If *file* is True, it will be a reference to the image file; otherwise, it will be base64-encoded content.

Warning: The client application will display the content as an image. Users will interact with the image through the application, with the exchange between the Rest API and the client occurring through the presented JSON object.

class vstutils.api.fields.**PasswordField**(**args*, ***kwargs*)

Extends CharField[131] but in schema set format to *password*. Show all characters as asterisks instead of real value in GUI.

class vstutils.api.fields.**PhoneField**(**args*, ***kwargs*)

Extends the 'rest_framework.serializers.CharField' class. Field for representing a phone number in international format.

This field is designed for capturing and validating phone numbers in international format. It extends the 'rest_framework.serializers.CharField' and adds custom validation to ensure that the phone number contains only digits.

Example:

In a serializer, include this field to handle phone numbers:

```
class MySerializer(VSTSerializer):
    phone_number = PhoneField()
```

This example assumes a serializer where the `phone_number` field represents a phone number in international format. The `PhoneField` will then handle the validation and representation of phone numbers, making it convenient for users to input standardized phone numbers.

The field will be displayed in the client application as an input field for entering a phone number, including the country code.

[128] https://docs.python.org/3.8/library/stdtypes.html#str
[129] https://docs.djangoproject.com/en/4.2/ref/models/fields/#django.db.models.CharField
[130] https://docs.djangoproject.com/en/4.2/ref/models/fields/#django.db.models.TextField
[131] https://www.django-rest-framework.org/api-guide/fields/#charfield

class vstutils.api.fields.**QrCodeField**(**args*, ***kwargs*)

A versatile field for encoding various types of data into QR codes.

This field can encode a wide range of data into a QR code representation, making it useful for displaying QR codes in the user interface. It works by serializing or deserializing data using the specified child field.

Parameters

child (*rest_framework.fields.Field*) – The original data field for serialization or deserialization. Default: `rest_framework.fields.CharField`

Example:

Suppose you have a model with a *data* field, and you want to display its QR code representation in the GUI. You can use *QrCodeField* in your serializer:

```
class MyModel(BModel):
    data = models.CharField(max_length=255)

class MySerializer(VSTSerializer):
    qr_code_data = QrCodeField(child=serializers.CharField(source='data'))

    class Meta:
        model = MyModel
        fields = '__all__'
```

In this example, the qr_code_data field will represent the QR code generated from the data field in the GUI. Users can interact with this QR code, and if their device supports it, they can scan the code for further actions.

class vstutils.api.fields.**RatingField**(*min_value=0*, *max_value=5*, *step=1*, *front_style='stars'*, ***kwargs*)

Extends class 'rest_framework.serializers.FloatField'. This field represents a rating form input on frontend. Grading limits can be specified with 'min_value=' and 'max_value=', defaults are 0 to 5. Minimal step between grades are specified in 'step=', default - 1. Frontend visual representation can be chosen with 'front_style=', available variants are listed in 'self.valid_front_styles'.

For 'slider' front style, you can specify slider color, by passing valid color to 'color='. For 'fa_icon' front style, you can specify FontAwesome icon that would be used for displaying rating, by passing a valid FontAwesome icon code to 'fa_class='.

Parameters

- **min_value** (*float*[132], *int*[133]) – minimal level
- **max_value** (*float*[134], *int*[135]) – maximal level
- **step** (*float*[136], *int*[137]) – minimal step between levels
- **front_style** (*str*[138]) – visualization on frontend field. Allowed: ['stars', 'slider', 'fa_icon'].
- **color** (*str*[139]) – color of rating element (star, icon or slider) in css format
- **fa_class** (*str*[140]) – FontAwesome icon code

class vstutils.api.fields.**RedirectCharField**(**args*, ***kwargs*)

Field for redirect by string. Often used in actions for redirect after execution.

Note: Effective only in GUI. Works similar to `rest_framework.fields.IntegerField` in API.

---

class vstutils.api.fields.**RedirectFieldMixin**(***kwargs*)

Field mixin indicates that this field is used to send redirect address to frontend after some action.

Parameters

- **operation_name** (*str*[141]) – prefix for operation_id, for example if operation_id is *history_get* then operation_name is *history*
- **depend_field** (*str*[142]) – name of the field that we depend on, its value will be used for operation_id
- **concat_field_name** (*bool*[143]) – if True then name of the field will be added at the end of operation_id

class vstutils.api.fields.**RedirectIntegerField**(**args*, ***kwargs*)

Field for redirect by id. Often used in actions for redirect after execution.

Note: Effective only in GUI. Works similar to `rest_framework.fields.IntegerField` in API.

---

class vstutils.api.fields.**RelatedListField**(*related_name*, *fields*, *view_type='list'*, *serializer_class=None*, ***kwargs*)

Extends *VSTCharField* to represent a reverse ForeignKey relation as a list of related instances.

This field allows you to output the reverse ForeignKey relation as a list of related instances. To use it, specify the `related_name` kwarg (related manager for reverse ForeignKey) and the `fields` kwarg (list or tuple of fields from the related model to be included).

By default, *VSTCharField* is used to serialize all field values and represent them on the frontend. You can specify the *serializer_class* and override fields as needed. For example, title, description, and other field properties can be set to customize frontend behavior.

Parameters

- **related_name** (*str*[144]) – Name of a related manager for reverse ForeignKey.
- **fields** (*list*[145]*[str*[146]*]*, *tuple*[147]*[str*[148]*]*) – List of related model fields.
- **view_type** (*str*[149]) – Determines how fields are represented on the frontend. Must be either `list` or `table`.

[132] https://docs.python.org/3.8/library/functions.html#float
[133] https://docs.python.org/3.8/library/functions.html#int
[134] https://docs.python.org/3.8/library/functions.html#float
[135] https://docs.python.org/3.8/library/functions.html#int
[136] https://docs.python.org/3.8/library/functions.html#float
[137] https://docs.python.org/3.8/library/functions.html#int
[138] https://docs.python.org/3.8/library/stdtypes.html#str
[139] https://docs.python.org/3.8/library/stdtypes.html#str
[140] https://docs.python.org/3.8/library/stdtypes.html#str
[141] https://docs.python.org/3.8/library/stdtypes.html#str
[142] https://docs.python.org/3.8/library/stdtypes.html#str
[143] https://docs.python.org/3.8/library/functions.html#bool

- **fields_custom_handlers_mapping** (*dict*[150]) – Custom handlers mapping, where key: field_name, value: callable_obj that takes params: instance[dict], fields_mapping[dict], model, field_name[str].
- **serializer_class** (*type*[151]) – Serializer to customize types of fields. If no serializer is provided, *VSTCharField* will be used for every field in the *fields* list.

class vstutils.api.fields.**SecretFileInString**(**args*, ***kwargs*)

This field extends *FileInStringField* but hides its value in the admin interface.

The value must be text (not binary) and is saved in the model as is.

Parameters

media_types (*tuple*[152], *list*[153]) – A list of MIME types to filter on the user's side. Supports the use of `*` as a wildcard. Default: `['*/*']`

Note: This setting only takes effect in the GUI. In the API, it behaves like *VSTCharField*.

---

class vstutils.api.fields.**TextareaField**(**args*, ***kwargs*)

A specialized field that allows the input and display of multiline text.

Note: This field is designed for use in the graphical user interface (GUI) and functions similarly to *VSTCharField* in the API.

Example:

Suppose you have a model with a *description* field that can contain multiline text. You can use *TextareaField* in your serializer to enable users to input and view multiline text in the GUI:

```
class MyModel(BModel):
    description = models.TextField()

class MySerializer(VSTSerializer):
    multiline_description = TextareaField(source='description')

    class Meta:
        model = MyModel
        fields = '__all__'
```

class vstutils.api.fields.**UptimeField**(**args*, ***kwargs*)

Time duration field, in seconds, specifically designed for computing and displaying system uptime.

This field is effective only in the GUI and behaves similarly to `rest_framework.fields.IntegerField` in the API.

The *UptimeField* class transforms time in seconds into a user-friendly representation on the frontend. It intelligently selects the most appropriate pattern from the following templates:

[144] https://docs.python.org/3.8/library/stdtypes.html#str
[145] https://docs.python.org/3.8/library/stdtypes.html#list
[146] https://docs.python.org/3.8/library/stdtypes.html#str
[147] https://docs.python.org/3.8/library/stdtypes.html#tuple
[148] https://docs.python.org/3.8/library/stdtypes.html#str
[149] https://docs.python.org/3.8/library/stdtypes.html#str
[150] https://docs.python.org/3.8/library/stdtypes.html#dict
[151] https://docs.python.org/3.8/library/functions.html#type
[152] https://docs.python.org/3.8/library/stdtypes.html#tuple
[153] https://docs.python.org/3.8/library/stdtypes.html#list

- `HH:mm:ss` (e.g., 23:59:59)
- `dd HH:mm:ss` (e.g., 01d 00:00:00)
- `mm dd HH:mm:ss` (e.g., 01m 30d 00:00:00)
- `yy mm dd HH:mm:ss` (e.g., 99y 11m 30d 22:23:24)

Example:

```
class MySerializer(serializers.ModelSerializer):
    uptime = UptimeField()
```

This example assumes a serializer where the *uptime* field represents a time duration in seconds. The *UptimeField* will then display the duration in a human-readable format on the frontend, making it convenient for users to interpret and input values.

Note: Effective only in GUI. Works similarly to `rest_framework.fields.IntegerField` in API.

---

class vstutils.api.fields.**VSTCharField**(**args*, ***kwargs*)

CharField (extends `rest_framework.fields.CharField`). This field translate any json type to string for model.

class vstutils.api.fields.**WYSIWYGField**(**args*, ***kwargs*)

Extends the *TextareaField* class to render the https://ui.toast.com/tui-editor on the frontend.

This field is specifically designed for rendering a WYSIWYG editor on the frontend, using the https://ui.toast.com/tui-editor. It saves data as markdown and escapes all HTML tags.

Parameters

escape (*bool*[154]) – HTML-escape input. Enabled by default to prevent HTML injection vulnerabilities.

Example:

In a serializer, include this field to render a WYSIWYG editor on the frontend:

```
class MySerializer(serializers.Serializer):
    wysiwyg_content = WYSIWYGField()
```

This example assumes a serializer where the `wysiwyg_content` field represents data to be rendered in a WYSIWYG editor on the frontend. The `WYSIWYGField` ensures that the content is saved as markdown and HTML tags are escaped to enhance security.

Warning: Enabling the `escape` option is recommended to prevent potential HTML injection vulnerabilities.

Note: The rendering on the frontend is achieved using the https://ui.toast.com/tui-editor.

---

[154] https://docs.python.org/3.8/library/functions.html#bool

3.2.2 Validators

There are validation classes for fields.

class vstutils.api.validators.**FileMediaTypeValidator**(*extensions=None*, ***kwargs*)

Base Image Validation class. Validates media types.

Parameters

extensions (typing.Union[155][typing.Tuple[156], typing.List[157], None[158]]) – Tuple or List of file extensions, that should pass the validation

Raises rest_framework.exceptions.ValidationError: in case file extension are not in the list

class vstutils.api.validators.**ImageBaseSizeValidator**(*extensions=None*, ***kwargs*)

Validates image size. To use this class for validating image width/height, rewrite self.orientation to ('height',) or ('width',) or ('height', 'width')

Raises rest_framework.exceptions.ValidationError: if not(min <= (height or width) <= max)

Parameters

extensions (typing.Union[159][typing.Tuple[160], typing.List[161], None[162]]) –

class vstutils.api.validators.**ImageHeightValidator**(*extensions=None*, ***kwargs*)

Wrapper for ImageBaseSizeValidator that validates only height

Parameters

- **min_height** – minimal height of an image being validated
- **max_height** – maximal height of an image being validated
- **extensions** (typing.Union[163][typing.Tuple[164], typing.List[165], None[166]]) –

class vstutils.api.validators.**ImageOpenValidator**(*extensions=None*, ***kwargs*)

Image validator that checks if image can be unpacked from b64 to PIL Image obj. Won't work if Pillow isn't installed.

Raises rest_framework.exceptions.ValidationError if PIL throws error when trying to open image

Parameters

extensions (typing.Union[167][typing.Tuple[168], typing.List[169], None[170]]) –

class vstutils.api.validators.**ImageResolutionValidator**(*extensions=None*, ***kwargs*)

Wrapper for ImageBaseSizeValidator that validates both height and width

Parameters

[155] https://docs.python.org/3.8/library/typing.html#typing.Union
[156] https://docs.python.org/3.8/library/typing.html#typing.Tuple
[157] https://docs.python.org/3.8/library/typing.html#typing.List
[158] https://docs.python.org/3.8/library/constants.html#None
[159] https://docs.python.org/3.8/library/typing.html#typing.Union
[160] https://docs.python.org/3.8/library/typing.html#typing.Tuple
[161] https://docs.python.org/3.8/library/typing.html#typing.List
[162] https://docs.python.org/3.8/library/constants.html#None
[163] https://docs.python.org/3.8/library/typing.html#typing.Union
[164] https://docs.python.org/3.8/library/typing.html#typing.Tuple
[165] https://docs.python.org/3.8/library/typing.html#typing.List
[166] https://docs.python.org/3.8/library/constants.html#None
[167] https://docs.python.org/3.8/library/typing.html#typing.Union
[168] https://docs.python.org/3.8/library/typing.html#typing.Tuple
[169] https://docs.python.org/3.8/library/typing.html#typing.List
[170] https://docs.python.org/3.8/library/constants.html#None

- **min_height** – minimal height of an image being validated
- **max_height** – maximal height of an image being validated
- **min_width** – minimal width of an image being validated
- **max_width** – maximal width of an image being validated
- **extensions** (typing.Union[171][typing.Tuple[172], typing.List[173], None[174]]) –

class vstutils.api.validators.**ImageValidator**(*extensions=None*, ***kwargs*)

Base Image Validation class. Validates image format. Won't work if Pillow isn't installed. Base Image Validation class. Validates media types.

Parameters

extensions (typing.Union[175][typing.Tuple[176], typing.List[177], None[178]]) – Tuple or List of file extensions, that should pass the validation

Raises rest_framework.exceptions.ValidationError: in case file extension are not in the list

property has_pillow

Check if Pillow is installed

class vstutils.api.validators.**ImageWidthValidator**(*extensions=None*, ***kwargs*)

Wrapper for ImageBaseSizeValidator that validates only width

Parameters

- **min_width** – minimal width of an image being validated
- **max_width** – maximal width of an image being validated
- **extensions** (typing.Union[179][typing.Tuple[180], typing.List[181], None[182]]) –

class vstutils.api.validators.**RegularExpressionValidator**(*regexp=None*)

Class for regular expression based validation

Raises

rest_framework.exceptions.ValidationError – in case value does not match regular expression

Parameters

regexp (typing.Optional[183][typing.Pattern[184]]) –

class vstutils.api.validators.**UrlQueryStringValidator**(*regexp=None*)

Class for validation url query string, for example a=&b=1

[171] https://docs.python.org/3.8/library/typing.html#typing.Union
[172] https://docs.python.org/3.8/library/typing.html#typing.Tuple
[173] https://docs.python.org/3.8/library/typing.html#typing.List
[174] https://docs.python.org/3.8/library/constants.html#None
[175] https://docs.python.org/3.8/library/typing.html#typing.Union
[176] https://docs.python.org/3.8/library/typing.html#typing.Tuple
[177] https://docs.python.org/3.8/library/typing.html#typing.List
[178] https://docs.python.org/3.8/library/constants.html#None
[179] https://docs.python.org/3.8/library/typing.html#typing.Union
[180] https://docs.python.org/3.8/library/typing.html#typing.Tuple
[181] https://docs.python.org/3.8/library/typing.html#typing.List
[182] https://docs.python.org/3.8/library/constants.html#None
[183] https://docs.python.org/3.8/library/typing.html#typing.Optional
[184] https://docs.python.org/3.8/library/typing.html#typing.Pattern

Parameters
regexp (typing.Optional[185][typing.Pattern[186]]) –

vstutils.api.validators.**resize_image**(*img*, *width*, *height*)

Utility function to resize image proportional to specific values. Can create white margins if it's needed to satisfy required size

Parameters

- **img** (*PIL.Image*) – Pillow Image object
- **width** (*int*[187]) – Required width
- **height** (*int*[188]) – Required height

Returns
Pillow Image object

Return type
PIL.Image

vstutils.api.validators.**resize_image_from_to**(*img*, *limits*)

Utility function to resize image proportional to values between min and max values for each side. Can create white margins if it's needed to satisfy restrictions

Parameters

- **img** (*PIL.Image*) – Pillow Image object
- **limits** (*dict*[189]) – Dict with min/max side restrictions like: `{'width': {'min': 300, 'max: 600'}, 'height': {'min': 400, 'max: 800'}}`

Returns
Pillow Image object

Return type
PIL.Image

3.2.3 Serializers

Default serializer classes for web-api. Read more in Django REST Framework documentation for Serializers[190].

class vstutils.api.serializers.**BaseSerializer**(**args*, ***kwargs*)

Default serializer with logic to work with objects.

This serializer serves as a base class for creating serializers to work with non-model objects. It extends the 'rest_framework.serializers.Serializer' class and includes additional logic for handling object creation and updating.

Note: You can set the `generated_fields` attribute in the `Meta` class to automatically include default CharField fields. You can also customize the field creation using the `generated_field_factory` attribute.

Example:

[185] https://docs.python.org/3.8/library/typing.html#typing.Optional
[186] https://docs.python.org/3.8/library/typing.html#typing.Pattern
[187] https://docs.python.org/3.8/library/functions.html#int
[188] https://docs.python.org/3.8/library/functions.html#int
[189] https://docs.python.org/3.8/library/stdtypes.html#dict
[190] https://www.django-rest-framework.org/api-guide/serializers/

```
class MySerializer(BaseSerializer):
    class Meta:
        generated_fields = ['additional_field']
        generated_field_factory = lambda f: drf_fields.IntegerField()
```

In this example, the `MySerializer` class extends `BaseSerializer` and includes an additional generated field.

class vstutils.api.serializers.**DisplayMode**(*value*, *names=None*, ***, *module=None*, *qualname=None*, *type=None*, *start=1*, *boundary=None*)

For any serializer displayed on frontend property *_display_mode* can be set to one of this values.

DEFAULT = 'DEFAULT'

Will be used if no mode provided.

STEP = 'STEP'

Each properties group displayed as separate tab. On creation displayed as multiple steps.

class vstutils.api.serializers.**EmptySerializer**(**args*, ***kwargs*)

Default serializer for empty responses. In generated GUI this means that action button won't show additional view before execution.

class vstutils.api.serializers.**VSTSerializer**(**args*, ***kwargs*)

Default model serializer based on `rest_framework.serializers.ModelSerializer`. Read more in DRF documentation[191] how to create Model Serializers. This serializer matches model fields to extended set of serializer fields.

List of available pairs specified in *VSTSerializer.serializer_field_mapping*. For example, to set *vstutils.api.fields.FkModelField* in serializer use *vstutils.models.fields.FkModelField* in a model.

Example:

```
class MyModel(models.Model):
    name = models.CharField(max_length=255)

class MySerializer(VSTSerializer):
    class Meta:
        model = MyModel
```

In this example, the `MySerializer` class extends `VSTSerializer` and is associated with the `MyModel` model.

3.2.4 Views

VSTUtils extends the standard behavior of ViewSets from Django REST Framework (DRF) by providing built-in mechanisms for managing model views that cater to the most commonly encountered development patterns. This framework enhancement simplifies the process of creating robust and scalable web applications by offering a rich set of tools and utilities that automate much of the boilerplate code required in API development. Through these extensions, developers can easily implement custom business logic, data validation, and access control with minimal effort, thus significantly reducing development time and improving code maintainability.

class vstutils.api.base.**CopyMixin**(***kwargs*)

Mixin for viewsets which adds *copy* endpoint to view.

[191] https://www.django-rest-framework.org/api-guide/serializers/#modelserializer

copy (*request*, ***kwargs*)

Endpoint which copy instance with deps.

copy_field_name = 'name'

Name of field which will get a prefix.

copy_prefix = 'copy-'

Value of prefix which will be added to new instance name.

copy_related = ()

List of related names which will be copied to new instance.

class vstutils.api.base.**FileResponseRetrieveMixin**(***kwargs*)

ViewSet mixin for retrieving FileResponse from models with file fields data.

Example:

```
import datetime
import os
from django.db import models
from django.db.models.functions import Now
from rest_framework import permissions, fields as drf_fields
from vstutils.api.serializers import BaseSerializer, DataSerializer
from vstutils.models.decorators import register_view_action
from vstutils.models.custom_model import ListModel, FileModel,␣
↪ViewCustomModel
from vstutils.api import fields, base, responses

from .cacheable import CachableModel


class TestQuerySerializer(BaseSerializer):
    test_value = drf_fields.ChoiceField(required=True, choices=("TEST1",
↪"TEST2"))


class FileViewMixin(base.FileResponseRetrieveMixin):
    # required always
    instance_field_data = 'value'
    # required for response caching in browser
    instance_field_timestamp = 'updated'
    @register_view_action(
        methods=['get'],
        detail=False,
        query_serializer=TestQuerySerializer,
        serializer_class=DataSerializer,
        suffix='Instance'
    )
    def query_serializer_test(self, request):
        query_validated_data = self.get_query_serialized_data(request)
        return responses.HTTP_200_OK(query_validated_data)

    @register_view_action(
        methods=['get'],
        detail=False,
        query_serializer=TestQuerySerializer,
        is_list=True
    )
```

(continues on next page)

(continued from previous page)

```
    def query_serializer_test_list(self, request):
        return self.list(request)
```

serializer_class_retrieve

alias of `FileResponse`

class vstutils.api.base.**GenericViewSet**(***kwargs*)

The base class for all views. Extends the standard features of the DRF class. Here are some of the possibilities:

- Provides `model` attribute instead of `queryset`.
- Provides to set serializers for each action separately through a dictionary `action_serializers` or attributes starting with `serializer_class_[action name]`.
- Provides to specify a serializer for lists and detail views separately.
- Optimizes the database query for GET requests, if possible, by selecting only the fields necessary for the serializer.

create_action_serializer(**args*, ***kwargs*)

A method that implements the standard logic for actions. It relies on the passed arguments to build logic. So, if the named argument data was passed, then the serializer will be validated and saved.

Parameters

- **autosave** (*bool*[192]) – Enables/disables the execution of saving by the serializer if named argument *data* passed. Enabled by default.
- **custom_data** (*dict*[193]) – Dict with data which will passed to *validated_data* without validation.
- **serializer_class** (*None*, *type*[194]*[rest_framework.serializers.Serializer]*) – Serializer class for this execution. May be useful when request and response serializers are different.

Param
data: Default serializer class argument with serializable data. Enables validation and saving.

Param
instance: Default serializer class argument with serializable instance.

Returns
Ready serializer with default logic performed.

Return type
rest_framework.serializers.Serializer

get_query_serialized_data(*request*, *query_serializer=None*, *raise_exception=True*)

Get request query data and serialize values if *query_serializer_class* attribute exists or attribute was send.

Parameters

- **request** – DRF request object.
- **query_serializer** – serializer class for processing parameters in query_params.
- **raise_exception** – flag that indicates whether an exception should be thrown during validation in the serializer or not.

get_serializer(**args*, ***kwargs*)

Return the serializer instance that should be used for validating and deserializing input, and for serializing output.

Provide to use `django.http.StreamingHttpResponse`[195] as serializer init.

get_serializer_class()

Provides to setup serializer class for each action.

nested_allow_check()

Just raise or pass. Used for nested views for easier access checking.

class vstutils.api.base.**HistoryModelViewSet**(***kwargs*)

Default viewset like ReadOnlyModelViewSet but for historical data (allow to delete, but can't create and update). Inherited from *GenericViewSet*.

class vstutils.api.base.**ModelViewSet**(***kwargs*)

A viewset that provides CRUD actions under model. Inherited from *GenericViewSet*.

Variables

- **model** (vstutils.models.BModel) – DB model with data.
- **serializer_class** (vstutils.api.serializers.VSTSerializer) – Serializer for view of Model data.
- **serializer_class_one** (vstutils.api.serializers.VSTSerializer) – Serializer for view one instance of Model data.
- **serializer_class_[ACTION_NAME]** (vstutils.api.serializers.VSTSerializer) – Serializer for view of any endpoint like *.create*.

Examples:

```
from vstutils.api.base import ModelViewSet
from . import serializers as sers


class StageViewSet(ModelViewSet):
    # This is difference with DRF:
    # we use model instead of queryset
    model = sers.models.Stage
    # Serializer for list view (view for a list of Model instances
    serializer_class = sers.StageSerializer
    # Serializer for page view (view for one Model instance).
    # This property is not required, if its value is the same as `serializer_
↪class`.
    serializer_class_one = sers.StageSerializer
    # Allowed to set decorator to custom endpoint like this:
    # serializer_class_create - for create method
    # serializer_class_copy - for detail endpoint `copy`.
    # etc...
```

[192] https://docs.python.org/3.8/library/functions.html#bool
[193] https://docs.python.org/3.8/library/stdtypes.html#dict
[194] https://docs.python.org/3.8/library/functions.html#type
[195] https://docs.djangoproject.com/en/4.2/ref/request-response/#django.http.StreamingHttpResponse

class vstutils.api.base.**ReadOnlyModelViewSet**(***kwargs*)

Default viewset like vstutils.api.base.ModelViewSet for readonly models. Inherited from *GenericViewSet*.

class vstutils.api.decorators.**nested_view**(*name*, *arg=None*, *methods=None*, **args*, ***kwargs*)

By default, DRF does not support nested views. This decorator solves this problem.

You need two or more models with nested relationship (Many-to-Many or Many-to-One) and two viewsets. Decorator nests viewset to parent viewset class and generate paths in API.

Parameters

- **name** (*str*[196]) – Name of nested path. Also used as default name for related queryset (see *manager_name*).
- **arg** (*str*[197]) – Name of nested primary key field.
- **view** (vstutils.api.base.ModelViewSet, vstutils.api.base.HistoryModelViewSet, vstutils.api.base.ReadOnlyModelViewSet) – Nested viewset class.
- **allow_append** (*bool*[198]) – Flag for allowing to append existed instances.
- **manager_name** (*str*[199], *Callable*[200]) – Name of model-object attr which contains nested queryset.
- **methods** (*list*[201]) – List of allowed methods to nested view endpoints.
- **subs** (*list*[202], *None*) – List of allowed subviews or actions to nested view endpoints.
- **queryset_filters** – List of callable objects which returns filtered queryset of main.

Note: Some view methods will not call for performance reasons. This also applies to some of the class attributes that are usually initialized in the methods. For example, `.initial()` will never call. Each viewset wrapped by nested class with additional logic.

Example:

```
from vstutils.api.decorators import nested_view
from vstutils.api.base import ModelViewSet
from . import serializers as sers


class StageViewSet(ModelViewSet):
    model = sers.models.Stage
    serializer_class = sers.StageSerializer


nested_view('stages', 'id', view=StageViewSet)
class TaskViewSet(ModelViewSet):
    model = sers.models.Task
    serializer_class = sers.TaskSerializer
```

This code generates api paths:

- */tasks/* - GET, POST
- */tasks/{id}/* - GET, PUT, PATCH, DELETE
- */tasks/{id}/stages/* - GET, POST
- */tasks/{id}/stages/{stages_id}/* - GET, PUT, PATCH, DELETE

vstutils.api.decorators.**subaction**(**args*, ***kwargs*)

Decorator which wrap object method to subaction of viewset.

Parameters

- **methods** – List of allowed HTTP-request methods. Default is `["post"]`.
- **detail** – Flag to set method execution to one instance.
- **serializer_class** – Serializer for this action.
- **permission_classes** – Tuple or list permission classes.
- **url_path** – API-path name for this action.
- **description** – Description for this action in OpenAPI.
- **multiaction** – Allow to use this action in multiactions. Works only with *vstutils.api.serializers.EmptySerializer* as response.
- **require_confirmation** – Sets whether the action must be confirmed before being executed.
- **is_list** – Mark this action as paginated list with all rules and parameters.
- **title** – Override action title.
- **icons** – Setup action icon classes.

An ETag (Entity Tag) is a mechanism defined by the HTTP protocol for web cache validation and to manage resource versions efficiently. It represents a unique identifier for the content of a resource at a given time, allowing client and server to determine if the resource has changed without downloading the entire content. This mechanism significantly reduces bandwidth and improves web performance by enabling conditional requests. Servers send ETags in responses to clients, which can cache these tags along with the resources. On subsequent requests, clients send the cached ETag back to the server in an If-None-Match header. If the resource has not changed (the ETag matches), the server responds with a 304 Not Modified status, indicating that the client's cached version is up-to-date.

Beyond GET requests, ETags can also be instrumental in other HTTP methods like PUT or DELETE to ensure consistency and prevent unintended overwrites or deletions, known as "mid-air collision avoidance." By including an ETag in the If-Match header of non-GET requests, clients signal that the operation should proceed only if the resource's current state matches the specified ETag, thus safeguarding against concurrent modifications by different clients. This application of ETags enhances the reliability and integrity of web applications by ensuring that operations are performed on the correct version of a resource.

Here is main functionality provided for working with ETag's mechanism:

class vstutils.api.base.**CachableHeadMixin**(***kwargs*)

A mixin designed to enhance caching for GET responses in Django REST framework views, leveraging the standard HTTP caching mechanism. It returns a 304 Not Modified status for reading requests like GET or HEAD when the ETag (Entity Tag) in the client's request matches the current resource state, and a 412 Precondition Failed status for writing requests when the condition fails. This approach reduces unnecessary network traffic and load times for unchanged resources, and ensures data consistency for write operations.

[196] https://docs.python.org/3.8/library/stdtypes.html#str
[197] https://docs.python.org/3.8/library/stdtypes.html#str
[198] https://docs.python.org/3.8/library/functions.html#bool
[199] https://docs.python.org/3.8/library/stdtypes.html#str
[200] https://docs.python.org/3.8/library/typing.html#typing.Callable
[201] https://docs.python.org/3.8/library/stdtypes.html#list
[202] https://docs.python.org/3.8/library/stdtypes.html#list

The mixin relies on *get_etag_value()* and *check_request_etag()* functions within the *GenericViewSet* context to dynamically generate and validate ETags for resource states. By comparing ETags, it determines whether content has changed since the last request, allowing clients to reuse cached responses when applicable and preventing concurrent write operations from overwriting each other without acknowledgment of updated state.

> **Warning:** This mixin is designed to work with models that inherit from *vstutils.models.BModel*. Usage with other models may not provide the intended caching behavior and could lead to incorrect application behavior.

Note: For effective use, ensure model classes are compatible with ETag generation and validation by implementing the *get_etag_value()* method for custom ETag computation. Additionally, the *GenericViewSet* using this mixin should properly handle ETag headers in client requests to leverage HTTP caching.

An additional feature of the `CachableHeadMixin` is its automatic inclusion in the generated view from a *vstutils.models.BModel* if the model class has the `_cache_responses` class attribute set to `True`. This enables automatic caching capabilities for models indicating readiness for HTTP-based caching, streamlining the process of optimizing response times and reducing server load for frequently accessed resources.

class vstutils.api.base.**EtagDependency**(*value*, *names=None*, ***, *module=None*, *qualname=None*, *type=None*, *start=1*, *boundary=None*)

A custom enumeration that defines potential dependencies for ETag generation. It includes:

LANG = 4

Indicates dependency on the user's language preference.

SESSION = 2

Indicates dependency on the user's session.

USER = 1

Indicates dependency on the user's identity.

vstutils.api.base.**check_request_etag**(*request*, *etag_value*, *header_name='If-None-Match'*, *operation_handler=<slot wrapper '__eq__' of 'str' objects>*)

The function plays a crucial role within the context of the ETag mechanism, providing a flexible way to validate client-side ETags against the server-side version for both cache validation and ensuring data consistency in web applications. It supports conditional handling of HTTP requests based on the match or mismatch of ETag values, accommodating various scenarios such as cache freshness checks and prevention of concurrent modifications.

Parameters

- **request** (`rest_framework.request.Request`) – The HTTP request object containing the client's headers, from which the ETag for comparison is retrieved.
- **etag_value** (`str`[203]) – The server-generated ETag value that represents the current state of the resource. This unique identifier is recalculated whenever the resource's content changes.
- **header_name** (`str`[204]) – Specifies the HTTP header to look for the client's ETag. Defaults to "If-None-Match", commonly used in GET requests for cache validation. For operations requiring confirmation that the client is acting on the latest version of a resource (e.g., PUT or DELETE), "If-Match" should be used instead.
- **operation_handler** – A function to compare the ETags. By default, this is set to `str.__eq__`, which checks for an exact match between the client's and server's ETags, suitable for validating caches with `If-None-Match`. To handle `If-Match` scenarios, where the

operation should proceed only if the ETags do not match, indicating the resource has been modified, `str.__ne__` (not equal) can be used as the operation handler. This flexibility allows for precise control over how and when clients are allowed to read from or write to resources based on their version.

Returns

Returns a tuple containing the server's ETag and a boolean flag. The flag is `True` if the operation handler condition between the server's and client's ETag is met, indicating the request should proceed based on the matching logic defined by the operation handler; otherwise, it returns `False`.

`vstutils.api.base.`**`get_etag_value`**(*view*, *model_class*, *request*, *pk=None*)

The get_etag_value function is designed to compute a unique ETag value based on the model's state, request parameters, and additional dependencies such as user language, session, and user identity. This function supports both single models and collections of models.

Parameters

- **`view`** (*GenericViewSet*) – An instance of *GenericViewSet*, responsible for view operations.
- **`model_class`** (`django.db.models.Model`[205], `list`[206], `tuple`[207], or `set`[208]) – The model class for which the ETag value is being generated. This parameter can be a single model class or a collection of model classes (`list`[209], `tuple`[210], or `set`[211]). Each model class may optionally implement a class method named `get_etag_value`.
- **`request`** (`rest_framework.request.Request`) – The request object from the Django REST framework, containing all the HTTP request information.
- **`pk`** – he primary key of the model instance for which the ETag is being calculated. This can be a specific value (int or str) for single model usage, or a dictionary mapping model classes to their respective primary key values for collections of models.

Returns

The computed ETag value as a hexadecimal string.

Return type

`str`[212]

The function operates differently based on the type of model_class provided:

- **Collection of Models**: When model_class is a collection, the function computes an ETag by concatenating ETag values of individual models in the collection, using a recursive call for each model. The ETag value for each model is encoded and hashed using Blake2s algorithm.
- **Model with ``get_etag_value`` method**: If the model class has a get_etag_value method, the function calls this method to obtain a base ETag value. It then appends language, user ID, and session key information if they are marked as dependencies in the model's `_cache_response_dependencies` attribute. This base ETag may be further processed to include the application's full version string and hashed if user or session information is included.
- **Model without ``get_etag_value`` method**: For models lacking a custom get_etag_value method, the function generates an ETag based on the model's class name and a hash of the application's full version string.

[203] https://docs.python.org/3.8/library/stdtypes.html#str
[204] https://docs.python.org/3.8/library/stdtypes.html#str

3.2.5 Actions

Vstutils has the advanced system of working with actions. REST API works with data through verbs, which are called methods. However, to work with one or a list of entities, such actions may not be enough.

To expand the set of actions, you need to create an action that will work with some aspect of the described model. For these purposes, there is a standard `rest_framework.decorators.action()`, which can also be extended using the scheme. But for the greater convenience, there is a set of decorator objects in vstutils to eliminate the routine of writing boilerplate code.

The main philosophy for these wrappers is that the developer writes business logic without being distracted by the boilerplate code. Often, most of the errors in the code appear precisely because of the blurry look from the routine writing of the code.

class vstutils.api.actions.**Action**(*detail=True*, *methods=None*, *serializer_class=<class 'vstutils.api.serializers.DataSerializer'>*, *result_serializer_class=None*, *query_serializer=None*, *multi=False*, *title=None*, *icons=None*, *is_list=False*, *hidden=False*, *require_confirmation=False*, ***kwargs*)

Base class of actions. Has minimal of required functionality to create an action and write only business logic. This decorator is suitable in cases where it is not possible to implement the logic using *SimpleAction* or the algorithm is much more complicated than standard CRUD.

Examples:

```
...
from vstutils.api.fields import VSTCharField
from vstutils.api.serializers import BaseSerializer
from vstutils.api.base import ModelViewSet
from vstutils.api.actions import Action
...

class RequestSerializer(BaseSerializer):
    data_field1 = ...
    ...


class ResponseSerializer(BaseSerializer):
    detail = VSTCharField(read_only=True)


class AuthorViewSet(ModelViewSet):
    model = ...
    ...

    @Action(serializer_class=RequestSerializer, result_serializer_
↪class=ResponseSerializer, ...)
    def profile(self, request, *args, **kwargs):
        ''' Got `serializer_class` body and response with `result_
↪serializer_class`. '''
```

(continues on next page)

[205] https://docs.djangoproject.com/en/4.2/ref/models/instances/#django.db.models.Model
[206] https://docs.python.org/3.8/library/stdtypes.html#list
[207] https://docs.python.org/3.8/library/stdtypes.html#tuple
[208] https://docs.python.org/3.8/library/stdtypes.html#set
[209] https://docs.python.org/3.8/library/stdtypes.html#list
[210] https://docs.python.org/3.8/library/stdtypes.html#tuple
[211] https://docs.python.org/3.8/library/stdtypes.html#set
[212] https://docs.python.org/3.8/library/stdtypes.html#str

(continued from previous page)

```
        serializer = self.get_serializer(self.get_object(), data=request.
↪data)
        serializer.is_valid(raise_exception=True)
        return {"detail": "Executed!"}
```

Parameters

- **detail** – Flag indicating which type of action is used: on a list or on a single entity. Affects where this action will be displayed - on a detailed record or on a list of records.
- **methods** – List of available HTTP-methods for this action. Default has only *POST* method.
- **serializer_class** – Request body serializer. Also used for default response.
- **result_serializer_class** – Response body serializer. Required, when request and response has different set of fields.
- **query_serializer** – GET-request query data serializer. It is used when it is necessary to get valid data in the query data of a GET-request and cast it to the required type.
- **multi** – Used only with non-GET requests and notify GUI, that this action should be rendered over the selected list items.
- **title** – Title for action in UI. For non-GET actions, title is generated from method's name.
- **icons** – List of icons for UI button.
- **is_list** – Flag indicating whether the action type is a list or a single entity. Typically used with GET actions.
- **require_confirmation** – If true user will be asked to confirm action execution on frontend.
- **kwargs** – Set of named arguments for `rest_framework.decorators.action()`.

class vstutils.api.actions.**EmptyAction**(***kwargs*)

In this case, actions on an object do not require any data and manipulations with them. For such cases, there is a standard method that allows you to simplify the scheme and code to work just with the object.

Optionally, you can also overload the response serializer to notify the interface about the format of the returned data.

Examples:

```
...
from vstutils.api.fields import RedirectIntegerField
from vstutils.api.serializers import BaseSerializer
from vstutils.api.base import ModelViewSet
from vstutils.api.actions import EmptyAction
...

class ResponseSerializer(BaseSerializer):
    id = RedirectIntegerField(operation_name='sync_history')


class AuthorViewSet(ModelViewSet):
    model = ...
    ...
```

(continues on next page)

(continued from previous page)

```
    @EmptyAction(result_serializer_class=ResponseSerializer, ...)
    def sync_data(self, request, *args, **kwargs):
        ''' Example of action which get object, sync data and redirect␣
↪user to another view. '''
        sync_id = self.get_object().sync().id
        return {"id": sync_id}
```

```
...
from django.http.response import FileResponse
from vstutils.api.base import ModelViewSet
from vstutils.api.actions import EmptyAction
...

class AuthorViewSet(ModelViewSet):
    model = ...
    ...

    @EmptyAction(result_serializer_class=ResponseSerializer, ...)
    def resume(self, request, *args, **kwargs):
        ''' Example of action which response with generated resume in pdf.
↪ '''
        instance = self.get_object()

        return FileResponse(
            streaming_content=instance.get_pdf(),
            as_attachment=True,
            filename=f'{instance.last_name}_{instance.first_name}_resume.
↪pdf'
        )
```

class vstutils.api.actions.**SimpleAction**(**args*, ***kwargs*)

The idea of this decorator is to get the full CRUD for the instance in a minimum of steps. The instance is the object that was returned from the method being decorated. The whole mechanism is very similar to the standard property decorator, with a description of a getter, setter, and deleter.

If you're going to create an entry point for working with a single object, then you do not need to define methods. The presence of a getter, setter, and deleter will determine which methods will be available.

In the official documentation of Django, an example is given with moving data that is not important for authorization to the Profile model. To work with such data that is outside the main model, there is this action object, which implements the basic logic in the most automated way.

It covers most of the tasks for working with such data. By default, it has a GET method instead of POST. Also, for better organization of the code, it allows you to change the methods that will be called when modifying or deleting data.

When assigning an action on an object, the list of methods is also filled with the necessary ones.

Examples:

```
...
from vstutils.api.fields import PhoneField
from vstutils.api.serializers import BaseSerializer
from vstutils.api.base import ModelViewSet
from vstutils.api.actions import Action
...
```

(continues on next page)

(continued from previous page)

```
class ProfileSerializer(BaseSerializer):
    phone = PhoneField()


class AuthorViewSet(ModelViewSet):
    model = ...
    ...

    @SimpleAction(serializer_class=ProfileSerializer, ...)
    def profile(self, request, *args, **kwargs):
        ''' Get profile data to work. '''
        return self.get_object().profile

    @profile.setter
    def profile(self, instance, request, serializer, *args, **kwargs):
        instance.save(update_fields=['phone'])

    @profile.deleter
    def profile(self, instance, request, serializer, *args, **kwargs):
        instance.phone = ''
        instance.save(update_fields=['phone'])
```

3.2.6 Filtersets

For greater development convenience, the framework provides additional classes and functions for filtering elements by fields.

class vstutils.api.filters.**DefaultIDFilter**(*data=None*, *queryset=None*, ***, *request=None*, *prefix=None*)

Basic filterset to search by id. Provides a search for multiple values separated by commas. Uses *extra_filter()* in fields.

class vstutils.api.filters.**DefaultNameFilter**(*data=None*, *queryset=None*, ***, *request=None*, *prefix=None*)

Basic filterset to search by part of name. Uses *LIKE* DB condition by *name_filter()*.

class vstutils.api.filters.**FkFilterHandler**(*related_pk='id'*, *related_name='name'*, *pk_handler=<class 'int'>*)

Simple handler for filtering by relational fields.

Parameters

- **related_pk** (str[213]) – Field name of related model's primary key. Default is 'id'.
- **related_name** (str[214]) – Field name of related model's charfield. Default is 'name'.
- **pk_handler** (typing.Callable[215]) – Changes handler for checking value before search. Sends "0" if handler falls. Default is 'int()'.

Example:

```
class CustomFilterSet(filters.FilterSet):
    author = CharFilter(method=vst_filters.FkFilterHandler(related_pk='pk',␣
↪related_name='email'))
```

Where `author` is ForeignKey to *User* and you want to search by primary key and email.

vstutils.api.filters.**extra_filter**(*queryset*, *field*, *value*)

Method for searching values in a comma-separated list.

Parameters

- **queryset** (*django.db.models.query.QuerySet*[216]) – model queryset for filtration.
- **field** (*str*[217]) – field name in FilterSet. Also supports *__not* suffix.
- **value** (*str*[218]) – comma separated list of searching values.

Returns
filtered queryset.

Return type
django.db.models.query.QuerySet[219]

vstutils.api.filters.**name_filter**(*queryset*, *field*, *value*)

Method for searching by part of name. Uses *LIKE* DB condition or *contains* qs-expression.

Parameters

- **queryset** (*django.db.models.query.QuerySet*[220]) – model queryset for filtration.
- **field** (*str*[221]) – field name in FilterSet. Also supports *__not* suffix.
- **value** (*str*[222]) – searching part of name.

Returns
filtered queryset.

Return type
django.db.models.query.QuerySet[223]

3.2.7 Responses

DRF provides a standard set of variables whose names correspond to the human-readable name of the HTTP code. For convenience, we have dynamically wrapped it in a set of classes that have appropriate names and additionally provides following capabilities:

- String responses are wrapped in json like `{ "detail": "string response" }`.
- Attribute timings are kept for further processing in middleware.
- Status code is set by class name (e.g. `HTTP_200_OK` or `Response200` has code 200).

All classes inherit from:

[213] https://docs.python.org/3.8/library/stdtypes.html#str
[214] https://docs.python.org/3.8/library/stdtypes.html#str
[215] https://docs.python.org/3.8/library/typing.html#typing.Callable
[216] https://docs.djangoproject.com/en/4.2/ref/models/querysets/#django.db.models.query.QuerySet
[217] https://docs.python.org/3.8/library/stdtypes.html#str
[218] https://docs.python.org/3.8/library/stdtypes.html#str
[219] https://docs.djangoproject.com/en/4.2/ref/models/querysets/#django.db.models.query.QuerySet
[220] https://docs.djangoproject.com/en/4.2/ref/models/querysets/#django.db.models.query.QuerySet
[221] https://docs.python.org/3.8/library/stdtypes.html#str
[222] https://docs.python.org/3.8/library/stdtypes.html#str
[223] https://docs.djangoproject.com/en/4.2/ref/models/querysets/#django.db.models.query.QuerySet

class vstutils.api.responses.**BaseResponseClass**(**args*, ***kwargs*)

API response class with default status code.

Variables

- **status_code** (*int*[224]) – HTTP status code.
- **timings** (*int*[225], *None*) – Response timings.

Parameters

timings – Response timings.

3.2.8 Middleware

By default, Django supposes[226] that a developer creates Middleware class manually, but it's often a routine. The vstutils library offers a convenient request handler class for elegant OOP development. Middleware is used to process incoming requests and send responses before they reach final destination.

class vstutils.middleware.**BaseMiddleware**(*get_response*)

Middleware base class for handling:

- Incoming requests by *BaseMiddleware.request_handler()*;
- Outgoing response before any calling on server by *BaseMiddleware.get_response_handler()*;
- Outgoing responses by *BaseMiddleware.handler()*.

Middleware must be added to *MIDDLEWARE* list in settings.

Example:

```
from vstutils.middleware import BaseMiddleware
from django.http import HttpResponse


class CustomMiddleware(BaseMiddleware):
    def request_handler(self, request):
        # Add header to request
        request.headers['User-Agent'] = 'Mozilla/5.0'
        return request

    def get_response_handler(self, request):
        if not request.user.is_stuff:
            # Return 403 HTTP status for non-stuff users.
            # This request never gets in any view.
            return HttpResponse(
                "Access denied!",
                content_type="text/plain",
                status_code=403
            )
        return super().get_response_handler(request)

    def handler(self, request, response):
        # Add header to response
        response['Custom-Header'] = 'Some value'
        return response
```

[224] https://docs.python.org/3.8/library/functions.html#int
[225] https://docs.python.org/3.8/library/functions.html#int
[226] https://docs.djangoproject.com/en/4.1/topics/http/middleware/#writing-your-own-middleware

get_response_handler(*request*)

Entrypoint for breaking or continuing request handling. This function must return *django.http.HttpResponse* object or result of parent class calling.

Since the release of 5.3, it has been possible to write this method as asynchronous. This should be used in cases where the middleware makes queries to the database or cache. However, such a middleware should be excluded from bulk requests.

> **Warning:** Never do asynchronous middleware in dependent chains. They are designed to send independent requests to external sources.
>
> Set `async_capable` to `True` and `sync_capable` to `False` for such middleware.

Parameters

request (*django.http.HttpRequest*[227]) – HTTP-request object which is wrapped from client request.

Return type

django.http.HttpResponse[228]

handler(*request*, *response*)

The response handler. Method to process responses.

Parameters

- **request** (*django.http.HttpRequest*[229]) – HTTP-request object.
- **response** (*django.http.HttpResponse*[230]) – HTTP-response object which will be sended to client.

Returns

Handled response object.

Return type

django.http.HttpResponse[231]

request_handler(*request*)

The request handler. Called before request is handled by a view.

Parameters

request (*django.http.HttpRequest*[232]) – HTTP-request object which is wrapped from client request.

Returns

Handled request object.

Return type

django.http.HttpRequest[233]

[227] https://docs.djangoproject.com/en/4.2/ref/request-response/#django.http.HttpRequest
[228] https://docs.djangoproject.com/en/4.2/ref/request-response/#django.http.HttpResponse
[229] https://docs.djangoproject.com/en/4.2/ref/request-response/#django.http.HttpRequest
[230] https://docs.djangoproject.com/en/4.2/ref/request-response/#django.http.HttpResponse
[231] https://docs.djangoproject.com/en/4.2/ref/request-response/#django.http.HttpResponse
[232] https://docs.djangoproject.com/en/4.2/ref/request-response/#django.http.HttpRequest
[233] https://docs.djangoproject.com/en/4.2/ref/request-response/#django.http.HttpRequest

3.2.9 Filter Backends

Filter Backends[234] are used to modify model queryset. To create custom filter backend to, (i.e. annotate model queryset), you should inherit from *vstutils.api.filter_backends.VSTFilterBackend* and override `vstutils.api.filter_backends.VSTFilterBackend.filter_queryset()` and in some cases `vstutils.api.filter_backends.VSTFilterBackend.get_schema_fields()`.

class vstutils.api.filter_backends.**DeepViewFilterBackend**

Backend that filters queryset by column from *deep_parent_field* property of the model. Value for filtering must be provided in query param *__deep_parent*.

If param is missing then no filtering is applied.

If param is empty value (*/?__deep_parent=*) then objects with no parent (the value of the field whose name is stored in the property *deep_parent_field* of the model is None) returned.

This filter backend and nested view is automatically added when model has *deep_parent_field* property.

Example:

```
from django.db import models
from vstutils.models import BModel

class DeepNestedModel(BModel):
    name = models.CharField(max_length=10)
    parent = models.ForeignKey('self', null=True, default=None, on_
↪delete=models.CASCADE)

    deep_parent_field = 'parent'
    deep_parent_allow_append = True

    class Meta:
        default_related_name = 'deepnested'
```

In example above if we add this model under path '*deep*', following views will be created: */deep/* and */deep/{id}/deepnested/*.

Filter backend can be used as */deep/?__deep_parent=1* and will return all *DeepNestedModel* objects whose parent's primary key is *1*.

You can also use generic DRF views, for that you still must set *deep_parent_field* to your model and manually add *DeepViewFilterBackend* to filter_backends[235] list.

class vstutils.api.filter_backends.**HideHiddenFilterBackend**

Filter Backend that hides all objects with hidden=True from the queryset

filter_queryset(*request*, *queryset*, *view*)

Clear objects with hidden attr from queryset.

class vstutils.api.filter_backends.**SelectRelatedFilterBackend**

Filter Backend that will automatically call prefetch_related and select_related on all relations in queryset.

filter_queryset(*request*, *queryset*, *view*)

Select+prefetch related in queryset.

class vstutils.api.filter_backends.**VSTFilterBackend**

A base filter backend class to be inherited from. Example:

[234] https://www.django-rest-framework.org/api-guide/filtering/#djangofilterbackend
[235] https://www.django-rest-framework.org/api-guide/filtering/#djangofilterbackend

```
from django.utils import timezone
from django.db.models import Value, DateTimeField

from vstutils.api.filter_backends import VSTFilterBackend

class CurrentTimeFilterBackend(VSTFilterBackend):
    def filter_queryset(self, request, queryset, view):
        return queryset.annotate(current_time=Value(timezone.now(),␣
↪output_field=DateTimeField()))
```

In this example Filter Backend annotates time in current timezone to any connected model's queryset.

In some cases it may be necessary to provide a parameter from a query of request. To define this parameter in the schema, you must overload the get_schema_operation_parameters function and specify a list of parameters to use.

Example:

```
from django.utils import timezone
from django.db.models import Value, DateTimeField

from vstutils.api.filter_backends import VSTFilterBackend

class ConstantCurrentTimeForQueryFilterBackend(VSTFilterBackend):
    query_param = 'constant'

    def filter_queryset(self, request, queryset, view):
        if self.query_param in request.query_params and request.query_
↪params[self.query_param]:
            queryset = queryset.annotate(**{
                request.query_params[self.query_param]: Value(timezone.
↪now(), output_field=DateTimeField())
            })
        return queryset

        def get_schema_operation_parameters(self, view):
            return [
                {
                    "name": self.query_param,
                    "required": False,
                    "in": openapi.IN_QUERY,
                    "description": "Annotate value to queryset",
                    "schema": {
                        "type": openapi.TYPE_STRING,
                    }
                },
            ]
```

In this example Filter Backend annotates time in current timezone to any connected model's queryset with field name from query *constant*.

get_schema_operation_parameters(*view*)

You can also make the filter controls available to the schema autogeneration that REST framework provides, by implementing this method. The method should return a list of OpenAPI schema mappings.

3.3 Celery

Celery is a distributed task queue. It's used to execute some actions asynchronously in a separate worker. For more details on Celery, check it's official docs[236]. For Celery related vstutils features to work, you need to specify [rpc] and [worker] sections in settings.ini. Also you need to include extra [rpc] requirements.

3.3.1 Tasks

class vstutils.tasks.**TaskClass**

Wrapper for Celery BaseTask class. Usage is same as Celery standard class, but you can execute task without creating instance with *TaskClass.do()* method.

Example:

```
from vstutils.environment import get_celery_app
from vstutils.tasks import TaskClass

app = get_celery_app()

class Foo(TaskClass):
    def run(*args, **kwargs):
        return 'Foo task has been executed'

app.register_task(Foo())
```

Now you can call your task with various methods:

- by executing `Foo.do(*args, **kwargs)`
- get registered task instance like that - app.tasks['full_path.to.task.class.Foo']

Also you can make your registered task periodic, by adding it to CELERY_BEAT_SCHEDULE in settings.py:

```
CELERY_BEAT_SCHEDULE = {
    'foo-execute-every-month': {
        'task': 'full_path.to.task.class.Foo',
        'schedule': crontab(day_of_month=1),
    },
}
```

classmethod do(**args*, ***kwargs*)

Method which send signal to celery for start remote task execution. All arguments will passed to the task *TaskClass.run()* method.

Return type
`celery.result.AsyncResult`

property name

property for proper Celery task execution, needed for *TaskClass.do()* method to work

run(**args*, ***kwargs*)

The body of the task executed by workers.

[236] https://docs.celeryproject.org/en/stable/

3.4 Endpoint

Endpoint view has two purposes: bulk requests execution and providing OpenAPI schema.

Endpoint url is `/{API_URL}/endpoint/`, for example value with default settings is `/api/endpoint/`.

`API_URL` can be changed in `settings.py`.

class vstutils.api.endpoint.**EndpointViewSet**(***kwargs*)

Default API-endpoint viewset.

get(*request*)

Returns response with swagger ui or openapi json schema if ?format=openapi

Parameters
request (vstutils.api.endpoint.BulkRequestType) –

Return type
django.http.response.HttpResponse

get_client(*request*)

Returns test client and guarantees that if bulk request comes authenticated than test client will be authenticated with the same user

Parameters
request (vstutils.api.endpoint.BulkRequestType) –

Return type
vstutils.api.endpoint.BulkClient

get_serializer(**args*, ***kwargs*)

Return the serializer instance that should be used for validating and deserializing input, and for serializing output.

Return type
vstutils.api.endpoint.OperationSerializer

get_serializer_context(*context*)

Extra context provided to the serializer class.

Return type
dict[237]

operate(*operation_data*, *context*)

Method used to handle one operation and return result of it

Parameters

- **operation_data** (typing.Dict[238]) –
- **context** (typing.Dict[239]) –

Return type
typing.Tuple[240][typing.Dict[241], typing.SupportsFloat[242]]

post(*request*)

Execute transactional bulk request

Parameters
request (vstutils.api.endpoint.BulkRequestType) –

Return type
vstutils.api.responses.BaseResponseClass

put (*request*, *allow_fail=True*)

Execute non transaction bulk request

Parameters
request (vstutils.api.endpoint.BulkRequestType) –

Return type
vstutils.api.responses.BaseResponseClass

serializer_class

One operation serializer class.

alias of OperationSerializer

versioning_class

alias of QueryParameterVersioning

3.4.1 Bulk requests

Bulk request allows you send multiple requests to api at once, it accepts json list of operations.

Method	Transactional (all operations in one transaction)	Synchronous (operations executed one by one in given order)
PUT /{API_URL}/endpoint/	NO	YES
POST /{API_URL}/endpoint/	YES	YES
PATCH /{API_URL}/endpoint/	NO	NO

Parameters of one operation (required parameter marked by *):

- method* - http method of request
- path* - path of request, can be str or list
- data - data to send
- query - query parameters as str
- let - string with name of variable (used for access to response result in templates)
- headers - dict with headers which will be sent (key - header's name, value - header's value string).
- version - str with specified version of api, if not provided then VST_API_VERSION will be used

[237] https://docs.python.org/3.8/library/stdtypes.html#dict
[238] https://docs.python.org/3.8/library/typing.html#typing.Dict
[239] https://docs.python.org/3.8/library/typing.html#typing.Dict
[240] https://docs.python.org/3.8/library/typing.html#typing.Tuple
[241] https://docs.python.org/3.8/library/typing.html#typing.Dict
[242] https://docs.python.org/3.8/library/typing.html#typing.SupportsFloat

Warning: In previous versions header's names must follow CGI specification[243] (e.g., `CONTENT_TYPE`, `GATEWAY_INTERFACE`, `HTTP_*`)

Since version 5.3 and after migrate to Django 4 names must follow HTTP specification instead of CGI.

In any request parameter you can insert result value of previous operations (`<<{OPERATION_NUMBER or LET_VALUE}[path][to][value]>>`), for example:

```
[
    {"method": "post", "path": "user", "data": {"name": "User 1"}),
    {"method": "delete", "version": "v2", "path": ["user", "<<0[data][id]>>"]}
]
```

Result of bulk request is json list of objects for operation:

- `method` - http method
- `path` - path of request, always str
- `data` - data that needs to be sent
- `status` - response status code

Transactional bulk request returns `502 BAG GATEWAY` and does rollback after first failed request.

Warning: If you send non-transactional bulk request, you will get `200` status and must validate statuses on each operation responses.

3.4.2 OpenAPI schema

Request on `GET /{API_URL}/endpoint/` returns Swagger UI.

Request on `GET /{API_URL}/endpoint/?format=openapi` returns OpenAPI schema in json format. Also you can specify required version of schema using `version` query parameter (e.g., `GET /{API_URL}/endpoint/?format=openapi&version=v2`).

To change the schema after generating and before sending to user use hooks. Define one or more function, each taking 2 named arguments:

- `request` - user request object.
- `schema` - ordered dict with OpenAPI schema.

Note: Sometimes hooks may raise an exception; in order to keep a chain of data modification, such exceptions are handled. The changes made to the schema before the exception however, are saved.

Example hook:

```
def hook_add_username_to_guiname(request, schema):
    schema['info']['title'] = f"{request.username} - {schema['info']['title']}"
```

To connect hook(s) to your app add function import name to the `OPENAPI_HOOKS` list in `settings.py`

[243] https://www.w3.org/CGI/

```
OPENAPI_HOOKS = [
    '{{appName}}.openapi.hook_add_username_to_guiname',
]
```

3.5 Testing Framework

VST Utils Framework includes a helper in base test case class and improves support for making API requests. That means if you want make bulk request to endpoint you don't need create and init test client, but just need to call:

```
endpoint_results = self.bulk([
    # list of endpoint requests
])
```

3.5.1 Creating test case

`test.py` module contains test case classes based on *vstutils.tests.BaseTestCase*. At the moment, we officially support two styles of writing tests: classic and simple query wrappers with run check and runtime optimized bulk queries with manual value checking.

3.5.2 Simple example with classic tests

For example, if you have api endpoint like `/api/v1/project/` and model Project you can write test case like this:

```
from vstutils.tests import BaseTestCase


class ProjectTestCase(BaseTestCase):
    def setUp(self):
        super(ProjectTestCase, self).setUp()
        # init demo project
        self.initial_project = self.get_model_class('project.Test').objects.
↪create(name="Test")

    def tearDown(self)
        super(ProjectTestCase, self).tearDown()
        # remove it after test
        self.initial_project.delete()

    def test_project_endpoint(self):
        # Test checks that api returns valid values
        self.list_test('/api/v1/project/', 1)
        self.details_test(
            ["project", self.initial_project.id],
            name=self.initial_project.name
        )
        # Try to create new projects and check list endpoint
        test_data = [
            {"name": f"TestProject{i}"}
            for i in range(2)
        ]
        id_list = self.mass_create("/api/v1/project/", test_data, 'name')
        self.list_test('/api/v1/project/', 1 + len(id_list))
```

This example demonstrates functionality of default test case class. Default projects are initialized for the fastest and most efficient result. We recommend to divide tests for different entities into different classes. This example demonstrate classic style of testing, but you can use bulks in your test cases.

3.5.3 Bulk requests in tests

Bulk query system is well suited for testing and executing valid queries. Previous example could be rewritten as follows:

```
from vstutils.tests import BaseTestCase


class ProjectTestCase(BaseTestCase):
    def setUp(self):
        super(ProjectTestCase, self).setUp()
        # init demo project
        self.initial_project = self.get_model_class('project.Test').objects.
↪create(name="Test")

    def tearDown(self)
        super(ProjectTestCase, self).tearDown()
        # remove it after test
        self.initial_project.delete()

    def test_project_endpoint(self):
        test_data = [
            {"name": f"TestProject{i}"}
            for i in range(2)
        ]
        bulk_data = [
            {"method": "get", "path": ["project"]},
            {"method": "get", "path": ["project", self.initial_project.id]}
        ]
        bulk_data += [
            {"method": "post", "path": ["project"], "data": i}
            for i in test_data
        ]
        bulk_data.append(
            {"method": "get", "path": ["project"]}
        )
        results = self.bulk_transactional(bulk_data)

        self.assertEqual(results[0]['status'], 200)
        self.assertEqual(results[0]['data']['count'], 1)
        self.assertEqual(results[1]['status'], 200)
        self.assertEqual(results[1]['data']['name'], self.initial_project.name)

        for pos, result in enumerate(results[2:-1]):
            self.assertEqual(result['status'], 201)
            self.assertEqual(result['data']['name'], test_data[pos]['name'])

        self.assertEqual(results[-1]['status'], 200)
        self.assertEqual(results[-1]['data']['count'], 1 + len(test_data))
```

In this case, you have more code, but your tests are closer to GUI workflow, because vstutils-projects uses `/api/endpoint/` for requests. Either way, bulk queries are much faster due to optimization; Testcase execution time is less comparing to non-bulk requests.

3.5.4 Test case API

class vstutils.tests.**BaseTestCase**(*methodName='runTest'*)

Main test case class extends `django.test.TestCase`[244].

assertCheckDict(*first*, *second*, *msg=None*)

Fail if the two fields in dicts are unequal as determined by the '==' operator. Checks if first not contains or not equal field in second

Parameters

- **first** (`typing.Dict`[245]) –
- **second** (`typing.Dict`[246]) –
- **msg** (`str`[247]) –

assertCount(*iterable*, *count*, *msg=None*)

Calls `len()`[248] over `iterable` and checks equality with `count`.

Parameters

- **iterable** (`typing.Sized`[249]) – any iterable object which could be sended to `len()`[250].
- **count** (`int`[251]) – expected result.
- **msg** (`typing.Any`[252]) – error message

assertRCode(*resp*, *code=200*, **additional_info*)

Fail if response code is not equal. Message is response body.

Parameters

- **resp** (*`django.http.HttpResponse`*[253]) – response object
- **code** (*`int`*[254]) – expected code

bulk(*data*, *code=200*, ***kwargs*)

Makes non-transactional bulk request and asserts status code (default is 200)

Parameters

- **data** (`typing.Union`[255][`typing.List`[256][`typing.Dict`[257][`str`[258], `typing.Any`[259]]], `str`[260], `bytes`[261], `bytearray`[262]) – request data
- **code** (`int`[263]) – http status to assert
- **kwargs** – named arguments for *`get_result()`*

Return type

`typing.Union`[264][`typing.List`[265][`typing.Dict`[266][`str`[267], `typing.Any`[268]]], `str`[269], `bytes`[270], `bytearray`[271], `typing.Dict`[272], `typing.Sequence`[273][`typing.Union`[274][`typing.List`[275][`typing.Dict`[276][`str`[277], `typing.Any`[278]]], `str`[279], `bytes`[280], `bytearray`[281]]]]

Returns

bulk response

bulk_transactional(*data*, *code=200*, ***kwargs*)

Make transactional bulk request and assert status code (default is 200)

Parameters

- **data** (typing.Union[282][typing.List[283][typing.Dict[284][str[285], typing.Any[286]]], str[287], bytes[288], bytearray[289]) – request data
- **code** (int[290]) – http status to assert
- **kwargs** – named arguments for *get_result()*

Return type
typing.Union[291][typing.List[292][typing.Dict[293][str[294], typing.Any[295]]], str[296], bytes[297], bytearray[298], typing.Dict[299], typing.Sequence[300][typing.Union[301][typing.List[302][typing.Dict[303][str[304], typing.Any[305]]], str[306], bytes[307], bytearray[308]]]

Returns
bulk response

call_registration(*data*, ***kwargs*)

Function for calling registration. Just got form data and headers.

Parameters

- **data** (*dict*[309]) – Registration form data.
- **kwargs** – named arguments with request headers.

details_test(*url*, ***kwargs*)

Test for get details of model. If you setup additional named arguments, the method check their equality with response data. Uses *get_result()* method.

Parameters

- **url** – url to detail record. For example: `/api/v1/project/1/` (where `1` is uniq id of project). You can use *get_url()* for building url.
- **kwargs** – params that's should be checked (key - field name, value - field value).

endpoint_call(*data=None*, *method='get'*, *code=200*, ***kwargs*)

Makes request to endpoint and asserts response status code if specified (default is 200). Uses *get_result()* method for execution.

Parameters

- **data** (typing.Union[310][typing.List[311][typing.Dict[312][str[313], typing.Any[314]]], str[315], bytes[316], bytearray[317]) – request data
- **method** (str[318]) – http request method
- **code** (int[319]) – http status to assert
- **query** – dict with query data (working only with *get*)

Return type
typing.Union[320][typing.List[321][typing.Dict[322][str[323], typing.Any[324]]], str[325], bytes[326], bytearray[327], typing.Dict[328], typing.Sequence[329][typing.Union[330][typing.List[331][typing.Dict[332][str[333], typing.Any[334]]], str[335], bytes[336], bytearray[337]]]

Returns
bulk response

endpoint_schema(***kwargs*)

Make request to schema. Returns dict with swagger data.

Parameters
version – API version for schema parser.

get_count(*model*, ***kwargs*)

Simple wrapper over *get_model_filter()* which returns counter of items.

Parameters

- **model** (*str*[338], *django.db.models.Model*[339]) – string which contains model name (if attribute `model` is set to the test case class), module import, `app.ModelName` or `django.db.models.Model`[340].
- **kwargs** – named arguments to `django.db.models.query.QuerySet.filter()`[341].

Returns
number of instances in database.

Return type
int[342]

get_model_class(*model*)

Getting model class by string or return model arg.

Parameters
model (*str*[343], *django.db.models.Model*[344]) – string which contains model name (if attribute `model` is set to the test case class), module import, `app.ModelName` or `django.db.models.Model`[345].

Returns
Model class.

Return type
django.db.models.Model[346]

get_model_filter(*model*, ***kwargs*)

Simple wrapper over *get_model_class()* which returns filtered queryset from model.

Parameters

- **model** (*str*[347], *django.db.models.Model*[348]) – string which contains model name (if attribute `model` is set to the test case class), module import, `app.ModelName` or `django.db.models.Model`[349].
- **kwargs** – named arguments to `django.db.models.query.QuerySet.filter()`[350].

Return type
django.db.models.query.QuerySet[351]

get_result(*rtype*, *url*, *code=None*, **args*, ***kwargs*)

Executes and tests response code on request with returning parsed result of request. The method uses the following procedure:

- Test client authorization (with `user` which creates in `setUp()`).
- Executing a request (sending args and kwargs to request method).
- Parsing the result (converts json string to python-object).
- Checking the http status code with *assertRCode()* (if you have not specified it, the code will be selected in accordance with the request method from the standard set *std_codes*).

- Logout client.
- Return parsed result.

Parameters

- **rtype** – request type (methods from Client cls): get, post etc.
- **url** – requested url string or tuple for *get_url()*. You can use *get_url()* for url building or setup it as full string.
- **code** (int[352]) – expected return code from request.
- **relogin** – execute force login and logout on each call. Default is True.
- **args** – extra-args for Client class request method.
- **kwargs** – extra-kwargs for Client class request method.

Return type
typing.Union[353][typing.List[354][typing.Dict[355][str[356], typing.Any[357]]], str[358], bytes[359], bytearray[360], typing.Dict[361], typing.Sequence[362][typing.Union[363][typing.List[364][typing.Dict[365][str[366], typing.Any[367]]], str[368], bytes[369], bytearray[370]]]

Returns
result of request.

get_url (**items*)

Function for creating url path based on VST_API_URL and VST_API_VERSION settings. Without arguments returns path to default version of api.

Return type
str[371]

Returns
string like /api/v1/.../.../ where ... is args of function.

list_test (*url*, *count*)

Test for get list of models. Checks only list count. Uses *get_result()* method.

Parameters

- **url** – url to abstract layer. For example: /api/v1/project/. You can use *get_url()* for building url.
- **count** – count of objects in DB.

models = None

Attribute with default project models module.

classmethod patch (**args*, ***kwargs*)

Simple unittest.mock.patch()[372] class-method wrapper.

Return type
typing.ContextManager[373][unittest.mock.Mock[374]]

classmethod patch_field_default (*model*, *field_name*, *value*)

This method helps to path default value in the model's field. It's very useful for DateTime fields where django.utils.timezone.now()[375] is used in defaults.

Parameters

- **model** (django.db.models.base.Model) –
- **field_name** (str[376]) –
- **value** (typing.Any[377]) –

Return type
typing.ContextManager[378][unittest.mock.Mock[379]]

random_name()

Simple function which returns uuid1 string.

Return type
str[380]

std_codes: typing.Dict[381][str[382], int[383]] = {'delete': 204, 'get': 200, 'patch': 200, 'post': 201}

Default http status codes for different http methods. Uses in *get_result()*

class user_as(*testcase*, *user*)

Context for execute bulk or something as user. The context manager overrides self.user in TestCase and revert this changes on exit.

Parameters
user (*django.contrib.auth.models.AbstractUser*[384]) – new user object for execution.

[244] https://docs.djangoproject.com/en/4.2/topics/testing/tools/#django.test.TestCase
[245] https://docs.python.org/3.8/library/typing.html#typing.Dict
[246] https://docs.python.org/3.8/library/typing.html#typing.Dict
[247] https://docs.python.org/3.8/library/stdtypes.html#str
[248] https://docs.python.org/3.8/library/functions.html#len
[249] https://docs.python.org/3.8/library/typing.html#typing.Sized
[250] https://docs.python.org/3.8/library/functions.html#len
[251] https://docs.python.org/3.8/library/functions.html#int
[252] https://docs.python.org/3.8/library/typing.html#typing.Any
[253] https://docs.djangoproject.com/en/4.2/ref/request-response/#django.http.HttpResponse
[254] https://docs.python.org/3.8/library/functions.html#int
[255] https://docs.python.org/3.8/library/typing.html#typing.Union
[256] https://docs.python.org/3.8/library/typing.html#typing.List
[257] https://docs.python.org/3.8/library/typing.html#typing.Dict
[258] https://docs.python.org/3.8/library/stdtypes.html#str
[259] https://docs.python.org/3.8/library/typing.html#typing.Any
[260] https://docs.python.org/3.8/library/stdtypes.html#str
[261] https://docs.python.org/3.8/library/stdtypes.html#bytes
[262] https://docs.python.org/3.8/library/stdtypes.html#bytearray
[263] https://docs.python.org/3.8/library/functions.html#int
[264] https://docs.python.org/3.8/library/typing.html#typing.Union
[265] https://docs.python.org/3.8/library/typing.html#typing.List
[266] https://docs.python.org/3.8/library/typing.html#typing.Dict
[267] https://docs.python.org/3.8/library/stdtypes.html#str
[268] https://docs.python.org/3.8/library/typing.html#typing.Any
[269] https://docs.python.org/3.8/library/stdtypes.html#str
[270] https://docs.python.org/3.8/library/stdtypes.html#bytes
[271] https://docs.python.org/3.8/library/stdtypes.html#bytearray
[272] https://docs.python.org/3.8/library/typing.html#typing.Dict
[273] https://docs.python.org/3.8/library/typing.html#typing.Sequence
[274] https://docs.python.org/3.8/library/typing.html#typing.Union
[275] https://docs.python.org/3.8/library/typing.html#typing.List
[276] https://docs.python.org/3.8/library/typing.html#typing.Dict
[277] https://docs.python.org/3.8/library/stdtypes.html#str
[278] https://docs.python.org/3.8/library/typing.html#typing.Any
[279] https://docs.python.org/3.8/library/stdtypes.html#str
[280] https://docs.python.org/3.8/library/stdtypes.html#bytes
[281] https://docs.python.org/3.8/library/stdtypes.html#bytearray
[282] https://docs.python.org/3.8/library/typing.html#typing.Union
[283] https://docs.python.org/3.8/library/typing.html#typing.List
[284] https://docs.python.org/3.8/library/typing.html#typing.Dict
[285] https://docs.python.org/3.8/library/stdtypes.html#str
[286] https://docs.python.org/3.8/library/typing.html#typing.Any
[287] https://docs.python.org/3.8/library/stdtypes.html#str
[288] https://docs.python.org/3.8/library/stdtypes.html#bytes
[289] https://docs.python.org/3.8/library/stdtypes.html#bytearray
[290] https://docs.python.org/3.8/library/functions.html#int
[291] https://docs.python.org/3.8/library/typing.html#typing.Union
[292] https://docs.python.org/3.8/library/typing.html#typing.List
[293] https://docs.python.org/3.8/library/typing.html#typing.Dict
[294] https://docs.python.org/3.8/library/stdtypes.html#str
[295] https://docs.python.org/3.8/library/typing.html#typing.Any
[296] https://docs.python.org/3.8/library/stdtypes.html#str
[297] https://docs.python.org/3.8/library/stdtypes.html#bytes
[298] https://docs.python.org/3.8/library/stdtypes.html#bytearray
[299] https://docs.python.org/3.8/library/typing.html#typing.Dict
[300] https://docs.python.org/3.8/library/typing.html#typing.Sequence
[301] https://docs.python.org/3.8/library/typing.html#typing.Union
[302] https://docs.python.org/3.8/library/typing.html#typing.List
[303] https://docs.python.org/3.8/library/typing.html#typing.Dict
[304] https://docs.python.org/3.8/library/stdtypes.html#str
[305] https://docs.python.org/3.8/library/typing.html#typing.Any
[306] https://docs.python.org/3.8/library/stdtypes.html#str
[307] https://docs.python.org/3.8/library/stdtypes.html#bytes
[308] https://docs.python.org/3.8/library/stdtypes.html#bytearray
[309] https://docs.python.org/3.8/library/stdtypes.html#dict
[310] https://docs.python.org/3.8/library/typing.html#typing.Union
[311] https://docs.python.org/3.8/library/typing.html#typing.List
[312] https://docs.python.org/3.8/library/typing.html#typing.Dict
[313] https://docs.python.org/3.8/library/stdtypes.html#str
[314] https://docs.python.org/3.8/library/typing.html#typing.Any
[315] https://docs.python.org/3.8/library/stdtypes.html#str
[316] https://docs.python.org/3.8/library/stdtypes.html#bytes
[317] https://docs.python.org/3.8/library/stdtypes.html#bytearray

3.6 Utils

This is tested set of development utilities. Utilities include a collection of code that will be useful in one way or another for developing the application. Vstutils uses mosts of these functions under the hood.

class vstutils.utils.**BaseEnum**(*value*, *names=None*, ***, *module=None*, *qualname=None*, *type=None*, *start=1*, *boundary=None*)

BaseEnum extends *Enum* class and used to create enum-like objects that can be used in django serializers or django models.

Example:

```
from vstutils.models import BModel

class ItemCLasses(BaseEnum):
    FIRST = BaseEnum.SAME
    SECOND = BaseEnum.SAME
    THIRD = BaseEnum.SAME


class MyDjangoModel(BModel):
    item_class = models.CharField(max_length=ItemCLasses.max_len,␣
↪choices=ItemCLasses.to_choices())

    @property
    def is_second(self):
        # Function check is item has second class of instance
        return ItemCLasses.SECOND.is_equal(self.item_class)
```

Note: For special cases, when value must be in lower or upper case, you can setup value as `BaseEnum.LOWER`` or ``BaseEnum.UPPER`. But in default cases we recommend use `BaseEnum.SAME` for memory optimization.

class vstutils.utils.**BaseVstObject**

Default mixin-class for custom objects which needed to get settings and cache.

classmethod get_django_settings(*name*, *default=None*)

Get params from Django settings.

Parameters

- **name** (*str*[385]) – name of param
- **default** (*object*[386]) – default value of param

Returns

Param from Django settings or default.

class vstutils.utils.**Dict**

Wrapper over *dict* which return JSON on conversion to string.

class vstutils.utils.**Executor**(*stdout=-1*, *stderr=-2*, ***environ_variables*)

Command executor with realtime output write and line handling. By default and by design executor initialize string

[385] https://docs.python.org/3.8/library/stdtypes.html#str
[386] https://docs.python.org/3.8/library/functions.html#object

attribute `output` which will be modified by += operator with new lines by *Executor.write_output()* procedure. Override the method if you want change behavior.

Executor class supports periodically (0.01 sec) handling process and execute some checks by overriding *Executor.working_handler()* procedure method. If you want disable this behavior override the method by None value or use *UnhandledExecutor*.

Parameters
environ_variables (str[387]) –

exception CalledProcessError(*returncode*, *cmd*, *output=None*, *stderr=None*)

Raised when run() is called with check=True and the process returns a non-zero exit status.

Attributes:
cmd, returncode, stdout, stderr, output

property stdout

Alias for output attribute, to match stderr

async aexecute(*cmd*, *cwd*, *env=None*)

Executes commands and outputs its result. Asynchronous implementation.

Parameters

- **cmd** – – list of cmd command and arguments
- **cwd** – – workdir for executions
- **env** – – extra environment variables which overrides defaults

Returns
– string with full output

execute(*cmd*, *cwd*, *env=None*)

Executes commands and outputs its result.

Parameters

- **cmd** – – list of cmd command and arguments
- **cwd** – – workdir for executions
- **env** – – extra environment variables which overrides defaults

Returns
– string with full output

async post_execute(*cmd*, *cwd*, *env*, *return_code*)

Runs after execution end.

Parameters

- **cmd** – – list of cmd command and arguments
- **cwd** – – workdir for executions
- **env** – – extra environment variables which overrides defaults
- **return_code** – – return code of executed process

async pre_execute(*cmd*, *cwd*, *env*)

Runs before execution starts.

Parameters

- **cmd** – – list of cmd command and arguments

- **cwd** – – workdir for executions
- **env** – – extra environment variables which overrides defaults

async working_handler(*proc*)

Additional handler for executions.

Parameters
proc (*asyncio.subprocess.Process*) – running process

write_output(*line*)

Parameters
line (*str*[388]) – – line from command output

Returns
None

Return type
None

class vstutils.utils.**KVExchanger**(*key*, *timeout=None*)

Class for transmit data using key-value fast (cache-like) storage between services. Uses same cache-backend as Lock.

class vstutils.utils.**Lock**(*id*, *payload=1*, *repeat=1*, *err_msg=''*, *timeout=None*)

Lock class for multi-jobs workflow. Based on *KVExchanger*. The Lock allows only one thread to enter the part that's locked and shared between apps using one locks cache (see also [locks]).

Parameters

- **id** (*int*[389], *str*[390]) – – unique id for lock.
- **payload** – – lock additional info. Should be any boolean True value.
- **repeat** (*int*[391]) – – time to wait lock.release. Default 1 sec.
- **err_msg** (*str*[392]) – – message for AcquireLockException error.

Note:

- Used django.core.cache lib and settings in *settings.py*
- Have Lock.SCHEDULER and Lock.GLOBAL id

Example:

```
from vstutils.utils import Lock

with Lock("some_lock_identifier", repeat=30, err_msg="Locked by another␣
↪proccess") as lock:
    # where
    # ``"some_lock_identifier"`` is unique id for lock and
    # ``30`` seconds lock is going wait until another process will release␣
↪lock id.
    # After 30 sec waiting lock will raised with :class:`.Lock.
↪AcquireLockException`
```

(continues on next page)

[387] https://docs.python.org/3.8/library/stdtypes.html#str
[388] https://docs.python.org/3.8/library/stdtypes.html#str

(continued from previous page)

```
    # and ``err_msg`` value as text.
    some_code_execution()
    # ``lock`` object will has been automatically released after
    # exiting from context.
```

Another example without context manager:

```
from vstutils.utils import Lock

# locked block after locked object created
lock = Lock("some_lock_identifier", repeat=30, err_msg="Locked by another␣
↪process")
# deleting of object calls ``lock.release()`` which release and remove lock␣
↪from id.
del lock
```

exception AcquireLockException

Exception which will be raised on unreleased lock.

class vstutils.utils.**ModelHandlers**(*type_name*, *err_message=None*)

Handlers for some models like 'INTEGRATIONS' or 'REPO_BACKENDS'. Based on *ObjectHandlers* but more specific for working with models. All handlers backends get by first argument model object.

Attributes:

Parameters

- **objects** (*dict*[393]) – – dict of objects like: {<name>: <backend_class>}
- **keys** (*list*[394]) – – names of supported backends
- **values** (*list*[395]) – – supported backends classes
- **type_name** – type name for backends.Like name in dict.

get_object(*name*, *obj*)

Parameters

- **name** – – string name of backend
- **name** – str
- **obj** (*django.db.models.Model*[396]) – – model object

Returns

backend object

Return type

object[397]

[389] https://docs.python.org/3.8/library/functions.html#int
[390] https://docs.python.org/3.8/library/stdtypes.html#str
[391] https://docs.python.org/3.8/library/functions.html#int
[392] https://docs.python.org/3.8/library/stdtypes.html#str
[393] https://docs.python.org/3.8/library/stdtypes.html#dict
[394] https://docs.python.org/3.8/library/stdtypes.html#list
[395] https://docs.python.org/3.8/library/stdtypes.html#list
[396] https://docs.djangoproject.com/en/4.2/ref/models/instances/#django.db.models.Model
[397] https://docs.python.org/3.8/library/functions.html#object

class vstutils.utils.**ObjectHandlers**(*type_name*, *err_message=None*)

Handlers wrapper for get objects from some settings structure.

Example:

```
from vstutils.utils import ObjectHandlers

'''
In `settings.py` you should write some structure:

SOME_HANDLERS = {
    "one": {
        "BACKEND": "full.python.path.to.module.SomeClass"
    },
    "two": {
        "BACKEND": "full.python.path.to.module.SomeAnotherClass",
        "OPTIONS": {
            "some_named_arg": "value"
        }
    }
}
'''

handlers = ObjectHandlers('SOME_HANDLERS')

# Get class handler for 'one'
one_backend_class = handlers['one']
# Get object of backend 'two'
two_obj = handlers.get_object()
# Get object of backend 'two' with overriding constructor named arg
two_obj_overrided = handlers.get_object(some_named_arg='another_value')
```

Parameters

type_name (*str*[398]) – type name for backends.Like name in dict.

backend(*name*)

Get backend class

Parameters

name (*str*[399]) – – name of backend type

Returns

class of backend

Return type

type[400],types.ModuleType[401],object[402]

class vstutils.utils.**Paginator**(*qs*, *chunk_size=None*)

Class for fragmenting the query for small queries.

class vstutils.utils.**SecurePickling**(*secure_key=None*)

Secured pickle wrapper by Vigenère cipher.

[398] https://docs.python.org/3.8/library/stdtypes.html#str
[399] https://docs.python.org/3.8/library/stdtypes.html#str
[400] https://docs.python.org/3.8/library/functions.html#type
[401] https://docs.python.org/3.8/library/types.html#types.ModuleType
[402] https://docs.python.org/3.8/library/functions.html#object

> **Warning:** Do not use it with untrusted transport anyway.

Example:

```
from vstutils.utils import SecurePickling


serializer = SecurePickling('password')

# Init secret object
a = {"key": "value"}
# Serialize object with secret key
pickled = serializer.dumps(a)
# Deserialize object
unpickled = serializer.loads(pickled)

# Check, that object is correct
assert a == unpickled
```

class vstutils.utils.**URLHandlers**(*type_name='URLS'*, **args*, ***kwargs*)

Object handler for GUI views. Uses *GUI_VIEWS* from settings.py. Based on *ObjectHandlers* but more specific to urlpatterns.

Example:

```
from vstutils.utils import URLHandlers


# By default gets from `GUI_VIEWS` in `settings.py`
urlpatterns = list(URLHandlers())
```

Parameters
type_name – type name for backends.Like name in dict.

get_object(*name*, **argv*, ***kwargs*)

Get url object tuple for urls.py

Parameters

- **name** (*str*[403]) – url regexp from
- **argv** – overridden args
- **kwargs** – overridden kwargs

Returns
url object

Return type
django.urls.re_path

class vstutils.utils.**UnhandledExecutor**(*stdout=-1*, *stderr=-2*, ***environ_variables*)

Class based on *Executor* but disables *working_handler*.

Parameters
environ_variables (*str*[404]) –

[403] https://docs.python.org/3.8/library/stdtypes.html#str

class vstutils.utils.**apply_decorators**(**decorators*)

Decorator which apply list of decorators on method or class.

Example:

```
from vstutils.utils import apply_decorators

def decorator_one(func):
    print(f"Decorated {func.__name__} by first decorator.")
    return func

def decorator_two(func):
    print(f"Decorated {func.__name__} by second decorator.")
    return func

@apply_decorators(decorator_one, decorator_two)
def decorated_function():
    # Function decorated by both decorators.
    print("Function call.")
```

class vstutils.utils.**classproperty**(*fget*, *fset=None*)

Decorator which makes class method as class property.

Example:

```
from vstutils.utils import classproperty

class SomeClass(metaclass=classproperty.meta):
    # Metaclass is needed for set attrs in class
    # instead of and not only object.

    some_value = None

    @classproperty
    def value(cls):
        return cls.some_value

    @value.setter
    def value(cls, new_value):
        cls.some_value = new_value
```

Parameters

- **fget** – function for getting an attribute value.
- **fset** – function for setting an attribute value.

vstutils.utils.**create_view**(*model*, ***meta_options*)

A simple function for getting the generated view by standard means, but with overloaded meta-parameters. This method can completely get rid of the creation of proxy models.

Example:

```
from vstutils.utils import create_view

from .models import Host
```

(continues on next page)

[404] https://docs.python.org/3.8/library/stdtypes.html#str

(continued from previous page)

```
# Host model has full :class:`vstutils.api.base.ModelViewSet` view.
# For overriding and create simple list view just setup this:
HostListViewSet = create_view(
    HostList,
    view_class='list_only'
)
```

Note: This method is also recommended in cases where there is a problem of recursive imports.

Warning: This function is oldstyle and will be deprecated in future versions. Use native call of method **:method:`vstutils.models.BModel.get_view_class`**.

Parameters

model (*Type[vstutils.models.BaseModel]*) – Model class with *.get_view_class* method. This method also has *vstutils.models.BModel*.

Return type

vstutils.api.base.GenericViewSet

vstutils.utils.**decode**(*key*, *enc*)

Decode string from encoded by Vigenère cipher.

Parameters

- **key** (*str*[405]) – – secret key for encoding
- **enc** (*str*[406]) – – encoded string for decoding

Returns

– decoded string

Return type

str[407]

vstutils.utils.**deprecated**(*func*)

This is a decorator which can be used to mark functions as deprecated. It will result in a warning being emitted when the function is used.

Parameters

func – any callable that will be wrapped and will issue a deprecation warning when called.

vstutils.utils.**encode**(*key*, *clear*)

Encode string by Vigenère cipher.

Parameters

- **key** (*str*[408]) – – secret key for encoding
- **clear** (*str*[409]) – – clear value for encoding

[405] https://docs.python.org/3.8/library/stdtypes.html#str
[406] https://docs.python.org/3.8/library/stdtypes.html#str
[407] https://docs.python.org/3.8/library/stdtypes.html#str

Returns
– encoded string

Return type
str[410]

vstutils.utils.**get_render**(*name*, *data*, *trans='en'*)

Render string from template.

Parameters

- **name** (*str*[411]) – – full template name
- **data** (*dict*[412]) – – dict of rendered vars
- **trans** (*str*[413]) – – translation for render. Default 'en'.

Returns
– rendered string

Return type
str[414]

vstutils.utils.**lazy_translate**(*text*)

The `lazy_translate` function has the same behavior as *translate()*, but wraps it in a lazy promise.

This is very useful, for example, for translating error messages in class attributes before the language code is known.

Parameters
text – Text message which should be translated.

vstutils.utils.**list_to_choices**(*items_list*, *response_type=<class 'list'>*)

Method to create django model choices from flat list of values.

Parameters

- **items_list** – list of flat values.
- **response_type** – casting type of returned mapping

Returns
list of tuples from *items_list* values

class vstutils.utils.**model_lock_decorator**(***kwargs*)

Decorator for functions where 'pk' kwarg exist for lock by id.

> **Warning:**
> - On locked error raised `Lock.AcquireLockException`
> - Method must have and called with `pk` named arg.

class vstutils.utils.**raise_context**(**args*, ***kwargs*)

Context for exclude exceptions.

[408] https://docs.python.org/3.8/library/stdtypes.html#str
[409] https://docs.python.org/3.8/library/stdtypes.html#str
[410] https://docs.python.org/3.8/library/stdtypes.html#str
[411] https://docs.python.org/3.8/library/stdtypes.html#str
[412] https://docs.python.org/3.8/library/stdtypes.html#dict
[413] https://docs.python.org/3.8/library/stdtypes.html#str
[414] https://docs.python.org/3.8/library/stdtypes.html#str

class vstutils.utils.**raise_context_decorator_with_default**(**args*, ***kwargs*)

Context for exclude errors and return default value.

Example:

```
from yaml import load
from vstutils.utils import raise_context_decorator_with_default


@raise_context_decorator_with_default(default={})
def get_host_data(yaml_path, host):
    with open(yaml_path, 'r') as fd:
        data = load(fd.read(), Loader=Loader)
    return data[host]
    # This decorator used when you must return some value even on error
    # In log you also can see traceback for error if it occur

def clone_host_data(host):
    bs_data = get_host_data('inventories/aws/hosts.yml', 'build_server')
    ...
```

class vstutils.utils.**redirect_stdany**(*new_stream=<_io.StringIO object>*, *streams=None*)

Context for redirect any output to own stream.

Note:

- On context returns stream object.
- On exit returns old streams.

---

vstutils.utils.**send_mail**(*subject*, *message*, *from_email*, *recipient_list*, *fail_silently=False*, *auth_user=None*, *auth_password=None*, *connection=None*, *html_message=None*, ***kwargs*)

Wrapper over `django.core.mail.send_mail()`[415] which provide additional named arguments.

vstutils.utils.**send_template_email**(*sync=False*, ***kwargs*)

Function executing sync or async email sending; according *sync* argument and settings variable "RPC_ENABLED". If you don't set settings for celery or don't have celery it sends synchronously mail. If celery is installed and configured and *sync* argument of the function is set to *False*, it sends asynchronously email.

Parameters

- **sync** – argument for determining how send email, asynchronously or synchronously
- **subject** – mail subject.
- **email** – list of strings or single string, with email addresses of recipients
- **template_name** – relative path to template in *templates* directory, must include extension in file name.
- **context_data** – dictionary with context for rendering message template.

vstutils.utils.**send_template_email_handler**(*subject*, *email_from*, *email*, *template_name*, *context_data=None*, ***kwargs*)

Function for email sending. The function convert recipient to list and set context before sending if it possible.

Parameters

[415] https://docs.djangoproject.com/en/4.2/topics/email/#django.core.mail.send_mail

- **subject** – mail subject.
- **email_from** – sender that be setup in email.
- **email** – list of strings or single string, with email addresses of recipients
- **template_name** – relative path to template in *templates* directory, must include extension in file name.
- **context_data** – dictionary with context for rendering message template.
- **kwargs** – additional named arguments for *send_mail*

Returns
Number of emails sent.

class vstutils.utils.**tmp_file**(*data=''*, *mode='w'*, *bufsize=-1*, ***kwargs*)

Temporary file with name generated and auto removed on close.

Attributes:

Parameters

- **data** (*str*[416]) – – string to write in tmp file.
- **mode** (*str*[417]) – – file open mode. Default 'w'.
- **bufsize** (*int*[418]) – – buffer size for tempfile.NamedTemporaryFile
- **kwargs** – – other kwargs for tempfile.NamedTemporaryFile

write(*wr_string*)

Write to file and flush

Parameters
wr_string (*str*[419]) – – writable string

Returns
None

Return type
None

class vstutils.utils.**tmp_file_context**(**args*, ***kwargs*)

Context object for work with tmp_file. Auto close on exit from context and remove if file still exist.

This context manager over *tmp_file*

vstutils.utils.**translate**(*text*)

The `translate` function supports translation message dynamically with standard i18n vstutils'es mechanisms usage.

Uses `django.utils.translation.get_language()`[420] to get the language code and tries to get the translation from the list of available ones.

Parameters
text – Text message which should be translated.

[416] https://docs.python.org/3.8/library/stdtypes.html#str
[417] https://docs.python.org/3.8/library/stdtypes.html#str
[418] https://docs.python.org/3.8/library/functions.html#int
[419] https://docs.python.org/3.8/library/stdtypes.html#str
[420] https://docs.djangoproject.com/en/4.2/ref/utils/#django.utils.translation.get_language

3.7 Integrating Web Push Notifications

Web push notifications are an effective way to engage users with real-time messaging. To integrate web push notifications in your VSTUtils project, follow these steps:

1. **Configuration**: First, include the `vstutils.webpush` module in the `INSTALLED_APPS` section of your `settings.py` file. This enables the web push functionality provided by VSTUtils. Additionally, configure the necessary settings as described in the web push settings section (see *here* for details).
2. **Creating Notifications**: To create a web push notification, you need to define a class that inherits from either `vstutils.webpush.BaseWebPush` or `vstutils.webpush.BaseWebPushNotification`. VSTUtils automatically detects and utilizes web push classes defined in the `webpushes` module of all `INSTALLED_APPS`. Below is an example that illustrates how to implement custom web push classes:

```python
from vstutils.api.models import Language
from vstutils.webpush.base import BaseWebPush, BaseWebPushNotification
from vstutils.webpush.models import WebPushDeviceSubscription, WebPushNotificationSubscription


class TestWebPush(BaseWebPush):
    """
    Webpush that is sent to all subscribed users
    """

    def get_subscriptions(self):
        return WebPushDeviceSubscription.objects.filter(
            user_id__in=WebPushNotificationSubscription.objects.filter(
                type=self.get_key(),
                enabled=True,
            ).values('user_id'),
        )

    def get_payload(self, lang: Language):
        return {"some": "data", "lang": lang.code}


class TestNotification(BaseWebPushNotification):
    """
    Webpush notification that is sent only to selected users
    """

    def __init__(self, name: str, user_id: int):
        self.name = name
        self.user_id = user_id
        self.message = f"Hello {self.name}"

    def get_users_ids(self):
        return (self.user_id,)

    def get_notification(self, lang: Language):
        return {
            "title": self.message,
            "options": {
                "body": "Test notification body",
                "data": {"url": "/"},
            },
```

(continues on next page)

(continued from previous page)

```
        }


class StaffOnlyNotification(BaseWebPushNotification):
    """
    Webpush notification that only staff user can subscribe to.
    """

    @staticmethod
    def is_available(user):
        return user.is_staff
```

This example contains three classes:

- *TestWebPush*: Sends notifications to all subscribed users.
- *TestNotification*: Targets notifications to specific users.
- *StaffOnlyNotification*: Restricts notifications to staff users only. Sometimes you may want to allow only some users to subscribe on specific notifications.

3. **Sending Notifications**: To dispatch a web push notification, invoke the `send` or `send_in_task` method on an instance of your web push class. For instance, to send a notification using *TestNotification*, you can do the following:

```
from test_proj.webpushes import TestNotification

# Sending a notification immediately (synchronously)
TestNotification(name='Some user', user_id=1).send()

# Sending a notification as a background task (asynchronously)
TestNotification.send_in_task(name='Some user', user_id=1)
```

Warning: The asynchronous sending of web push notifications (using methods like `send_in_task`) requires a configured Celery setup in your project, as it relies on Celery tasks "under the hood". Ensure that Celery is properly set up and running to utilize asynchronous notification dispatching.

By following these steps, you can fast integrate and utilize web push notifications in projects with VSTUtils.

4

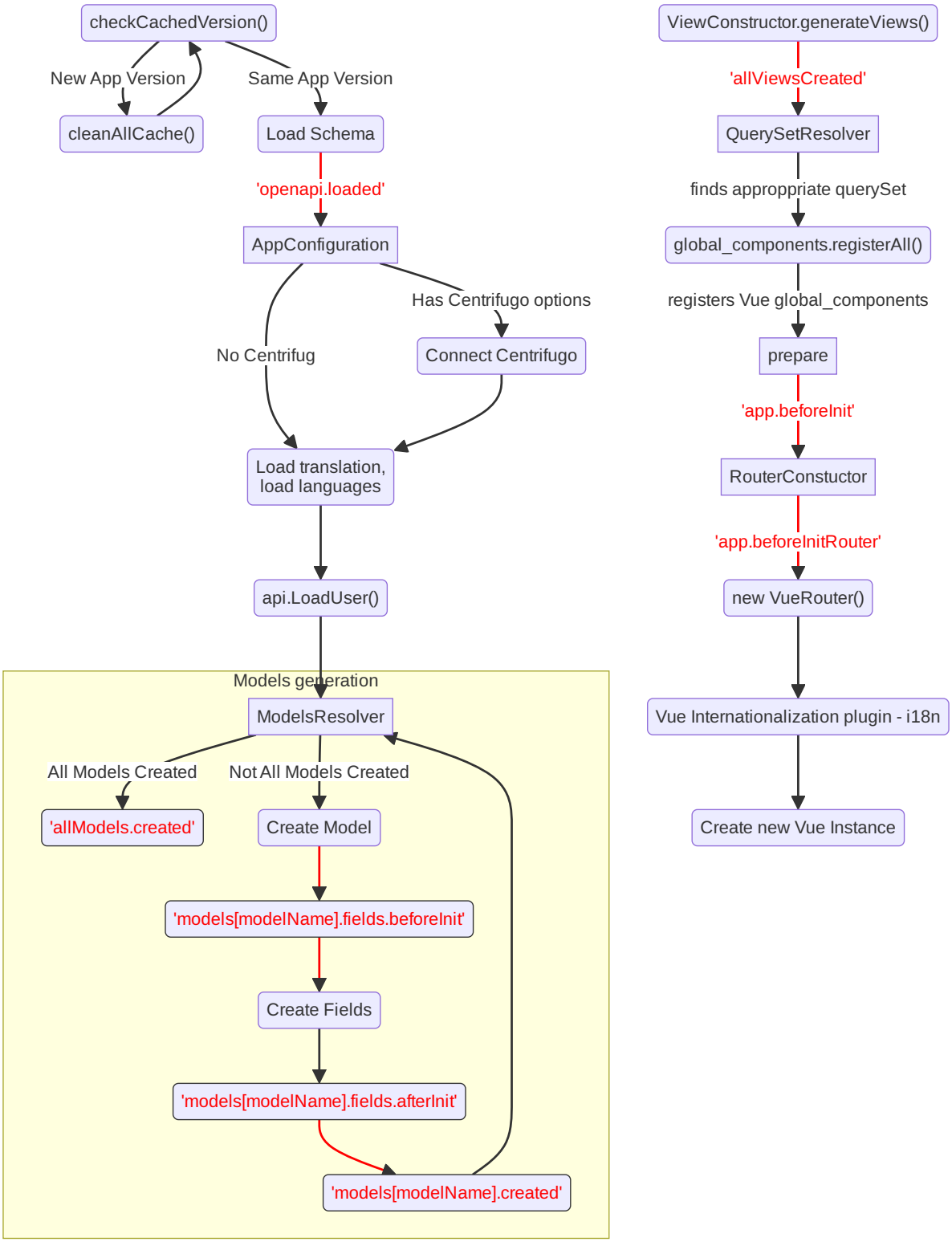
Frontend Quickstart

VST utils framework uses Vue ecosystem to render frontend. The quickstart manual will guide you through the most important steps to customize frontend features. App installation and setting up described in - *Backend Section* of this docs.

There are several stages in vstutils app:

1. Before app started:
   - *checkCacheVersions()* checks if app version has been changed since last visit and cleans all cached data if so;
   - loading open api schema from backend. Emits 'openapi.loaded' signal;
   - loading all static files from *SPA_STATIC* in setting.py;
   - sets *AppConfiguration* from OpenAPI schema;
2. App started:
   - if there is centrifugoClient in settings.py connects it. To read more about centrifugo configuration check "*Centrifugo client settings*";
   - downloading a list of available languages and translations;
   - *api.loadUser()* returns user data;
   - *ModelsResolver* creates models from schema, emits signal *models[${modelName}].created* for each created model and *allModels.created* when all models created;
   - *ViewConstructor.generateViews()* inits *View* fieldClasses and modelClasses;
   - *QuerySetsResolver* finds appropriate queryset by model name and view path;
   - *global_components.registerAll()* registers Vue *global_components*;
   - *prepare()* emits *app.beforeInit* with { app: this };
   - initialize model with *LocalSettings*. Find out more about this in the section *LocalSettings*;
   - creates routerConstructor from *this.views*, emits 'app.beforeInitRouter' with { routerConstructor } and gets new VueRouter({this.routes});
   - inits application *Vue()* from schema.info, pinia store and emits 'app.afterInit' with {app: this};
3. Application mounted.

There is a flowchart representing application initialization process (signal names have red font):



4.1 Field customization

To add custom script to the project, set script name in settings.py

```
SPA_STATIC += [
    {'priority': 101, 'type': 'js', 'name': 'main.js', 'source': 'project_lib'},
]
```

and put the script (*main.js*) in *{appName}/static/* directory.

1. In *main.js* create new field by extending it from BaseField (or any other appropriate field)

   For example lets create a field that renders HTML h1 element with 'Hello World!` text:

```
class CustomField extends spa.fields.base.BaseField {
    static get mixins() {
        return super.mixins.concat({
            render(createElement) {
                return createElement('h1', {}, 'Hello World!');
            },
        });
    }
}
```

Or render person's name with some prefix

```
class CustomField extends spa.fields.base.BaseField {
  static get mixins() {
    return super.mixins.concat({
      render(h) {
        return h("h1", {}, `Mr ${this.$props.data.name}`);
      },
    });
  }
}
```

2. Register this field to *app.fieldsResolver* to provide appropriate field format and type to a new field

```
const customFieldFormat = 'customField';
app.fieldsResolver.registerField('string', customFieldFormat, CustomField);
```

3. Listen for a appropriate *models[ModelWithFieldToChange].fields.beforeInit* signal to change field Format

```
spa.signals.connect(`models[ModelWithFieldToChange].fields.beforeInit`, (fields) => {
    fields.fieldToChange.format = customFieldFormat;
});
```

List of models and their fields is available during runtime in console at *app.modelsClasses*

To change Filed behavior, create new field class with a desired logic. Let's say you need to send number of milliseconds to API, user however wants to type in number of seconds. A solution would be to override field's *toInner* and *toRepresent* methods.

```
class MilliSecondsField extends spa.fields.numbers.integer.IntegerField {
  toInner(data) {
    return super.toInner(data) * 1000;
  }
  toRepresent(data) {
```

(continues on next page)

(continued from previous page)

```
    return super.toRepresent(data)/1000;
  }
}

const milliSecondsFieldFormat = 'milliSeconds'
app.fieldsResolver.registerField('integer', milliSecondsFieldFormat,␣
↪MilliSecondsField);
spa.signals.connect(`models[OneAllFields].fields.beforeInit`, (fields) => {
  fields.integer.format = milliSecondsFieldFormat;
});
```

Now you have field that show seconds, but saves/receives data in milliseconds on detail view of AllFieldsModel.

Note: If you need to show some warning or error to developer console you can use field *warn* and *error* methods. You can pass some message and it will print it with field type, model name and field name.

4.2 Change path to FkField

Sometime you may need to request different set of objects for FkField. For example to choose from only famous authors, create *famous_author* endpoint on backend and set FkField request path to *famous_author*. Listen for *app.beforeInit* signal.

```
spa.signals.connect('app.beforeInit', ({ app }) => {
  app.modelsResolver.get('OnePost').fields.get('author').querysets.get('/post/new/
↪')[0].url = '/famous_author/'
});
```

Now when we create new post on */post/* endpoint Author FkField makes get request to */famous_author/* instead of */author/*. It's useful to get different set of authors (that may have been previously filtered on backend).

4.3 CSS Styling

1. Like scripts, css files may be added to SPA_STATIC in setting.py

```
SPA_STATIC += [

    {'priority': 101, 'type': 'css', 'name': 'style.css', 'source': 'project_lib'},
]
```

Let's inspect page and find css class for our customField. It is *column-format-customField* and generated with *column-format-{Field.format}* pattern.

2. Use regular css styling to change appearance of the field.

```
.column-format-customField:hover {
    background-color: orangered;
    color: white;
}
```

Other page elements are also available for styling: for example, to hide certain column set corresponding field to none.

```
.column-format-customField {
    display: none;
}
```

4.4 Show primary key column on list

Every pk column has *pk-column* CSS class and hidden by default (using *display: none;*).

For example this style will show pk column on all list views of *Order* model:

```
.list-Order .pk-column {
    display: table-cell;
}
```

4.5 View customization

Listen for signal *"allViews.created"* and add new custom mixin to the view.

Next code snippet depicts rendering new view instead of default view.

```
spa.signals.once('allViews.created', ({ views }) => {
    const AuthorListView = views.get('/author/');
    AuthorListView.mixins.push({
        render(h) {
            return h('h1', {}, `Custom view`);
        },
    });
});
```

Learn more about Vue *render()* function at Vue documentation[421].

It is also possible to fine tune View by overriding default computed properties and methods of existing mixins. For example, override breadcrumbs computed property to turn off breadcrumbs on Author list View

```
import { ref } from 'vue';

spa.signals.once("allViews.created", ({ views }) => {
    const AuthorListView = views.get("/author/");
    AuthorListView.extendStore((store) => {
        return {
            ...store,
            breadcrumbs: ref([]),
        };
    });
});
```

Sometimes you may need to hide detail page for some reason, but still want all actions and sublinks to be accessible from list page. To do it you also should listen signal *"allViews.created"* and change parameter *hidden* from default *false* to *true*, for example:

[421] https://v3.vuejs.org/guide/render-function.html

```
spa.signals.once('allViews.created', ({ views }) => {
    const authorView = views.get('/author/{id}/');
    authorView.hidden = true;
});
```

4.6 Changing title of the view

To change title and string displayed in the breadcrumbs change *title* property of the view or method *getTitle* for more complex logic.

```
spa.signals.once('allViews.created', ({ views }) => {
    const usersList = views.get('/user/');
    usersList.title = 'Users list';

    const userDetail = views.get('/user/{id}/');
    userDetail.getTitle = (state) => (state?.instance ? `User: ${state.instance.id}`␣
↪: 'User');
});
```

4.7 Basic Webpack configuration

To use webpack in you project rename *webpack.config.js.default* to *webpack.config.js*. Every project based on vst-utils contains *index.js* in */frontend_src/app/* directory. This file is intended for your code. Run *yarn* command to install all dependencies. Then run *yarn devBuild* from root dir of your project to build static files. Final step is to add built file to *SPA_STATIC* in *settings.py*

```
SPA_STATIC += [
    {'priority': 101, 'type': 'js', 'name': '{AppName}/bundle/app.js', 'source':
↪'project_lib'},
]
```

Webpack configuration file allows to add more static files. In *webpack.config.js* add more entries

```
const config = {
  mode: setMode(),
  entry: {
    'app': entrypoints_dir + "/app/index.js" // default,
    'myapp': entrypoints_dir + "/app/myapp.js" // just added
  },
```

Output files will be built into *frontend_src/{AppName}/static/{AppName}/bundle* directory. Name of output file corresponds to name of entry in *config*. In the example above output files will have names *app.js* and *myapp.js*. Add all of these files to *STATIC_SPA* in *settings.py*. During vstutils installation trough *pip* frontend code are being build automatically, so you may need to add *bundle* directory to *gitignore*.

4.8 Page store

Every page has store that can be accessed globally *app.store.page* or from page component using *this.store*.

View method *extendStore* can be used to add custom logic to page's store.

```
import { computed } from 'vue';

spa.signals.once('allViews.created', ({ views }) => {
    views.get('/user/{id}/').extendStore((store) => {
        // Override title of current page using computed value
        const title = computed(() => `Current page has ${store.instances.hength}␣
↪instances`);

        async function fetchData() {
            await store.fetchData();  // Call original fetchData
            await callSomeExternalApi(store.instances.value);
        }

        return {
            ...store,
            title,
            fetchData,
        };
    });
});
```

4.9 Overriding root component

Root component of the application can be overridden using *app.beforeInit* signal. This can be useful for such things as changing layout CSS classes, back button behaviour or main layout components.

Example of customizing sidebar component:

```
const CustomAppRoot = {
    components: { Sidebar: CustomSidebar },
    mixins: [spa.AppRoot],
};
spa.signals.once('app.beforeInit', ({ app }) => {
    app.appRootComponent = CustomAppRoot;
});
```

4.10 Translating values of fields

Values tha displayed by *FKField* of *ChoicesField* can be translated using standard translations files.

Translation key must be defined as *:model:<ModelName>:<fieldName>:<value>*. For example:

```
TRANSLATION = {
    ':model:Category:name:Category 1': 'Категория 1',
}
```

Translation of values can be taxing as every model on backend usually generates more than one model on frontend, To avoid this, add *_translate_model = 'Category'* attribute to model on backend. It shortens

```
':model:Category:name:Category 1': 'Категория 1',
':model:OneCategory:name:Category 1': 'Категория 1',
':model:CategoryCreate:name:Category 1': 'Категория 1',
```

to

```
':model:Category:name:Category 1': 'Категория 1',
```

For *FKField* name of the related model is used. And *fieldName* should be equal to *viewField*.

4.11 Changing actions or sublinks

Sometimes using only schema for defining actions or sublinks is not enough.

For example we have an action to make user a superuser (*/user/{id}/make_superuser/*) and we want to hide that action if user is already a superuser (*is_superuser* is *true*). *<${PATH}>filterActions* signal can be used to achieve such result.

```
spa.signals.connect('</user/{id}/make_superuser/>filterActions', (obj) => {
    if (obj.data.is_superuser) {
        obj.actions = obj.actions.filter((action) => action.name !== 'make_superuser
↪');
    }
});
```

1. *<${PATH}>filterActions* recieves {actions, data}
2. *<${PATH}>filterSublinks* recieves {sublinks, data}

Data property will contain instance's data. Actions and sublinks properties will contain arrays with default items (not hidden action or sublinks), it can be changed or replaced completely.

4.12 LocalSettings

This model's fields are displayed in the left sidebar. All data from this model saves in browser Local Storage. If you want to add another options, you can do it using *beforeInit* signal, for example:

```
spa.signals.once('models[_LocalSettings].fields.beforeInit', (fields) => {
        const cameraField = new spa.fields.base.BaseField({ name: 'camera' });
        // You can add some logic here
        fields.camera = cameraField;
})
```

4.13 Store

There are three ways to store data:

- userSettingsStore - saves data on the server. By default, there are options for changing language and a button to turn on/off the dark mode. Data to userSettingsStore comes from schema.
- localSettingsStore - saves data in the browser Local Storage. This is where you can store your own fields, as described in *LocalSettings*.

- store - stores current page data.

To use any of this stores you need to run the following command: `app.[storeName]`, for example: `app.userSettingsStore`.

Note: If you are accessing the userSettingsStore from within the component then you need to use `this.$app` instead `app`.

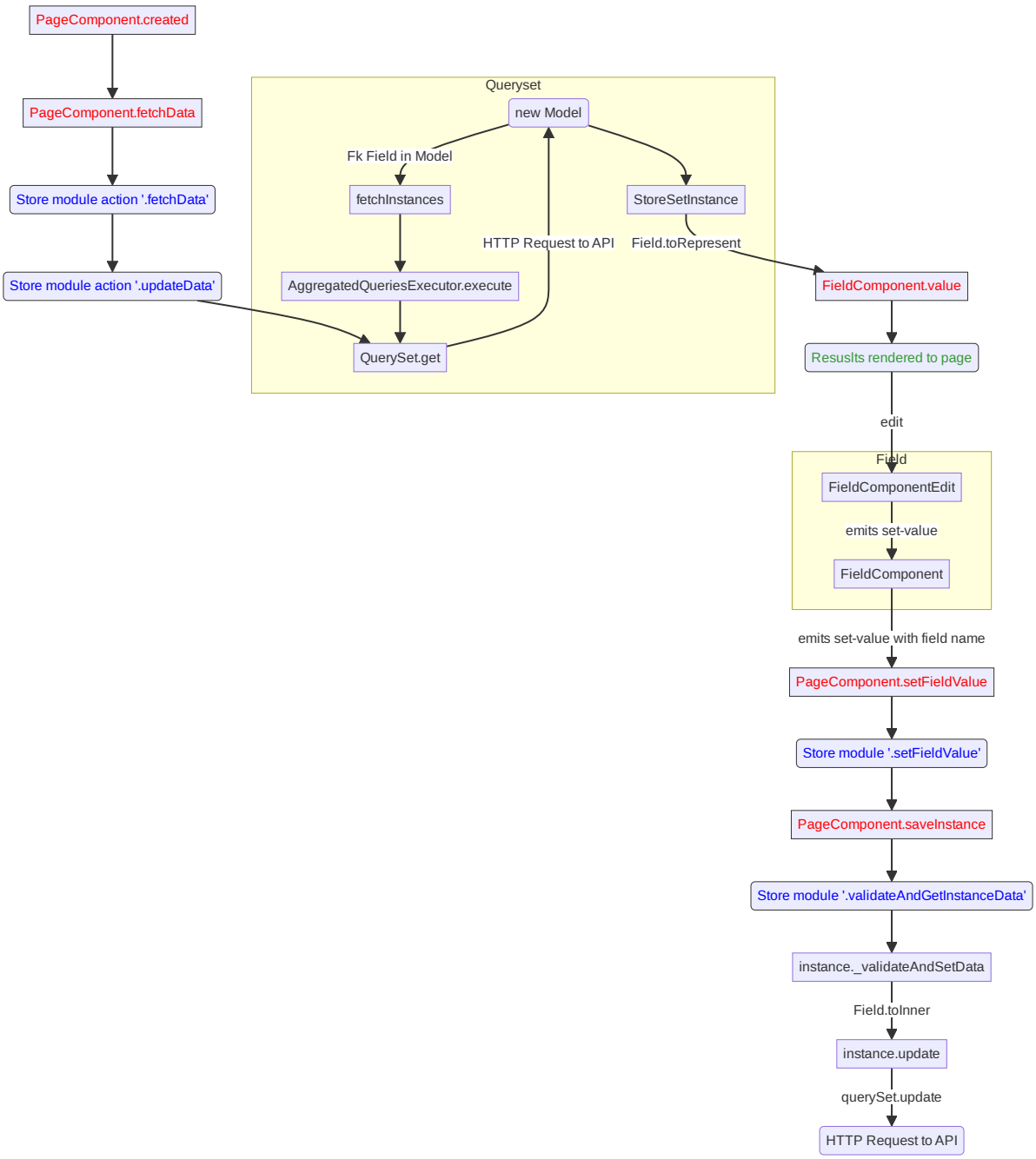
From *app.store* you may need:

- *vewsItems* and *viewItemsMap* - stores information about parent views for this page. It is used for example in breadcrumbs. The difference between them is only in the way information is stored: *viewItems* is an Array of Objects and *viewItemsMap* is a Map.
- *page* - saves all information about current page.
- *title* - title of current page.

5 Frontend documentation

5.1 API Flowchart

This flowchart shows how data goes though application from and to API.



5.2 Signals

System of signals is a mechanism, that VST Utils uses for app customization.

Let's look how it works.

Very often you need to modify something after some event has occurred. But how can you know about this event? And what if you need to know about this event in several blocks of code?

To solve this problem VST Utils uses system of signals, where:

- you can emit some signal, which tells all subscribers, that some event has occurred, and pass some data/variables from the context, where this event has occurred;
- you can subscribe to some signal, that notifies you about some event, and also you can pass some callback (handler) that can do something with data/variables, that were passed from the context, where event had occurred.

5.2.1 Emit signal

To emit some signal you need to write following in you code:

```
tabSignal.emit(name_of_signal, context);
```

where:

- **name_of_signal** - string, which stores name of signal (event);
- **context** - some variable of any type, that will be passed to the callback (handler) during connection to this signal.

Example of signal emitting:

```
let app = {
    name: 'example of app';
};

tabSignal.emit('app.created', app);
```

5.2.2 Connect to signal

To connect to some signal you need to write following in you code:

```
tabSignal.connect(name_of_signal, callback);
```

where:

- **name_of_signal** - string, which stores name of signal (event);
- **callback** - function, that can do something with variables, which will be passed from event's context to this callback as arguments.

Example of connecting to signal:

```
/* ... */
function callback(app) {
    app.title = 'example of app title';
}

tabSignal.connect('app.created', callback);
/* ... */
```

5.3 List of signals in VST Utils

VST Utils has some signals, that are emitting during application work. If you need to customize something in you project you can subscribe to these signals and add callback function with desired behavior. Also you can emit you own signals in your project.

5.3.1 openapi.loaded

Signal name: “openapi.loaded”.

Context argument: openapi - {object} - OpenAPI schema loaded from API.

Description: This signal is emitted after OpenAPI schema was loaded. You can use this signal if you need to change something in the OpenAPI schema, before it was parsed.

5.3.2 resource.loaded

Signal name: “resource.loaded”.

Context argument: None.

Description: This signal is emitted after all static files were successfully loaded and added to the page.

5.3.3 app.version.updated

Signal name: “app.version.updated”.

Context argument: None.

Description: This signal is emitted during app loading if VST Utils detects, that version of your project was updated.

5.3.4 app.beforeInitRouter

Signal name: “app.beforeInitRouter”.

Context argument: obj - {object} - Object with following structure: {routerConstructor: RouterConstructor}, where routerConstructor is an instance of RouterConstructor.

Description: This signal is emitted after creation of RouterConstructor instance and before app creation

5.3.5 app.beforeInit

Signal name: “app.beforeInit”.

Context argument: obj - {object} - Object with following structure: {app: app}, where app is an instance of App class.

Description: This signal is emitted after app variable initialization (OpenAPI schema was parsed, models and views were created), but before app was mounted to the page.

5.3.6 app.afterInit

Signal name: “app.afterInit”.

Context argument: obj - {object} - Object with following structure: {app: app}, where app is an instance of App class.

Description: This signal is emitted after app was mounted to the page.

5.3.7 app.language.changed

Signal name: “app.language.changed”.

Context argument: obj - {object} - Object with following structure: {lang: lang}, where lang is an code of applied language.

Description: This signal is emitted after app interface language was changed.

5.3.8 models[model_name].fields.beforeInit

Signal name: “models[” + model_name + “].fields.beforeInit”. For example, for User model: “models[User].fields.beforeInit”.

Context argument: fields - {object} - Object with pairs of key, value, where key - name of field, value - object with it options. On this moment, field - is just object with options, it is not guiFields instance.

Description: This signal is emitted before creation of guiFields instances for Model fields.

5.3.9 models[model_name].fields.afterInit

Signal name: “models[” + model_name + “].fields.afterInit”. For example, for User model: “models[User].fields.afterInit”.

Context argument: fields - {object} - Object with pairs of key, value, where key - name of field, value - guiFields instance.

Description: This signal is emitted after creation of guiFields instances for Model fields.

5.3.10 models[model_name].created

Signal name: “models[” + model_name + “].created”. For example, for User model: “models[User].created”.

Context argument: obj - {object} - Object with following structure: {model: model}, where model is the created Model.

Description: This signal is emitted after creation of Model object.

5.3.11 allModels.created

Signal name: “allModels.created”.

Context argument: obj - {object} - Object with following structure: {models: models}, where models is the object, storing Models objects.

Description: This signal is emitted after all models were created.

5.3.12 allViews.created

Signal name: “allViews.created”.

Context argument: obj - {object} - Object with following structure: {views: views}, where views - object with all View Instances.

Description: This signal is emitted after creation of all View Instances, with set actions / child_links / multi_actions / operations / sublinks properties.

5.3.13 routes[name].created

Signal name: “routes[” + name + “].created”. For example, for `/user/` view: “routes[/user/].created”.

Context argument: route - {object} - Object with following structure: {name: name, path: path, component: component}, where name - name of route, path - template of route’s path, component - component, that will be rendered for current route.

Description: This signal will be emitted after route was formed and added to routes list.

5.3.14 allRoutes.created

Signal name: “allRoutes.created”.

Context argument: routes - {array} - Array with route objects with following structure: {name: name, path: path, component: component}, where name - name of route, path - template of route’s path, component - component, that will be rendered for current route.

Description: This signal is emitted after all routes was formed and added to routes list.

5.3.15 <${PATH}>filterActions

Signal name: “<${PATH}>filterActions”.

Context argument: obj - {actions: Object[], data} - Actions is array of action objects. Data represents current instance’s data.

Description: This signal will be executed to filter actions.

5.3.16 <${PATH}>filterSublinks

Signal name: "<${PATH}>filterSublinks".

Context argument: obj - {sublinks: Object[], data} - Actions is array of sublink objects. Data represents current instance's data.

Description: This signal will be executed to filter sublinks.

5.4 Field Format

Very often during creation of some new app developers need to make common fields of some base types and formats (string, boolean, number and so on). Create everytime similar functionality is rather boring and ineffective, so we tried ti solve this problem with the help of VST Utils.

VST Utils has set of built-in fields of the most common types and formats, that can be used for different cases. For example, when you need to add some field to you web form, that should hide value of inserted value, just set appropriate field format to `password` instead of `string` to show stars instead of actual characters.

Field classes are used in Model Instances as fields and also are used in Views Instances of `list` type as filters.

All available fields classes are stored in the `guiFields` variable. There are 44 fields formats in VST Utils:

- **base** - base field, from which the most other fields are inherited;
- **string** - string field, for inserting and representation of some short 'string' values;
- **textarea** - string field, for inserting and representation of some long 'string' values;
- **number** - number field, for inserting and representation of 'number' values;
- **integer** - number field, for inserting and representation of values of 'integer' format;
- **int32** - number field, for inserting and representation of values of 'int32' format;
- **int64** - number field, for inserting and representation of values of 'int64' format;
- **double** - number field, for inserting and representation of values of 'double' format;
- **float** - number field, for inserting and representation of values of 'float' format;;
- **boolean** - boolean field, for inserting and representation of 'boolean' values;
- **choices** - string field, with strict set of preset values, user can only choose one of the available value variants;
- **autocomplete** - string field, with set of preset values, user can either choose one of the available value variants or insert his own value;
- **password** - string field, that hides inserted value by '*' symbols;
- **file** - string field, that can read content of the file;
- **secretfile** - string field, that can read content of the file and then hide it from representation;
- **binfile** - string field, that can read content of the file and convert it to the 'base64' format;
- **namedbinfile** - field of JSON format, that takes and returns JSON with 2 properties: name (string) - name of file and content(base64 string) - content of file;
- **namedbinimage** - field of JSON format, that takes and returns JSON with 2 properties: name (string) - name of image and content(base64 string) - content of image;
- **multiplenamedbinfile** - field of JSON format, that takes and returns array with objects, consisting of 2 properties: name (string) - name of file and content(base64 string) - content of file;

- **multiplenamedbinimage** - field of JSON format, that takes and returns array with objects, consisting of 2 properties: name (string) - name of image and content(base64 string) - content of image;
- **text_paragraph** - string field, that is represented as text paragraph (without any inputs);
- **plain_text** - string field, that saves all non-printing characters during representation;
- **html** - string field, that contents different html tags and that renders them during representation;
- **date** - date field, for inserting and representation of 'date' values in 'YYYY-MM-DD' format;
- **date_time** - date field, for inserting and representation of 'date' values in 'YYYY-MM-DD HH:mm' format;
- **uptime** - string field, that converts time duration (amount of seconds) into one of the most appropriate variants (23:59:59 / 01d 00:00:00 / 01m 30d 00:00:00 / 99y 11m 30d 22:23:24) due to the it's value size;
- **time_interval** - number field, that converts time from milliseconds into seconds;
- **crontab** - string field, that has additional form for creation schedule in 'crontab' format;
- **json** - field of JSON format, during representation it uses another guiFields for representation of current field properties;
- **api_object** - field, that is used for representation of some Model Instance from API (value of this field is the whole Model Instance data). This is read only field;
- **fk** - field, that is used for representation of some Model Instance from API (value of this field is the Model Instance Primary Key). During edit mode this field has strict set of preset values to choose;
- **fk_autocomplete** - field, that is used for representation of some Model Instance from API (value of this field is the Model Instance Primary Key or some string). During edit mode user can either choose of the preset values from autocomplete list or insert his own value;
- **fk_multi_autocomplete** - field, that is used for representation of some Model Instance from API (value of this field is the Model Instance Primary Key or some string). During edit mode user can either choose of the preset values from modal window or insert his own value;
- **color** - string field, that stores HEX code of selected color;
- **inner_api_object** - field, that is linked to the fields of another model;
- **api_data** - field for representing some data from API;
- **dynamic** - field, that can change its format depending on the values of surrounding fields;
- **hidden** - field, that is hidden from representation;
- **form** - field, that combines several other fields and stores those values as one JSON, where key - name of form field, value - value of form field;
- **button** - special field for form field, imitates button in form;
- **string_array** - field, that converts array with strings into one string;
- **string_id** - string field, that is supposed to be used in URLs as 'id' key. It has additional validation, that checks, that field's value is not equal to some other URL keys (new/ edit/ remove).

5.5 Layout customization with CSS

If you need to customize elements with css we have some functionality for it. There are classes applied to root elements of `EntityView` (if it contains *ModelField*), `ModelField`, `ListTableRow` and `MultiActions` depending on the fields they contain. Classes are formed for the fields with "boolean" and "choices" types. Also classes apply to operations buttons and links.

Classes generation rules

- `EntityView, ModelField and ListTableRow` - *field-[field_name]-[field-value]*

  Example:

  - *"field-active-true"* for model that contains "boolean" field with name "active" and value "true"
  - *"field-tariff_type-WAREHOUSE"* for model that contains "choices" field with name "tariff_type" and value "WAREHOUSE"

- **`MultiActions`** - ***selected__field-[field_name]-[field-value]***

  Example:
  "selected__field-tariff_type-WAREHOUSE" and *"selected__field-tariff_type-SLIDE"* if selected 2 `ListTableRow` that contains "choices" field with name "tariff_type" and values "WAREHOUSE" and "SLIDE" respectively.

- **`Operation`** - ***operation__[operation_name]***

  Warning
  If you hide operations using CSS classes and for example all actions were hidden then Actions dropdown button will still be visible.

  For better control over actions and sublinks see *Changing actions or sublinks*

  Example:
  operation__pickup_point if operation button or link has name *pickup_point*

Based on these classes, you can change the styles of various elements.

A few use cases:

- If you need to hide the button for the "change_category" action on a product detail view when product is not "active", you can do so by adding a CSS selector:

```
.field-status-true .operation__change_category {
    display: none;
}
```

- Hide the button for the "remove" action in `MultiActions` menu if selected at least one product with status "active":

```
.selected__field-status-true .operation__remove {
    display: none;
}
```

- If you need to change *background-color* to red for order with status "CANCELLED" on `ListView` component do this:

```
.item-row.field-status-CANCELLED {
    background-color: red;
}
```

In this case, you need to use the extra class “item-row” (Used for example, you can choose another one) for specify the element to be selected in the selector, because the class “field-status-CANCELLED” is added in different places on the page.

Python Module Index

V

Index

Q

R

S

T

U

V

W